UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

FUZZ TESTING APLICADO EN MÁQUINAS DE ESTADO JERÁRQUICAS DE SMACH PARA PRUEBAS DE COMPORTAMIENTOS ROBÓTICOS

TESIS PARA OPTAR AL GRADO DE
MAGÍSTER EN CIENCIAS, MENCIÓN COMPUTACIÓN

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

RODRIGO ALEXIS DELGADO VEGA

PROFESOR GUÍA:
ALEXANDRE BERGEL

PROFESOR CO-GUÍA:
MIGUEL CAMPUSANO ARAYA

MIEMBROS DE LA COMISIÓN:
ANDRÉS MUÑOZ ORDENES
ALCIDES QUISPE SANCA
ÉRIC TANTER

SANTIAGO DE CHILE
2022

Resumen

La robótica como disciplina ha aumentando considerablemente su desarrollo en la ultima década. Una de las principales razones de este aumento han sido los bajos costos de ventas y su uso en la vida diaria. Este aumento de uso también fomentó la creación de robots con mayores niveles de autonomía, haciéndolos cada vez mas independientes de las acciones de usuarios u operadores. Al minimizar la supervisión del robot mientras realiza sus comportamiento, se aumenta el riesgo de generar situaciones de peligro, ya sea para el robot o los usuarios. Ejecutar un comportamiento robótico no es una tarea instantánea, esta puede tomar un tiempo considerable dependiendo de la cantidad de operaciones que realice el robot. Estas operaciones podrían conllevar una situación de riesgo, por ello es necesario realizar pruebas para comprobar que el comportamiento robótico es seguro y libre de errores en diferentes escenarios. Realizar pruebas a un comportamiento robótico con un robot real es una tarea que consume tiempo y es potencialmente peligrosa, ya que se debe ejecutar el comportamiento robótico, por lo que el uso de simuladores es una practica común para realizar pruebas tempranas del comportamiento y asegurar que el robot se comportara bien en casos ideales.

Este trabajo de tesis propone una metodología que permite encontrar errores en fases tempranas del desarrollo de un comportamiento robótico, utilizando técnicas de *fuzz testing* para generar variados contextos de pruebas para los comportamientos. La metodología permite ir ajustando manualmente el contexto para poder generar datos cada vez mas ajustados a contextos reales. Estas pruebas son realizadas al modelo de máquina de estados que representa a un comportamiento robótico y las pruebas son realizadas a cada estado por separado para comprobar su correctitud individualmente. Esta metodología también permite realizar las pruebas sin tener un conocimiento acabado del comportamiento, solo basta con realizar inspecciones visuales al código para extraer la información necesaria.

Para desarrollar este trabajo se utilizó el software del equipo de robótica de la Universidad de Chile, UChile Peppers como fuente de pruebas para evaluar esta metodología, encontrando situaciones donde los comportamientos generaban errores o situaciones no esperadas. A través de un cuestionario se permitió obtener información sobre el error encontrado, aceptándolo o descartándolo como error por parte del mismo equipo de desarrollo del software.

Por último, un análisis detallado de las respuestas del cuestionario aplicado junto a apreciaciones del autor de este trabajo indican que el uso de *fuzz testing* en comportamientos robóticos es una herramienta útil que permite encontrar errores que pueden generarse en una ejecución del comportamiento robótico

Dedicada a mi padre y mi familia.

Agradecimientos

En primer lugar a mi señora Darling que me ha acompañado durante todo este largo viaje, muchas gracias por tu amor, comprensión, apoyo y cuidados.

A mi madre Elena y mis hermanos Ariel y Nicolás por el apoyo incondicional, la motivación a seguir estudiando y el apoyo económico que supuso esta aventura.

A mis profesores guías Miguel y Alexandre, por apoyarme en este camino incierto de la ingeniería de software en robótica e impulsarme a ir siempre mas allá.

Al equipo de robótica de la Universidad de Chile por su apoyo en esta tesis y por su amistad, dejando siempre un espacio para la distensión en cada visita al laboratorio y donando su tiempo desinteresadamente.

A la comisión revisora, Profesores Eric, Alcides y Andres, por hacerme cuestionar de buena manera el contenido y la redacción de este trabajo de tesis para hacerlo siempre mejor.

A mis amigos del DCC, Nicolás, Joaquín, Javier, Belisario, Juan, Sergio, Américo y Gabriel por la compañía y amistad a pesar de mi poca disponibilidad para todo.

A la corporación CUAC por el apañe en los años de trabajo que llevamos y la fraternidad a pesar de la distancia que existe.

A todas las personas que se cruzaron en este largo camino tanto en la Universidad de Chile como en la Universidad de O'Higgins que no están nombradas explícitamente, son tantas personas que no caben en una hoja.

Por último, a mi padre que siempre me impulsó a dar lo mejor de mi y estaría felíz de ver el resultado.

Tabla de Contenido

1. Introducción **1**
1.1. Motivación . . . 2
1.2. Hipótesis . . . 3
1.3. Objetivos . . . 4
1.4. Metodología . . . 5
1.5. Resultados . . . 6
1.6. Estructura de la tesis . . . 7

2. Marco Teórico **8**
2.1. Modelar comportamientos robóticos . . . 8
2.1.1. Conceptos de máquinas de estado . . . 9
2.1.2. Modelar una NSM . . . 10
2.1.3. Software para Robots . . . 12
2.2. Fuzz testing . . . 13
2.2.1. Generación de entradas . . . 14
2.2.2. Estructura de la entrada . . . 15
2.2.3. Estructura del programa . . . 16
2.2.4. El fuzzer utilizado . . . 17

3. Trabajo Relacionado **19**
3.1. Fuzz testing . . . 19
3.2. Pruebas en robots . . . 20
3.3. Fuzz testing en robots . . . 21

4. Pruebas en comportamientos robóticos **22**
4.1. Pruebas en comportamientos como máquinas de estado . . . 22
4.2. Pruebas de estados aislados . . . 25
4.3. Pruebas aleatorias en estados . . . 26
4.4. El Fuzzer . . . 30
4.4.1. Pruebas a nivel de estado . . . 30
4.4.2. Pruebas a nivel de máquina . . . 33

5. Experimentos **34**
5.1. Contexto del experimento . . . 34
5.2. Metodología . . . 35
5.3. Reporte al equipo . . . 39

6. Análisis Experimentos 40
6.1. Datos del experimento . . . 40
6.2. Caracterización del error . . . 41
6.3. Errores reales . . . 44
6.4. Dificultad en encontrar los errores . . . 45
6.5. Validez del experimento . . . 47
6.5.1. Validez interna . . . 47
6.5.2. Validez externa . . . 47

7. Conclusión 49
7.1. Contribuciones . . . 49
7.2. Trabajo futuro . . . 50
7.2.1. Otros tipos de fuzzer . . . 50
7.2.2. Gramáticas probabilísticas . . . 51
7.2.3. Fuzzer generados por aprendizaje de máquinas . . . 51
7.2.4. Automatización en la detección de tipos de datos . . . 52
7.2.5. Probar partes de la máquina de estados . . . 52

8. Bibliografía 53

Capítulo 1

Introducción

El uso de robots en diferentes lugares como hospitales, escuelas, industrias e incluso el hogar es una realidad que comienza a aumentar significativamente cada año que pasa. Cada vez la autonomía de estos sistemas aumenta más y más, llegando incluso a tener los primeros vehículos motorizados con un nivel de autonomía que permiten evitar accidentes simples por culpa de otros conductores. Estas mejoras en la autonomía suponen una mejor situación para los usuarios al tener que interactuar cada vez menos con los robots para realizar tareas, pero no representan necesariamente una mejor situación para quienes trabajan en robótica.

A diferencia de los robots operados a distancia, los robots autónomos deben tomar decisiones sin supervisión del usuario o de un operador, generando situaciones potencialmente peligrosas para el robot o los usuarios de éste. El desarrollo de autonomía supone que el robot debe probarse en innumerables situaciones y contextos diferentes para asegurar que esta autonomía no representa una situación de peligro para sus usuarios. Recientes estudios [1] muestran como profesionales de la industria robótica realizan pruebas en sus sistemas, los principales desafíos y dificultades que conlleva realizar pruebas.

Dado que ejecutar una prueba del robot representa una potencial situación de peligro, generalmente se utilizan simuladores para realizar pruebas [2]. Estas pruebas en simuladores permiten eliminar el posible peligro, pero limitan el comportamiento del robot al no representar completamente el mundo real. El beneficio que generan los simuladores los hace ideales para crear y probar hasta etapas avanzadas del desarrollo, reduciendo los costos en construcción del robot y el riesgo de las situaciones de peligro.

Otro de los costos asociados a las pruebas en robots es el tiempo que puedan tomar en realizarse. La ejecución de un comportamiento puede tardar desde pocos segundos a varios minutos e incluso horas, por lo que ejecutar una y otra vez pruebas en ellos se hace una tarea costosa en tiempo. Para poder encontrar un problema en la ejecución de un comportamiento se requiere de tiempo y conocimiento del comportamiento, es decir, es necesario saber que tiene que hacer el robot según las condiciones actuales y compararlo con el comportamiento mostrado.

Encontrar errores simples, como errores de lógica, provocados por el código fuente en una

ejecución de un comportamiento consume una cantidad considerable de tiempo al tener que ejecutar el comportamiento hasta que se genere el error. Resolverlo y luego comprobar la corrección del error forzará nuevamente la ejecución del comportamiento hasta el punto del error, por lo que se hace útil poseer una herramienta que permita identificar y corregir este tipo de errores antes de ejecutar el comportamiento. Si bien tener una aplicación libre de errores no es posible, cualquier herramienta que ayude a disminuirlos, reducirá el tiempo de desarrollo al invertir una menor cantidad de tiempo en encontrar y solucionar estos errores.

Este trabajo de tesis presenta una metodología que permite encontrar errores en fases tempranas del desarrollo de comportamientos robóticos utilizando *fuzz testing* para generar variados contextos de pruebas para los comportamientos. Estas pruebas se realizan al modelar el comportamiento robótico como una máquina de estados y se le realizan pruebas a cada uno de los estados para comprobar su correctitud por separado, confiando en que la conexión entre los estados está bien realizada.

1.1. Motivación

Cada vez se ven más robots que cumplen tareas diarias a menores costos, incluso compitiendo con las herramientas de uso no autónomo para las mismas tareas. También se ven más robots en industrias colaborando con personas ya sea para realizar tareas sencillas como robots asistentes en tiendas comerciales o tareas complejas como manejar una *warehouse*[1]. Una de las principales problemáticas que genera este aumento en el uso de robots es la posibilidad de que sean utilizados en situaciones o escenarios que no son buenos para el funcionamiento del robot e incluso en situaciones donde el fabricante desconoce cómo se comportará, generando un peligro para el entorno.

Para evitar estas posibles fallas o al menos conocerlas, se necesitan realizar pruebas para cada caso específico, si no se realizan pruebas, es imposible asegurar un buen funcionamiento. Aun así, no es posible probar todos los casos posibles. Por esto es necesario generar pruebas para robots en contextos aleatorios y que se puedan replicar las veces que sean necesarias para asegurar que funciona en ese contexto generado. A mayor cantidad de contextos distintos, mejor será la confiabilidad en el sistema y menos riesgos generará en situaciones similares a las probadas.

Dentro de la robótica existen diferentes tipos de robots que se pueden clasificar en diferentes categorías: robótica industrial, robótica médica, robótica militar y robótica de servicio, etc. Según la ISO[2], un robot de servicio se define como un robot "*que realiza tareas útiles para humanos o equipos, excluidas las aplicaciones de automatización industrial*"(traducción de la cita original en ingles), los casos más comunes de este tipo de robots son las aspiradoras robóticas. En la Universidad de Chile existe un equipo de robótica de servicio llamado UChile Peppers que compite a nivel internacional en la RoboCup@Home, una competencia que permite a equipos de diferentes Universidades mostrar sus avances en robótica de servicio en diferentes escenarios.

[1]https://www.wired.com/story/amazon-warehouse-robots
[2]International Standard Organization https://committee.iso.org/home/tc299

El equipo UChile Peppers desarrolla su software para el robot Pepper[3] de la empresa SoftBank, que es un modelo diseñado para la interacción con humanos. El foco de la competencia RoboCup es mostrar comportamientos en entornos reales de hogares y pasar una serie de tareas que debe realizar el robot en cada etapa. Cada tarea está constituida por diferentes pasos a seguir, por ejemplo, la competencia del 2021, que se realizó de manera virtual, define la primera etapa de limpieza como:

> El robot entra a una habitación predefinida en el escenario y comienza a buscar objetos que no están bien ubicados (Todos los objetos conocidos tienen una categoría y un lugar asignado). Los objetos conocidos y extraviados deben devolverse a sus ubicaciones predefinidas, mientras que los objetos desconocidos se consideran basura y deben desecharse en el basurero.

El puntaje de esta tarea se calculó como la suma de los objetos correctamente devueltos a sus posiciones, menos una penalización por cada objeto devuelto erróneamente, incluyendo objetos puestos fuera del basurero.

Equivocarse en un paso de la tarea genera un impacto negativo en el puntaje, pero que puede compensarse con los demás pasos, en el caso de ejemplo, si equivoca en poner un objeto, el siguiente objeto puede compensar ese error. Si el software del robot genera un error inesperado y no continúa realizando los pasos, el puntaje obtenido hasta ese punto es el puntaje de la etapa, por lo que es vital que el robot ejecute la mayor cantidad de pasos por cada tarea sin que se genere un error de código inesperado. Si bien el equipo es libre de probar el robot antes de la competencia, las condiciones específicas de la competencia no son conocidas, por lo que no es posible generar soluciones particulares a los problemas si no que soluciones generales y probar estas soluciones en diferentes contextos para asegurar su correcto funcionamiento.

Una forma de probar múltiples contextos diferentes es utilizar una simulación y generar diferentes ambientes de pruebas. Esto puede resultar beneficioso, pero a la vez consume tiempo en generar la prueba y crear un nuevo ambiente. En este trabajo de tesis se propone un método de generación de contextos diferentes utilizando la técnica de *fuzz testing* que permite crear datos bien definidos de manera artificial para simular los diferentes contextos y estresar el comportamiento robótico al ejecutarlo bajo diferentes circunstancias.

1.2. Hipótesis

Este trabajo de tesis busca validar o refutar la siguiente hipótesis de investigación:

"El uso de *fuzz testing* basado en gramáticas puede identificar automáticamente errores no triviales en el código de comportamientos robóticos modelados como máquinas de estado jerárquicas". Donde se entiende como "errores no triviales" a situaciones donde las configuraciones de laboratorio y las pruebas que se realizaron no fueron lo suficientemente profundas o específicas para poder detectarlas.

[3]https://www.softbankrobotics.com/emea/en/pepper

Para verificar o refutar esta hipótesis se formularon las siguientes preguntas de investigación:

(**PI1**) *¿Cuáles son las características de los errores identificados en un comportamiento robótico utilizando fuzz testing?* Esta pregunta busca identificar qué tipo de errores y categorías de errores pueden identificarse utilizando un *fuzzer* en software de comportamientos robóticos modelados como máquinas de estado jerárquicas.

(**PI2**) *¿Puede el fuzz testing detectar problemas en comportamientos robóticos que sean realista y representativos?* Esta pregunta hace relación al potencial de la técnica de *fuzz testing* para identificar errores reales y errores con condiciones de laboratorio, es decir, errores que son poco o nada reproducibles en ambientes reales.

(**PI3**) *¿Puede el fuzz testing detectar problemas difíciles de encontrar?* Con esta pregunta se busca analizar la complejidad de los errores encontrados. En este trabajo se utiliza la percepción del equipo de desarrollo del software probado para medir la complejidad de los errores.

1.3. Objetivos

Este trabajo de tesis se desarrolló para aumentar la fiabilidad en los sistemas robóticos basados en comportamientos, especialmente para dotar a desarrolladores de herramientas y metodologías que permitan generar un comportamiento que responderá como se espera bajo las circunstancias probadas. Además de permitir generar casos de pruebas que podrían ser difíciles de generar en un escenario real pero probables dentro de la ejecución.

El objetivo principal de este trabajo es desarrollar una metodología de trabajo para poder realizar pruebas con un *fuzzer* que permiten generar diferentes escenarios de prueba.

Para ello se llevaron a cabo los siguientes objetivos específicos:

- Crear un *fuzzer* capaz de ejecutar un comportamiento robótico modelado como una máquina de estados jerárquica.
- Diseñar una metodología para obtener los datos utilizados por el *fuzzer* de manera reproducible.
- Diseñar un ciclo de trabajo para mejorar manualmente la información que utiliza el *fuzzer*.
- Identificar errores en los resultados obtenidos.
- Generar reportes para revisar con el equipo de desarrollo.
- Identificar posibles causas que generen estos errores.

1.4. Metodología

La metodología aplicada en este trabajo de tesis se basó en la creación y ejecución de un *fuzzer* para obtener errores siguiendo una metodología estructurada, luego se analizaron esos errores e informaron al equipo de desarrollo para evaluar la eficacia de la herramienta y metodología construida a través de un cuestionario construido para recibir las apreciaciones.

El trabajo se realizó secuencialmente de la siguiente manera:

1. Revisión bibliográfica de técnicas de *fuzz testing*.
2. Solicitud de colaboración al equipo UChile Peppers.
3. Inspección visual del código de algunos comportamientos robóticos del equipo UChile Peppers.
4. Selección del tipo de *fuzzer*.
5. Ejecución de los diferentes comportamientos con el *fuzzer*.
6. Análisis de los resultados y separación de los casos de error o mal funcionamiento del comportamiento.
7. Creación del reporte por cada error encontrado.
8. Revisión de los reportes con miembros del equipo UChile Peppers.

La revisión bibliográfica sobre *fuzz testing* se realizó, en un principio, con el libro *FuzzingBook* [3]. Posteriormente se revisaron otras fuentes y dio paso al trabajo revisado en la Sección 2.2.

La colaboración con el equipo UChile Peppers se dio a través de permisos para inspeccionar y revisar el código fuente de su trabajo. Al ser un equipo que compite en la RoboCup, el código de sus comportamientos lo mantienen en repositorios privados. También se incluyeron visitas informales a su lugar de trabajo dada la colaboración en otros proyectos ajenos al proyecto de tesis con algunos de los miembros del equipo. Estas visitas permitieron conocer mejor el contexto de desarrollo y como se organiza el equipo para programar los comportamientos robóticos.

La inspección visual del código fue una parte fundamental del trabajo, ya que permitió conocer y comprender como se modela un comportamiento robótico a través de máquinas de estados, mostrando los parámetros necesarios para su funcionamiento y como se generan los datos utilizados en un comportamiento robótico. También permitió escoger el lenguaje de programación y las librerías a utilizar, enfocando el trabajo en un caso práctico con resultados medibles.

La elección del *fuzzer* a utilizar se basó en el resultado de la inspección visual, al encontrar un lenguaje dinámicamente tipado, lo mejor para evitar problemas triviales de tipos fue generar un *fuzzer* que obligue a generar entradas de algún tipo de dato específico o dentro

de una colección de posibilidades, para reducir estos errores triviales que no aportan en el objetivo de encontrar errores en el comportamiento robótico.

Al momento de ejecutar el *fuzzer* en los diferentes comportamientos se dio una situación particular, la mayoría de las máquinas de estados no recibían entradas al inicializarlas, por lo que todos los datos obtenidos eran generados en los estados y se enviaba información a través de ellos mismos. Estas ejecuciones permitieron detectar tempranamente este comportamiento y tomar la decisión de diseccionar la máquina de estados en cada uno de sus estados para realizar las pruebas, permitiendo mayor profundidad en las pruebas realizadas en contraste con probar la máquina de estados una y otra vez.

Dada la complejidad que podían llegar a generar las máquinas de estados al poder anidarse, fue necesario realizar un chequeo a mano de los resultados. Como se explica en la Sección 6.2, algunos errores no necesariamente generan un error en el lenguaje, sino que pueden generar un comportamiento diferente del esperado. Este tipo de errores es importante capturarlos ya que no detienen la ejecución del robot, sino que lo hacen realizar tareas que no son correctas. Por lo tanto, es vital entender cuando un comportamiento está mal ejecutado con respecto a los parámetros dados.

Encontrar un error no es suficiente, si no se reporta o soluciona es posible que vuelva a aparecer y se deba repetir el proceso para verificarlo. Los reportes creados en este trabajo consisten en identificar el nombre del comportamiento, el estado donde se genera, la información utilizada por el estado y el error que se genera en caso de existir.

Por último, en este experimento era necesario validar si el error considerado por el autor es un error según sus desarrolladores o no, por ejemplo, podría esperarse que bajo ciertos valores el comportamiento falle con un error propio del lenguaje.

1.5. Resultados

La metodología antes presentada generó una metodología para realizar pruebas a comportamientos robóticos propuesta en este trabajo de tesis explicada en la Sección 5.2, donde cada paso fue realizado por el autor para luego replicarlo. Adicionalmente se presenta un cuestionario que permite evaluar la importancia de las situaciones encontradas por el mismo equipo de desarrollo.

El desarrollo del *fuzzer* generó un repositorio de código abierto [4] que permite la ejecución de comportamientos a través de *fuzz testing*. Es necesario ajustar los valores descritos en el mismo repositorio, por lo que se necesita tener un ambiente bien configurado si se quiere probar con un robot simulado.

Este trabajo genero una publicación aceptada en la conferencia *International Conference on Robotics and Automation 2021* (ICRA 2021[5]), una de las conferencias más importantes de robótica a nivel mundial. La publicación lleva el título *Fuzz testing in behavior-based robotics*.

[4] https://github.com/rdelgadov/fuzz_testing
[5] www.icra2021.org

1.6. Estructura de la tesis

Este trabajo de tesis presenta un primer capítulo (Capítulo 2) donde se explican de manera detallada los elementos para entender el trabajo desarrollado, se habla sobre los comportamientos robóticos, como modelarlos y se explican algunos conceptos claves de SMACH, una librería que permite escribir comportamientos robóticos escrita en Python. También se presenta el concepto de *fuzz testing*, los diferentes *fuzzer* que pueden ser construidos y como se pueden clasificar.

En el Capítulo 3 se presenta el trabajo relacionado con este trabajo de tesis, comenzando con diferentes *fuzzer* utilizados en variados contextos y sus beneficios a la hora de probar aplicaciones. Luego se muestran como algunos autores han abordado el concepto de pruebas en robots de diferentes maneras. Por último, se muestran trabajos que utilizan *fuzz testing* para realizar pruebas enfocadas en robots.

Para el Capítulo 4 se abordan las pruebas en comportamientos robóticos, se explica en mayor detalle como nace la idea de realizar pruebas a nivel de estado aislado. Se habla sobre el *fuzzer* construido y los beneficios de utilizarlo para este contexto.

En el Capítulo 5 se explica el experimento realizado con el equipo de robótica UChile Peppers y como se obtuvieron los errores al utilizar el *fuzzer* en el código fuente de su robot. Se presenta la metodología para realizar este tipo de pruebas con una descripción del paso a paso.

El Capítulo 6 presenta el análisis de los resultados obtenidos junto al equipo de robótica, en él se responden las preguntas de investigación presentadas en base a las respuestas de un cuestionario que cada miembro del equipo respondió.

Por último, se presentan las conclusiones en el Capítulo 7 de este trabajo, las contribuciones que se realizaron y trabajos futuros que podrían extender este trabajo de tesis para seguir mejorando la calidad y fiabilidad del software de comportamientos robóticos.

Capítulo 2

Marco Teórico

En este capítulo se detallan los conceptos principales utilizados en este trabajo de tesis que se centra en presentar una metodología para utilizar la técnica de *fuzz testing* basado en gramáticas en máquinas de estado que representan comportamientos robóticos. Se presenta una base sobre comportamientos robóticos, diferentes formas de modelarlos y una manera de implementarlos utilizando el *middleware* ROS con la librería SMACH. Luego se presenta una técnica de pruebas automatizadas llamada *fuzz testing*. Se detalla el uso de ella con la generación de datos utilizando gramáticas para generar datos teóricamente válidos.

2.1. Modelar comportamientos robóticos

Un comportamiento robótico es la forma en que un robot actuará frente a ciertas variaciones en el mundo real. Cuando se habla de robótica basado en el comportamiento, se refiere a la creación de una inteligencia que intenta ir respondiendo a múltiples cambios o estímulos del mundo real sobre el robot [4]. Esta inteligencia recibe los estímulos a través de sus sensores y responde utilizando sus actuadores.

Los comportamientos robóticos pueden ser simples o complejos. Un comportamiento robótico simple es aquel que busca un propósito casi instintivo o reflejo en el robot, mientras que un comportamiento complejo es aquel que depende de algunos comportamientos simples y los utiliza para lograr su objetivo.

Por ejemplo, la navegación es una tarea ampliamente estudiada en robótica. Navegar para un robot es ir desde un punto A hacia un punto B, pero no de manera recta ya que eso sería un comportamiento **simple** de desplazamiento. La navegación en un robot es un comportamiento **complejo** que involucra pasos como planificar que camino tengo que tomar para ir desde A hacia B dependiendo del estado del entorno y desplazarse según esa planificación. Es posible que el robot no conozca todo lo que existe entre A y B, por lo que la planificación y el desplazamiento se deben ir ajustando en cada cambio que afecte la trayectoria planificada del robot.

El movimiento de un robot no siempre es preciso, incluso en robots industriales donde asegurar la precisión es algo crucial [5]. Si nos basamos solo en el trayecto entre A y B es

altamente probable perder el camino real a medida que se avanza.

En un robot basado en comportamientos la navegación se realiza constantemente, reaccionando a los cambios del mundo real y los cambios que presenta el robot. En la navegación es normal ver una rectificación del camino a seguir cada cierta cantidad de tiempo o cuando el ambiente varía mucho con respecto a las mediciones anteriores. Si un nuevo obstáculo es añadido, no necesariamente se debe cambiar la planificación, ya que este podría no interferir con el desplazamiento del robot y no tendrá ningún impacto en su comportamiento. Por el contrario, si el obstáculo interfiere con la planificación actual, es necesario volver a planificar tomando en cuenta este nuevo escenario. Si la decisión de que paso dar dependiera totalmente de las mediciones, cada paso podría estar variando mucho la trayectoria entre los dos puntos, más aún, quizás nunca podría avanzar por las infinitas formas de ir entre A y B.

Al actuar en base a los estímulos del entorno es probable que el camino que tome el robot no sea el óptimo ni mucho menos el más corto, ya que puede variar en cada movimiento que realiza.

Crear un comportamiento robótico de este estilo no es una tarea sencilla considerando que este debe ser robusto frente a diversas situaciones y responder de manera adecuada a cada una de ellas. En general, los comportamientos robóticos de bajo nivel o reflejos están bien estudiados por ser los más utilizados y probados en el campo de la robótica. Al intentar desarrollar un nuevo comportamiento utilizando estos comportamientos base, la capacidad de reacción del robot se puede entorpecer por la cantidad de cálculos que debe realizar al mismo tiempo o la cantidad de posibles acciones que puede tomar en base a estos cálculos.

El comportamiento del robot será una reproducción fiel de lo programado. El robot no tomará decisiones que no estén programadas y actuará en base a la información disponible según su programa y sensores. Uno de los objetivos de este trabajo de tesis es identificar cómo el cambio de valores en la información que el robot recibe o calcula afecta a los comportamientos, sin la necesidad de ejecutar este comportamiento en la vida real, lo que permite detectar posibles errores en la programación antes de que ocurran en un ambiente real o evitar situaciones donde el robot podría fallar dada las condiciones del entorno.

Para poder escribir un comportamiento robótico es necesario primero modelarlo de manera concreta. Existen diferentes formas de modelar un comportamiento robótico, entre ellas están los *statecharts* [6], las redes bayesianas [7] y las máquinas de estado anidadas (NSM por sus siglas en ingles), que son las que se utilizan en este trabajo de tesis. Las NSM son máquinas de estado que pueden contener máquinas de estado en sus nodos. Esta propiedad permite componer máquinas de estados sin la necesidad de modificar la máquina que se quiere agregar, solo basta con conectar la máquina de estados a anidar a la salida de algún nodo de la máquina de estados principal. Este trabajo está pensado y probado para trabajar con NSM ya que son ampliamente usadas en robótica [8, 9, 10].

2.1.1. Conceptos de máquinas de estado

Cuando se habla sobre máquinas de estados, ya sean finitas o anidadas, se hace una correlación natural con el concepto de autómata finito determinista, el cual está definido formalmente por la tupla $(Q,\sum,q_0,\delta,F)$ donde:

- Q: Conjunto finito de estados.
- $\sum$: Alfabeto.
- $q_0 \in Q$: Estado inicial perteneciente a Q.
- δ: $Q \times \sum \to Q$: Función de transición.
- $F \subseteq Q$: Subconjunto de aceptación o estados finales.

Este autómata finito parte su ejecución en el estado inicial q_0. La función de transición indica si dos estados están conectados. Las transiciones son seleccionadas dependiendo de los símbolos de entrada definidos por el alfabeto $\sum$. El estado cambia al recibir un símbolo y posee una transición del estado actual con el símbolo recibido. La ejecución termina de manera exitosa cuando se entrega una cadena finita de símbolos y el ultimo estado es un estado final del conjunto F, sino se llega a una ejecución errónea.

Llevando esto al contexto de comportamientos robóticos, cada estado $q \in Q$ representa un comportamiento del robot. El estado inicial q_0 es el comportamiento inicial del robot. Cada transición representa un cambio en el comportamiento del robot. El alfabeto $\sum$ se puede asociar a los diferentes eventos que el robot puede detectar. Las transiciones estarán ligadas a estos eventos y los estados correspondientes. Para cambiar entre los diferentes comportamientos es necesario que exista una transición entre ellos y que el evento necesario para que se realice sea detectado por el robot. Cuando se realizará el cambio entre un estado y otro dependerá en su totalidad de la implementación de máquinas de estado que se utiliza.

2.1.2. Modelar una NSM

Para modelar una NSM existen diferentes *frameworks*/librerías que se han desarrollado especialmente para trabajar con robots, entre ellas están SMACH [11], FlexBe [12] y la más reciente SMACC[1]. Al comienzo de este trabajo la librería más utilizada para trabajar con comportamientos robóticos era SMACH. Con el paso del tiempo FlexBe ha ido aumentando sus usuarios. pero cabe mencionar que FlexBe utiliza como base de sus máquinas de estado las máquinas de estado de SMACH, pero aporta un entorno de desarrollo de comportamientos mucho más simple que con SMACH. Por su parte, SMACC es más reciente y está inspirada en los *statecharts* aunque aclaran que también tomaron inspiración en SMACH para crearla.

SMACH. SMACH es una librería para crear máquinas de estados jerárquicas construida de manera independiente a algún *framework* de robótica, pero con una fuerte conexión con ROS y escrita en Python.

La librería de SMACH está compuesta de 4 elementos claves para el desarrollo de este trabajo de tesis y que se definen a continuación:

- Estados.
- Contenedores.

[1] https://github.com/reelrbtx/SMACC

- Userdata.
- Máquinas de estados.

Estados. SMACH presenta los estados como el tipo de dato base de su librería. Un estado es una interfaz que posee:

- Una lista de *String* llamada *outcomes* que representa las posibles salidas del estado.
- Una lista de *String* llamada *input keys* que representa las llaves que el estado puede **leer y usar** desde un *Userdata* en tiempo de ejecución.
- Una lista de *String* llamada *output keys* que representa las llaves que el estado puede **escribir** desde un *Userdata* (explicado en los siguientes párrafos) en tiempo de ejecución. Las *output keys* pueden ser **leídas y usadas** por el siguiente estado en la transición si están declaradas como *input keys*
- Una lista de *String* llamada *io keys* que representa las llaves que el estado puede **leer** y **escribir** desde un *userdata* en tiempo de ejecución. La **lectura** se hace para utilizar la información en el estado y la **escritura** para actualizar el valor guardado en esa llave.

También posee un método abstracto llamado *execute* que es el comportamiento que se debe sobrescribir para ejecutar al momento de hacer la transición hacia un estado.

Contenedores. SMACH posee una interfaz que implementa los estados llamada *contenedores*. Los contenedores definen la conexión entre contenedores, métodos para obtener y establecer el estado actual, así como llamados de inicio, transición y termino de los contenedores. También se definen herramientas de almacenamiento y registro de las transiciones y ejecuciones de los estados. Los contenedores inicializan los *userdata* y también reestructuran los *userdata*. En cada transición se revisan que las *keys* declaradas en los *input keys* se puedan copiar desde el *userdata* enviado cuando se inicializa un contenedor y que aquellas declaradas *output keys* se puedan escribir desde el *userdata* enviado cuando se termina de ejecutar un contenedor.

Userdata. Un *userdata* es una estructura similar a un diccionario. Posee *keys* y también posee *locks* para asegurar el acceso concurrente a los valores guardados. Los *userdata* permiten volver a asignar los valores a otras llaves. Este proceso puede ser necesario cuando se quieren componer dos contenedores que generan y/o reciben distintas llaves pero que poseen el mismo tipo de dato y tienen el mismo significado para los contenedores. Los contenedores son los encargados de restringir la lectura/escritura en los estados de los *userdata*. Si bien es posible escribir en un *userdata* información que no está declarada en los *output keys*, al momento de hacer una transición, el contenedor verificara cuales son las *io keys* y copiara solo las declaradas como *output keys*, ignorando la información extra.

Máquina de estados. Una *state machine* o máquina de estados es una implementación de un contenedor y, por lo tanto, una implementación de un estado. SMACH presenta un *composite pattern* [13] para manejar la arquitectura de sus máquinas de estados. La ventaja de aplicar este patrón de diseño es la creación de máquinas de estado jerárquicas o como ya

fueron nombradas: NSM. Un estado dentro de una máquina de estado puede ser un estado por si solo o una máquina de estados completa. La gran diferencia entre un estado y una máquina de estado es el comportamiento que ejecutan, mientras un estado ejecuta el comportamiento definido por su método *execute*, la máquina de estados va a ejecutar el comportamiento definido por su estado inicial y se moverá según las transiciones definidas, continuará hasta que no queden más estados que ejecutar siguiendo las transiciones definidas previamente. Si se desea terminar la máquina de estados en algún estado en particular, se debe definir que la salida de ese estado esté definida como la salida de la máquina de estados. Puede darse el caso de quedar en un estado sin salida a las salidas de la máquina de estados y dejar la máquina detenida sin terminar. Por ello es importante definir qué estados son terminales o de aceptación y realizar las conexiones con las salidas de la máquina de estados.

Una máquina de estado puede añadir estados siempre y cuando no haya empezado su ejecución. Para evitar errores en tiempo de ejecución por una máquina mal formada, se realiza un chequeo previo donde se revisa que exista un estado inicial, que el estado inicial pertenezca a los estados de la máquina de estado y que las transiciones de cada estado registrado sea un estado valido además de pertenecer a la máquina de estados.

2.1.3. Software para Robots

Para poder conectar una máquina de estados con un robot real no basta con utilizar SMACH. Existen diferentes conjuntos de librerías o frameworks que permiten el trabajo con robots como lo son Hop [14], MSRS [15], Carmen [2], ARIA [3], Webots [4], ROS [16], entre otros. Tsardoulias y Mitkas [17] nombran y comparan diferentes frameworks y middleware de robótica, confrontando 6 cualidades entre todos los mostrados. Para este trabajo se utilizó ROS por poseer una amplia gama de lenguajes compatibles además de utilizarse en diferentes universidades y empresas a lo largo del mundo.

ROS es un *middleware* que permite conectar diferentes librerías y herramientas con drivers enfocados en robots. Es un proyecto de código abierto disponible desde 2007 con 13 versiones desde su lanzamiento. El proyecto actualmente es liderado por Open Robotics quienes deciden los siguientes pasos y mantienen el código principal junto con las librerías más utilizadas. Se está trabajando en ROS2, un proyecto en paralelo con mejoras importantes, pero aún no logra la estabilidad de ROS. ROS está disponible para C++ y Python de manera oficial. Existen algunos trabajos para conectarlo a otros lenguajes como Pharo Smalltalk [5], Java [6] y JavaScript [7] que son mantenidos y desarrollados por la comunidad, pero no por el equipo principal detrás de ROS. ROS presenta una arquitectura de Publicador/Subscriptor a través de mensajes tipados y bien estructurados. Los mensajes son procesados por los drivers de los actuadores (dispositivos que generan movimiento lineal o rotacional e.g. motores) o sensores para realizar las tareas requeridas.

Si bien SMACH, en su creación, es una librería independiente de ROS, existe una fuerte

[2] http://carmen.sourceforge.net/home.html
[3] https://github.com/cinvesrob/Aria
[4] https://cyberbotics.com
[5] http://car.mines-douai.fr/category/pharos/
[6] http://wiki.ros.org/rosjava
[7] http://wiki.ros.org/roslibjs

conexión ya que se creó para trabajar con él. Toda conexión del robot con el estado es a través de ROS y sus herramientas de conexión. Es importante notar que los mensajes en ROS son una fuente de datos que puede utilizar la máquina de estados en su ejecución, por ejemplo, si se necesita leer información desde uno o varios sensores.

Al modelar un comportamiento robótico en una máquina de estados la comunicación con el robot puede tratarse como una habilidad que es capaz de realizar el robot, evitando realizar conexiones especificas en el estado con el robot. Esto permite que una máquina de estados pueda utilizarse para múltiples robots cumpliendo el mismo propósito si los robots poseen las mismas habilidades que requiere el comportamiento. El estudio de las habilidades es amplio [18, 19, 20] y permite realizar una abstracción similar a la pretendida en SMACH, pero con tareas más simples del robot.

2.2. Fuzz testing

Cuando hablamos de pruebas en código hay que hacer una diferencia en el tipo pruebas que se realizan, son pruebas que esperan un resultado o no. Las pruebas que esperan un resultado en general funcionan de manera similar: dado un conjunto de entrada definido, esperamos otro conjunto de salida definido. Si el conjunto de salida no es igual al conjunto de salida esperado, entonces el código no pasa la prueba.

Las pruebas que no esperan un resultado se centran en tomar un conjunto de entrada y al ejecutar ese conjunto se hace necesario revisar cuales son los valores del conjunto de salida. Son pruebas automáticas que requieren ser revisadas y pueden aportan a conocer el estado del programa junto con su comportamiento.

Cada prueba que se escribe debería tener un conjunto de entrada distinto, si no, estamos, potencialmente, repitiendo la misma prueba. Es común ver que los conjuntos de entrada para las pruebas sean conjuntos de entrada esperados y válidos, es decir, si un código espera una palabra, es muy probable que se utilicen palabras válidas para realizar las pruebas. Esto genera que las pruebas que se realizan sean pruebas de escenarios donde la aplicación debería funcionar o sean pruebas bajo supuestos que no necesariamente se cumplen, lo cual hace que no sea suficiente para asegurar que la aplicación no contenga errores.

Fuzz Testing es una técnica de pruebas de software que busca **generar** pruebas de manera automática. Para generar pruebas de manera automática lo más importante es la generación de los datos de entrada. Las técnicas de *fuzz testing* son implementada por los *fuzzers* que representan concretamente la técnica. Existen diferentes formas de clasificar los *fuzzers* [3, 21], entre ellas están:

- Como se generan las entradas.
- Se analiza la estructura de la entrada.
- Se analiza la estructura del programa.

2.2.1. Generación de entradas

Un *fuzzer* puede generar sus entradas de dos maneras, (i) utilizando una semilla de donde generar los siguientes datos modificándola o (ii) utilizar estructuras para crear los valores de entrada desde cero.

Basados en mutaciones

Un *fuzzer* que utiliza la estrategia de generar los datos en base a una semilla o valor de entrada se conoce como *fuzzer basado en mutaciones* o *mutation-based fuzzer*. Este valor de entrada debe ser válido para luego generar valores en torno a esa entrada. Por ejemplo, un *fuzzer* basado en mutaciones que desea probar una función que recibe direcciones web, debe recibir una dirección válida para comenzar a mutarla. Un posible valor es `https://www.google.cl`, el cual puede mutarse de diferentes formas como agregando caracteres entre los caracteres especiales `.` `,` `:` o `//`. Incluso es posible modificar estos caracteres especiales si así se requiere para probar cómo se comporta la función. Es claro que modificar ciertas partes podrían invalidar directamente la dirección web, como sería el caso si la dirección muta a `hts://www.google.cl`, por lo tanto, es importante realizar una mutación que permita generar entradas válidas para el programa a probar o semi válidas, es decir, que podrían llegar a entregarse al programa pero que respeten la estructura si existen restricciones.

Un *fuzzer* basado en mutaciones puede tener una tasa de éxito mayor que utilizar cualquier valor aleatorio para probar una función o programa ya que parte de una entrada valida. El tipo de mutación que se le aplique a la entrada puede generar tipos únicamente válidos o algunos semi válidos que puedan generar problemas.

Basados en la generación

La otra estrategia para generar entradas es comenzando desde cero. Esta estrategia se conoce como *fuzzer basado en la generación* o *generation-based fuzzer*. En este tipo de *fuzzer* la creación de los datos de entrada no depende de ninguna semilla si no que de estructuras para poder crear el dato. Por ejemplo, un *fuzzer* de este tipo puede tomar como base un modelo del tipo de dato de entrada y con el generar diferentes entradas válidas bajo ese esquema.

Para estructuras de datos complejas el *fuzzer* basado en la generación resulta mucho mejor que el basado en mutación ya que permite crear ejemplo validos que probaran la función o el programa sin errores por el dato entregado, enfocándose mejor en el correcto funcionamiento. Por el contrario, si se desea probar la robustez del sistema frente a entradas de usuarios, por ejemplo, el *fuzzer* basado en mutación permite emular de mejor manera entradas erróneas que podría generar un usuario y por lo tanto seria mucho más efectivo que un *fuzzer* que busca mantener una estructura fija pero que podría perder sentido semántico de las entradas.

Algunos *fuzzer* mezclan estas dos formas de trabajar las entradas tomando una estructura definida de los basados en la generación y una semilla para modelar parte de la estructura permitiendo construir entradas complejas en estructura, pero similares a ejemplos reales con variaciones en ella.

2.2.2. Estructura de la entrada

Es natural que un programa no responda bien a cualquier tipo de entrada, más aun, dependiendo del lenguaje de programación utilizado, es posible que el mismo lenguaje se encargue de desechar las entradas que no son del tipo esperado. El uso de estructuras dentro de la entrada es crucial para poder generar un fuzzer que permita encontrar errores. En esta categoría de *fuzzer* existen dos tipos bien distinguibles:

- Los *fuzzer* inteligentes.
- Los *fuzzer* torpes.

Fuzzer inteligentes

Los *fuzzer inteligentes* o *smart fuzzer* son aquellos que generan los datos en base a alguna estructura, modelo [22, 23], gramática [24, 25] o protocolo [26, 27]. Por ejemplo, si la entrada puede ser modelada por un árbol de sintaxis abstracta (AST) entonces un *fuzzer* basado en mutación inteligente puede tomar ese AST con un ejemplo y comenzar a variar su producción para ver cómo afecta al programa. Si la entrada puede modelarse como una gramática formal, entonces un *fuzzer* basado en la generación inteligente puede generar los datos a utiliza siguiendo las reglas de producción de la gramática. Para este caso el modelo debe ser explicito en que se puede crear y que no.

Los *fuzzer* inteligentes sirven principalmente cuando el software es conocido o se tiene acceso a parte de él. Existen técnicas como la inducción gramática [28] que permiten obtener las reglas de producción de una gramática desconocida en base a las observaciones que se tienen, esto permite obtener las reglas de producción si existen para un programa, pero puede tomar más trabajo incorporar esta técnica.

Fuzzer torpe

Un *fuzzer* torpe no requiere conocer el modelo ni se enfoca en crear entradas válidas. Por ejemplo, un *fuzzer* basado en mutación torpe puede ser aquel que toma una imagen en formato PNG y modifica parte de sus valores de píxeles a cualquier otro valor. Podemos dejar que modifique los valores por cualquier valor válido o no para probar como se comporta el programa frente a ese tipo de archivos o si los reconoce cómo archivos corruptos.

El *fuzzer* torpe es especialmente útil cuando no se conoce el programa a probar, no existe una estructura o modelo que pueda generarse. Es mucho más simple de implementar al no tener que entender el programa, pero puede que la mayoría de los errores que detecte sean problemas con la entrada que se está entregando.

Podemos ver algunas mezclas de estos dos *fuzzer* si aplicamos un *fuzzer* inteligente basado en un modelo que en los niveles más bajos genera valores sin sentido semántico utilizando un *fuzzer* torpe pero que de igual manera calzan con el modelo general.

2.2.3. Estructura del programa

Dentro de lo que se puede analizar de un *fuzzer* al hacer pruebas es ver cuanta cobertura del código se realiza. Normalmente si el programa o la estructura del programa no es conocida es difícil alcanzar un alto porcentaje de cobertura. Cuanto más se conozca del programa, mayor cobertura se podrá realizar al entender que cambios en la generación de datos de entrada pueden generar un cambio de comportamiento en el programa. Existen 3 formas de clasificar un *fuzzer* cuando se analiza la estructura del programa:

- *Black-box.*
- *White-box.*
- *Gray-box.*

Black-box

Un *fuzzer black box* [29, 30, 31] se define por tratar al programa como una caja negra y desconoce la estructura interna del programa. Por ejemplo, una herramienta que genere entradas aleatorias para cualquier programa se considera un *fuzzer black-box*. Esta estrategia es interesante ya que permite ejecutar muchas instancias del *fuzzer* de manera paralela para tratar de extraer problemas. A pesar de ello, un *fuzzer black-box* generalmente no encuentra problemas profundos o complejos, sino que sirve para comprobar que el programa tiene un funcionamiento correcto a gran escala. Existen trabajos [32, 33] que buscan mejorar la eficiencia de *fuzzers black-box* utilizando diferentes técnicas para aprender de las entradas y salidas obtenidas y con ellas generar nuevos casos para buscar problemas más complejos.

White-box

Los *fuzzer white-box* [30, 31] requieren de un análisis del programa para incrementar la cobertura de código o para probar partes importantes del programa. La diferencia principal con los *fuzzer black-box* es la especificación del programa que se tiene. Para poder ejecutar un *fuzzer white-box* es necesario tener información que permita conocer que entradas utiliza el programa, el tipo de las entradas y también conocer a los estados que podemos llegar con ciertas entradas. Con esta información es posible construir un *fuzzer* basado en el modelo o en gramáticas que permita especificar estas entradas y revisar si las salidas cumplen las especificaciones del programa.

Tener las especificaciones del programa también permite realizar pruebas más complejas, que involucre realizar llamadas a funciones o lugares del programa que no son ejecutadas constantemente. Sin embargo, el análisis necesario para obtener las especificaciones del programa puede resultar tan complejo y costoso que se vuelve ineficiente, más aún si se considera el tiempo de generación de las entradas.

Gray-box

A diferencia de los *fuzzer* del estilo *white-box* (internas) y *black-box* (externas) el *fuzzer gray-box* [34] utiliza técnicas instrumentación binaria liviana que permita conocer mas información que utilizando un *fuzzer black-box*. Al utilizar esta información el *fuzzer gray-box*

puede detectar vulnerabilidades de manera más eficiente que el *fuzzer white-box* y también más complejas que las detectadas por el *fuzzer black-box*.

La combinación de diferentes características dentro del *fuzzer* permite encontrar diferentes tipos de problemas en los programas probados. A su vez, el uso de una técnica sobre otra aumentara la complejidad de la herramienta, pero también generar mayor calidad en las pruebas a realizar, permitiendo complicar el contexto donde se realizan las pruebas.

2.2.4. El fuzzer utilizado

Este trabajo de tesis utiliza un fuzzer basado en gramáticas de generación e inteligente que combina técnicas de *black-box* y *white-box* para ir construyendo los datos de entrada. Una gramática se compone de:

- Un conjunto finito N de símbolos no terminales.
- Un conjunto finito Σ de símbolos terminales que es disjunto de N.
- Un conjunto finito P de reglas de producción, que cumple:

$$(\Sigma \cup N)^* N (\Sigma \cup N)^* \rightarrow (\Sigma \cup N)^*$$

  Con * el operador de Kleene o cerradura de Klenne.

- Un símbolo $S \in N$ que es el símbolo de inicio.

Una gramática se define formalmente por la tupla (N, Σ, P, S).

En este trabajo se utiliza una gramática que permite replicar los datos de entrada posibles de los diferentes comportamientos robóticos. Cada dato por producir debe ser de un tipo válido y cada uno de estos tipos válidos pueden crear diferentes datos. Esta gramática está fuertemente conectada con el lenguaje de programación utilizado, debiendo modificarse en caso de utilizar otro lenguaje de programación.

Para ejemplificar el potencial de un *fuzzer* basado en gramáticas, consideremos la gramática de una palabra que es una cadena de caracteres. El conjunto de símbolos terminales (Σ) está compuesto por todos los caracteres válidos (conjunto finito de caracteres). Sus reglas de producción (P) están definidas en la Figura 2.1. Su conjunto de símbolos no terminales (N) está compuesto por: <string>, <character>, <digit>, <symbol>y <letter>. El símbolo no terminal de entrada (S) es <string>.

Estas reglas de producción permiten generar cualquier cadena de caracteres de largo 1 en adelante. Es posible variar la generación de palabras añadiendo más reglas de producción cómo por ejemplo:

<digit-string>|= <digit>|<digit><digit-string>

o

<letter-string>|= <letter>|<letter><letter-string>

Así es posible separar la generación de palabras según los diferentes tipos de caracteres: letras, símbolos y números. Cada nueva regla de producción que se añade puede especificar más y más la generación de palabras, pudiendo especificar el largo, la cantidad de cada tipo de caracteres o incluso la posición en la que puede aparecer cada carácter. Utilizar una gramática general y variar el símbolo inicial (S) permite generar diferentes palabras de salida con una única gramática sin la necesidad de variar toda la tupla (N, Σ, P, S). A mayor complejidad en la gramática utilizada el *fuzzer* puede tomar más tiempo en generar los datos de entrada, incluso podría caer en ciclos profundos de resolución si existe recursividad dentro de la gramática. Esto influye especialmente comparando con otros tipos de *fuzzer* como el *black-box* que simplifica la generación de entradas o los basados en mutación que permiten comenzar de un valor inicial sin tener la necesidad de resolver o ajustar muchos valores como podría pasar en los basados en gramáticas.

$$
\begin{aligned}
\langle \text{string} \rangle &\models \langle \text{character} \rangle \mid \langle \text{character} \rangle \langle \text{string} \rangle \\
\langle \text{character} \rangle &\models \langle \text{digit} \rangle \mid \langle \text{symbol} \rangle \mid \langle \text{letter} \rangle \\
\langle \text{digit} \rangle &\models 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\
\langle \text{symbol} \rangle &\models \; \mid ! \mid \text{”} \mid \# \mid \$ \mid \% \mid \& \mid ' \mid) \mid (\mid * \mid + \mid - \mid , \mid . \mid / \\
\langle \text{letter} \rangle &\models A \mid ... \mid Z \mid a \mid ... \mid z
\end{aligned}
$$

Figura 2.1: Gramática de un string.

Capítulo 3

Trabajo Relacionado

Este capítulo muestra diferentes trabajos que están relacionados con este trabajo de tesis. La primera parte busca mostrar diferentes trabajos que hablan de *fuzz testing* y como su uso en diferentes contextos ha sido de ayuda. La segunda sección busca mostrar algunos trabajos relacionados con pruebas en robots, diferentes enfoques que se han utilizado en robótica y las múltiples capaz que pueden o deben probarse en el desarrollo de un robot. Finalmente se muestra un trabajo similar, pero con un enfoque complementario que puede potenciar el fuzzer construido.

3.1. Fuzz testing

Como se señala en la Sección 2.2 existen diferentes formas de construir un fuzzer para realizar pruebas en programas. Este trabajo de tesis utiliza un fuzzer basado en gramáticas que mezcla white-box con black-box para generar pruebas de código.

Uno de los primeros trabajos en utilizar gramática y la generación de pruebas a través de un fuzzer white-box fue Godefroid *et al.* [35], quienes emplean un algoritmo para la generación de las entradas que evita la generación de caminos similares en cada ejecución, buscando maximizar la cobertura de las pruebas. Luego comparan sus resultados obtenidos con diferentes tipos de fuzzers: black-box, black-box basado en gramáticas, white-box y white-box extendido con tokens. En sus resultados muestran que utilizar un fuzzer white-box basado en gramáticas para su contexto logra mejor cobertura que los otros fuzzers. También que utiliza menor cantidad de entradas que todos los demás fuzzer para lograr su objetivo, lo que atribuyen a una generación de entradas de mayor calidad. Se reconoce las dificultades de utilizar esta estrategia como lo es el tiempo de cómputo de las entradas y la necesidad de obtener o generar la gramática para el problema.

Otro trabajo similar es *Grammarinator* [25], una herramienta que utiliza gramáticas y mutación para generar pruebas. Grammarinator utiliza la gramática del lenguaje para crear escenarios de prueba para el programa, evaluando cómo se comporta frente a esos casos. Estos casos son gramaticalmente correctos por lo que es posible incluso manejar la complejidad de ellos. Adicionalmente utiliza los casos de prueba existentes para crear nuevos utilizando la

gramática y un algoritmo de mutación. Grammarinator se puede considerar un fuzzer black-box ya que a priori no es necesario conocer información del programa mayor a la gramática.

EvoGFuzz [36] es una herramienta que mezcla técnicas de fuzzer basados en gramáticas con algoritmos evolutivos para modificar la probabilidad de generación dentro de la gramática y ajustar en cada ejecución la gramática a utilizar, por ejemplo si no se están encontrando problemas los elementos elegidos en la gramática pierden probabilidad de escogerse nuevamente, pero si se detecta una vulnerabilidad la probabilidad se aumenta para seguir buscando con esas probabilidades de la gramática. Para ello se ajusta una función de fitness que permite aumentar o disminuir las probabilidades de cada elemento dentro de la gramática si se encuentran o no vulnerabilidades con la gramática utilizada. Una clara ventaja frente a los demás fuzzer es la posibilidad de modificar las probabilidades de generación dentro de la gramática para realizar pruebas con entradas similares a las que ya generaron errores. Si bien el uso de las probabilidades en la generación de errores puede ser beneficiosa también puede generar que las entradas se comiencen a centrar en entradas que detecten un error único o un mismo tipo de error con diferentes parámetros.

Estos trabajos son algunos ejemplos del potencial existente al utilizar fuzzer basados en gramática, ya sean white-box o black-box, con diferentes aproximaciones, uno buscando maximizar la cobertura del código a probar y el otro buscando generar herramienta capaz de construir diferentes pruebas para detectar errores en el código. También muestran que el análisis de las salidas es importante y se puede utilizar para generar más errores modificando la entrada que genero el error.

3.2. Pruebas en robots

Un reciente estudio muestra como realizan pruebas en robots diferentes profesionales de diferentes empresas y academias [1], en él se concluye que las principales dificultades al momento de realizar pruebas en un robot son las complicaciones del mundo real, la comunidad con sus diferentes estándares y por último la integración de componentes. Este trabajo de tesis busca reducir una parte de las dificultades de integración de componentes al permitir probar el software sin la necesidad de un robot real, reduciendo las complicaciones de integración entre el software y hardware del robot a problemas de compatibilidad entre los componentes y no a errores generados por el software.

Los robots se pueden ver como un conjunto de sensores, actuadores y procesadores que trabajan en conjunto para resolver tareas, esto es similar a lo que hacen los sistemas distribuidos [37]. Pero la gran diferencia entre los programas para robots con los programas comunes es que el robot depende del entorno, pero también puede modificarlo, por lo que cualquier acción que realice tendrá un efecto en el mundo real y cualquier acción que se realice cerca del robot podría afectar su funcionamiento. Por lo tanto, realizar pruebas a un robot es algo necesario pero que no simple de realizar al depender físicamente del entorno en el que se desenvuelve.

Laval *et al.* [38] presentan una metodología para realizar pruebas a robots generando 5 niveles de pruebas que se deben realizar, cada una de ellas depende de la anterior y su intención es maximizar la seguridad tanto de los operadores, el robot y el ambiente. Este

trabajo propone también la generación de pruebas reusables y replicables, para asegurar que el robot continuara funcionando de la misma forma que cuando paso las pruebas. Poder asegurar que un robot funcione en el mundo real de igual manera en dos ejecuciones distintas es difícil ya que el ambiente cambia y con ello la información que el robot recibe será distinta. El funcionamiento de sensores y actuadores es una parte crucial del robot que debe ser revisado antes de que el robot comience una tarea, pero es importante también verificar que el programa que va a ejecutar el robot sea válido y no tenga errores. Por ello, este trabajo de tesis propone trabajar en la parte de los comportamientos robóticos asociados al código fuente, no considerando las componentes físicas del robot, aunque representen una parte importante de los desafíos de pruebas en robots.

Si se acota el contexto de la robótica a las herramientas utilizadas en este trabajo es posible generar pruebas para estas herramientas. ROS posee un paquete de pruebas llamado *rostest*[1]. Rostest permite realizar pruebas de integración sobre los componentes de ROS. Estas pruebas también permiten realizar pruebas unitarias para comprobar si el sistema está funcionando como corresponde. Si se lleva al contexto de este trabajo de tesis, rostest permite realizar pruebas de integración en Python conectándolas con ROS de manera transparente.

Por último, *Robot unit testing* [39] acerca los principios clásicos de las pruebas unitarias de ingeniería de software aplicadas en robótica. RUT considera los simuladores como una herramienta válida y suficientemente precisa para realizar pruebas en software para robots. Las pruebas unitarias automatizan el monitoreo de algunos escenarios bien definidos. El enfoque de este trabajo de tesis es complementario al presentado en estas publicaciones dado que las técnicas de *fuzz testing* exploran un espacio de escenarios posibles en vez de restringir la fase de pruebas a un conjunto de escenarios acotados y bien definidos.

3.3. Fuzz testing en robots

Realizar pruebas en robots no es un tema nuevo [1, 2, 38, 39] y existen trabajos de *fuzz testing* en robótica. Un trabajo realizado para ROS llamado *ros1_fuzzer*[2] crea un fuzzer que opera a nivel de la capa de datos de ROS. Este fuzzer permite ejecutar una aplicación de ROS y generar los datos utilizados por esta aplicación de manera automática. El fuzzer creado es uno basado en modelos y utiliza la estructura de los tipos de datos de ROS para generar los datos necesarios. Es probable que un fuzzer basado en mutación mezclado con uno basado en modelos podría generar una mejor respuesta al generar mensajes desde una base real de mensaje. También debe ser ejecutado en paralelo a la ejecución del programa de ROS que se desea probar.

El fuzzer construido en este trabajo de tesis opera al nivel particular de las máquinas de estados construidas en SMACH y ROS, no opera en programas genéricos de ROS. El trabajo de *ros1_fuzzer* es una prueba de que existe potencial para realizar este tipo de pruebas en comportamientos robóticos. Ambos trabajos son complementarios y son herramientas considerablemente poderosas para probar un comportamiento de manera compleja al poder manejar tanto las entradas internas como externas del robot.

[1]http://wiki.ros.org/rostest
[2]https://github.com/aliasrobotics/ros1_fuzzer

Capítulo 4

Pruebas en comportamientos robóticos

Este capítulo detalla la parte técnica del trabajo realizado, muestra las principales similitudes y diferencias en las pruebas para comportamientos robóticos con pruebas para software general. Se explica el por qué es importante realizar pruebas con valores no necesariamente conocidos y las ventajas de realizar pruebas para un comportamiento robótico utilizando valores válidos automáticamente generados. Para finalizar se explica cómo se compone una prueba utilizando *fuzz testing*.

4.1. Pruebas en comportamientos como máquinas de estado

Como se señala en la Sección 2.1, un comportamiento robótico puede ser modelado como una máquina de estado, por lo tanto, si se quieren realizar pruebas es posible hacerlo de manera símil a las pruebas de máquinas de estados. En la literatura [40] existen diferentes formas de probar una máquina de estados finita que persiguen diferentes objetivos, algunos de ellos son:

(i) Probar los estados de la máquina de estados.

(ii) Probar las transiciones de la máquina de estados.

(iii) Probar los caminos de la máquina de estados.

Cada uno de estos objetivos también requiere cierto conocimiento de la máquina de estados, por ejemplo, para (i) probar los estados de la máquina de estados, es necesario conocer el conjunto de estados de la máquina de estados. De igual manera, para (ii) probar las transiciones de la máquina de estados, es necesario conocer los estados de la máquina de estados y las transiciones de ella. Para (iii) probar los caminos posibles de la máquina de estados es necesario conocer los estados y las transiciones existentes, además de contar con el estado inicial de la máquina.

En un contexto diferente como las pruebas unitarias o *unit testing*, si se le entrega el

conjunto $x_1, x_2, x_3, ..., x_n$ como conjunto de valores de entrada a una función se espera que el valor de salida sea el mismo para la función si se utiliza el mismo conjunto de entrada dos o más veces, es decir, no debería cambiar el resultado si no existe información extra a las entradas de las funciones. Esto no es diferente en una máquina de estados, dado un conjunto de entrada se espera que los cambios de estados sean los mismos si ese conjunto de entrada son todos los datos que utiliza la máquina, es decir, es una máquina de estados determinista.

En un comportamiento robótico los datos utilizados no solo dependen de los datos de entrada inicial, el comportamiento recibe estímulos que pueden afectar la decisión de cual será el siguiente estado a ejecutar o incluso terminar la ejecución. Dado que el robot interactúa con el mundo real, los datos que utilizan los comportamientos no siempre serán proporcionados por el código fuente como tal, sino que existe información externa al código fuente que debe analizarse para, en algunos casos, tomar decisiones. Un estado puede obtener esta información externa a través del robot y no necesariamente utilizarla, sino que puede ser utilizada para tomar decisiones en los estados posteriores. En términos técnicos un comportamiento se asemeja a una función con *side-effects*, pudiendo modificar el mundo real que sirve para otros comportamientos sin modificar los valores entregados a los demás comportamientos.

Generar pruebas que verifiquen si el conjunto de salida o estado final de la máquina de estados es uno determinado dado un conjunto inicial de entrada es poco efectivo si existen datos externos que no son analizados ni definidos, ya que estos pueden influir de mayor manera en la ejecución misma de la máquina que los datos de entrada iniciales. En una situación hipotética, si fuese posible generar todas las fuentes de datos de un comportamiento robótico entonces se podría generar una prueba completa, poniendo especial énfasis en que el comportamiento probado está correcto bajo el contexto específico de los valores entregados a las entradas.

Ahora bien, realizar pruebas unitarias en la ejecución de una máquina de estado no garantiza que se prueben todos sus estados ya que la máquina no siempre es lineal y puede tener que escoger entre dos o más caminos posibles. Si una prueba toma uno de esos caminos, los demás camino quedaran sin recorrerse. Recorrer todos los caminos de una máquina de estados es un problema que depende de la cantidad de estados y transiciones, por lo tanto, es necesario conocer la estructura de la máquina para poder probar todos los caminos posibles. La falta de pruebas sobre el funcionamiento de uno o varios estados dentro de la máquina de estados puede generar que, en una ejecución del comportamiento del robot, se tome un camino que lleve a los estados no probados y pueda generar un error o falla por tratarse de un comportamiento no probado previamente. Por ejemplo, en la Figura 4.1, existen estados que son llamados *estados de fallo* que permiten alertar al operador del robot que no se logró alguna de las tareas y que el comportamiento no se ejecutó correctamente. Estos tipos de estado también permiten llevar al programa y al robot a condiciones previas, en caso de necesitar volver a ejecutar el comportamiento. En caso de no conocer la estructura de la máquina de estados o que los valores de entrada varíen en el tiempo de ejecución de la máquina de estados, se hace difícil estimar que datos de entrada o cuando deben variar los datos para poder alcanzar estados que aún no se han probado.

Considerando el contexto de uso de estas máquinas de estados, el probarlas en un robot real es una tarea costosa en tiempo y recursos. Un comportamiento robótico podría tomar una

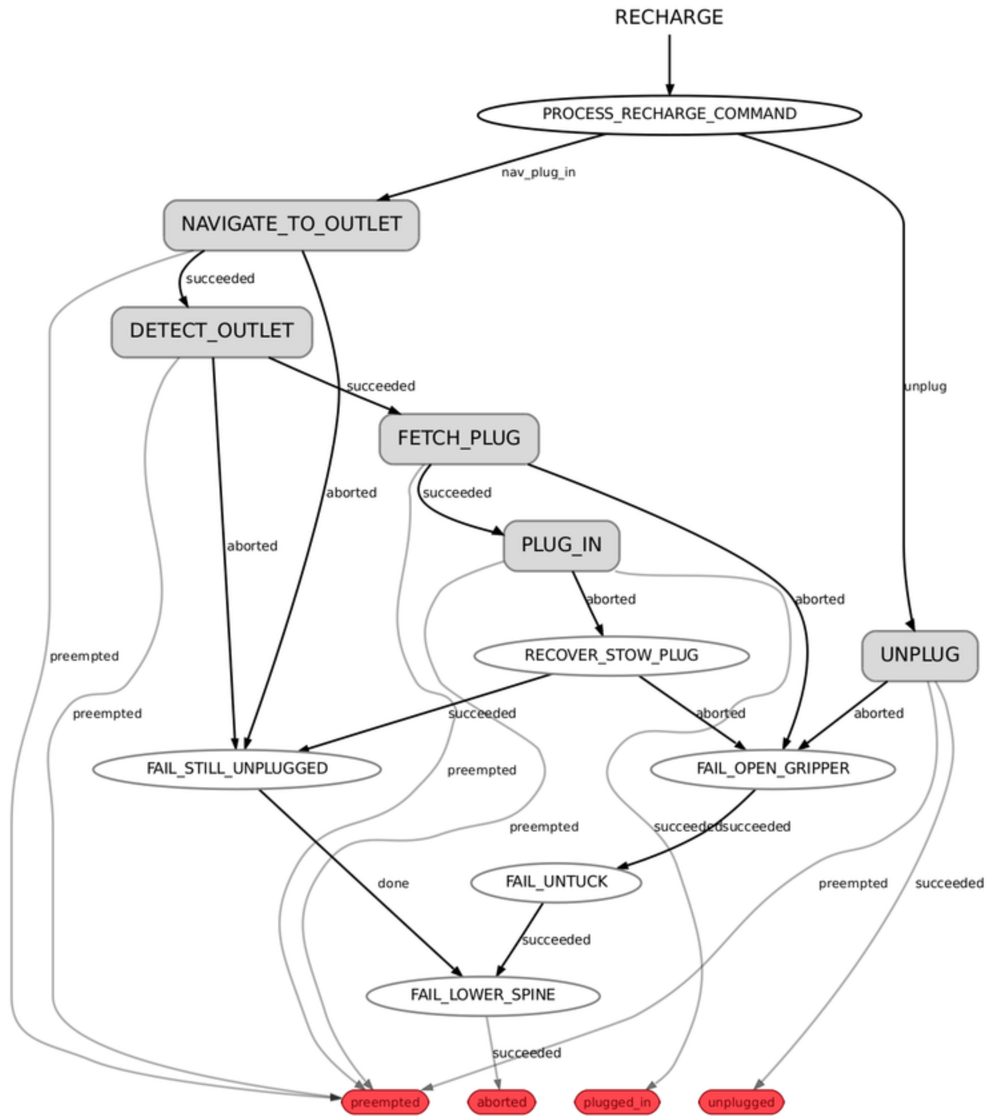

Figura 4.1: Máquina de estados para robot modelo PR2 que genera un comportamiento de auto conexión para recargar sus baterías, obtenida del trabajo de Curran *et al.* [41]

cantidad considerable de tiempo en realizar una sola tarea dada la necesidad de interactuar con el mundo real para realizar sus acciones y/u obtener datos. El uso de simuladores para asegurar que las máquinas de estados cumplen un con mínimo de pruebas es común en la industria robótica [2], pero, aun así, existe un alto costo en tiempo al probar pequeños cambios de código en simulador o probar que un problema de software se resolvió correctamente. Por ello, en este trabajo de tesis se proponen realizar pruebas menos costosas en tiempo al no precisar de un robot para su ejecución y así reducir el numero de pruebas infructuosas por errores que se puedan generar en tiempo de ejecución, pero que no son faciles de detectar.

4.2. Pruebas de estados aislados

Probar un comportamiento completo (de inicio a fin) es una tarea complicada y que consume tiempo, además de no asegura que se prueben todos los estados o caminos posibles. La técnica de pruebas de este trabajo de tesis se trata en descomponer la máquina de estados en cada uno de los estados que la componen. Con esta estrategia se asegura probar todos los estados dentro de la máquina de manera individual y sin depender de la cantidad de caminos que pasan por la entrada seleccionada para realizar la prueba. Como se explica en la Sección 2.1.2 cada estado de la máquina de estados ejecuta una sección de código previamente definida que representa el comportamiento que se ejecutará. Entonces, probar cada estado de la máquina de estados por separado permite saber si existe algún estado con un comportamiento errado o con algún problema mayor, como por ejemplo, errores en manejo de información, errores de tipos o incluso si existen entradas que permitan llegar a todas las salidas declaradas.

Como se expone en la Sección 4.1, no siempre los datos que necesita el estado para su ejecución son entregados a través del código fuente o por medio de las transiciones de los estados, si no que existen fuentes externas, como sensores del robot, que pueden entregar información útil para el estado. Por esto se definen dos fuentes de datos distintas:

- **Internos**: Las entradas del tipo internas son aquellas que su valor es fijado antes de la ejecución del estado y raramente estos varían mientras se ejecuta el estado. En caso de cambios en tiempo de ejecución, la metodología descrita en este trabajo de tesis no será aplicable ya que asume estas entradas como valores fijos que no varían. El uso de entradas internas sirve para enviar información de un estado a otro sin depender de parámetros globales que pueden ser modificados por diferentes fuentes. Un ejemplo de entradas internas serían los argumentos que se le entregan a una función que se quiere probar en una aplicación común.

- **Externos**: Las entradas del tipo externas normalmente varían en tiempo de ejecución. En general son valores obtenidos por los sensores del robot, aunque también pueden ser valores obtenidos por el software y que no son fijos durante la ejecución del estado, como acceder a una base de datos o algún servidor web donde se puede obtener información que influye en la ejecución del estado.

Cuando los estados interactúan con el robot lo pueden hacer de dos formas:

- Enviar una acción al robot y esperar hasta que esta termine de ejecutarse. e.g. Mover la cabeza del robot hasta cierta posición.

- Enviar una acción al robot y continuar la ejecución mientras la acción se lleva a cabo. e.g. Mover el robot hasta cierta posición esquivando obstáculos, mientras el robot avanza a la posición se debe analizar el ambiente en búsqueda de obstáculos, si existen algunos que afecten la trayectoria del robot, se debe volver planificar y continuar moviendo el robot, pero ajustando su velocidad y dirección.

Como este trabajo de tesis se centra en el software y no en el hardware del robot, las pruebas realizadas no se ejecutan en un robot real, sino que utilizamos uno simulado para

generar las características del robot real. Existen dos formas de simular un robot, una de ellas es utilizando un *mockup* [42] donde la comunicación entre los estados y el robot son reemplazadas con una implementación por software. La otra forma de simular un robot es a través de un simulador virtual, el cual simula un ambiente real por software donde el robot se puede mover e interactuar con este ambiente. Pitonakova *et al.* [43] compara diferentes simuladores utilizados en robótica, en su trabajo muestra ventajas y desventajas que tienen los simuladores, incluso compara el rendimiento en diferentes escenarios, donde en escenarios grande todos los simuladores muestran un uso intensivo de CPU. El robot simulado responde de manera dinámica como si fuese un robot real. El robot simulado necesita responder las peticiones realizadas por los estados y los valores de sus sensores deben variar mientras el estado se está ejecutando. Esto permite realizar pruebas a un estado modificando tanto sus entradas internas como sus entradas externas sin utilizar un robot real.

Para este trabajo la simulación se realizó sin un ambiente virtual utilizando un robot mock para poder ejecutar más pruebas en menos tiempo, con ello permitir hacer pruebas a bajo costo en tiempo y recursos en comparación al uso de simuladores virtuales. Esta estrategia permite validar el código fuente para luego poder realizar pruebas sobre el comportamiento en sí, evitando errores que más adelante en este trabajo son explicados.

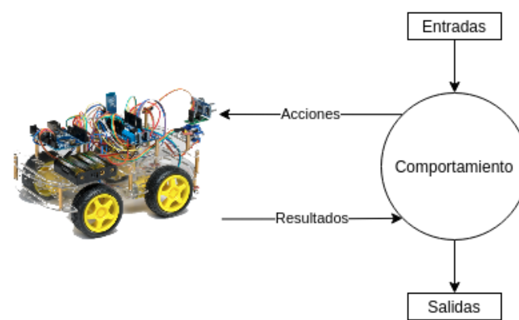

Figura 4.2: Interaccion entre el robot y el comportamiento.

La Figura 4.2 ilustra el proceso de prueba para un estado dentro de un comportamiento robótico. Es similar a las pruebas realizadas en un software en general, solo que en este contexto, el robot es una parte importante del problema y, en general, se utilizan simuladores que consumen un tiempo similar al del robot real en realizar las pruebas, reduciendo el costo de igual manera al no utilizar un robot físico para realizarlas.

Siguiendo las recomendaciones previas es posible escribir pruebas para un comportamiento utilizando un simulador del robot y ejecutando pruebas para cada uno de los estados del comportamiento de manera aislada en este robot. Esto permite probar el comportamiento, pero solo con datos que son conocidos y, por lo tanto, con una salida esperada por parte de la ejecución del estado, lo cual permite realizar pruebas del estilo unitarias al esperar una salida para un conjunto de entrada.

4.3. Pruebas aleatorias en estados

Probar una aplicación/función con un par entrada-salida generara un resultado favorable si la aplicación funciona correctamente. Utilizar esta práctica en comportamientos robóticos

no varía mucho ya que dado un conjunto de entrada se generará una única salida en caso de que el comportamiento dependa solo de este conjunto de entrada, lo cual no es posible de asegurar si el comportamiento no depende solo de ese conjunto de entrada. Crear pruebas con diferentes conjuntos de entrada permite obtener diferentes salidas que pueden ser conocidas previamente por el conocimiento del software. Por ejemplo, el equipo de desarrollo puede esperar cierto resultado de una prueba sin ejecutarla ya que saben que debería entregar en ese caso particular.

Este trabajo propone el uso de entradas aleatorias para probar los estados, de manera aislada e independiente, de la máquina de estados que modela un comportamiento robótico. Usar entradas aleatorias creará diferentes problemas en la ejecución del estado. Dependiendo del lenguaje que se utilice será posible experimentar dos comportamientos:

- Con lenguajes estáticamente tipados: El verificador de tipos se encargará de encontrar cualquier posible error en los tipos que se utilizan de entrada para probar, por lo tanto, será muy común encontrar errores de tipos al momento previo de ejecutar una prueba. De igual manera las entradas aleatoreas pueden generar entradas del tipo específico pero con valores que no sean los esperados.

- Con lenguajes dinámicamente tipados: El intérprete tratara de ejecutar hasta donde pueda adaptando el tipo de la entrada si es posible de hacer. En este caso el error puede ser o no generado ya que una instrucción podría llegar a ser válida si el dato es interpretado con otro tipo de dato. El programa fallará en tiempo de ejecución. En general no se realiza verificación de tipos explícitamente en los lenguajes dinámicamente tipados ya que se asume cierta coherencia en los componentes internos de un programa, solo la capa más externa de la aplicación contiene la verificación.

El problema se encuentra especialmente en los lenguajes dinámicamente tipados por permitir ejecutar comportamientos que no necesariamente están bien programados a diferencia de los con tipos estáticos donde existe un chequeo previo a la ejecución. Por ejemplo, una función que recibe parámetros `a` y `b` de tipo `enteros`, ya que realiza operaciones con este tipos de datos, no siempre revisa que los parámetros son del tipo `entero` y asume que quien utiliza la función lo hará de la manera correcta. De igual manera, si se realizan revisiones constantemente de los tipos o del valor del dato que ingresa en el comportamiento en tiempo de ejecución, puede afectar en el tiempo de respuesta, lo cual puede ser perjudicial para un robot trabajando en un ambiente real. Dado que en SMACH la máquina de estados es capaz de entregar información de un estado a otro, esta información podría contener errores de tipo, por lo que es necesario creer que al menos los datos internos del robot están bien definidos y no se producirán errores de tipos incompatibles, es decir, los estados entregaran valores razonables en sus salidas para que los utilicen los siguientes estados sin producir un error de tipo. Si no es así, existirá un potencial error en el comportamiento.

Para poder realizar pruebas con entradas aleatorias que sean razonables en el contexto de uso del software se puede utilizar un *dominio específico* de creación de estas entradas. Un dominio específico es acotar el espacio de valores posibles a generar según algún criterio. Por ejemplo, si se requiere una entrada de un tipo específico es necesario que la generación de valores sea producida por ese tipo de dato y no por otros tipos de datos. Es necesario fijar este dominio lo más especifico posible, es posible que con el tipo de dato esperado se

$$
\begin{array}{rcl}
\langle \text{start} \rangle & \models & \{\langle \text{ud} \rangle\} \\
\langle \text{ud} \rangle & \models & \langle \text{vars} \rangle, \langle \text{ud} \rangle \ \mid \ \langle \text{vars} \rangle \\
\langle \text{vars} \rangle & \models & \langle \text{key} \rangle : \langle \text{value} \rangle \\
\langle \text{key} \rangle & \models & \langle \text{str} \rangle \\
\langle \text{array} \rangle & \models & [\langle \text{selemements} \rangle] \ \mid \ [\langle \text{belements} \rangle] \ \mid \ [\langle \text{nelements} \rangle] \ \mid \\
 & & [\langle \text{melements} \rangle] \ \mid \ [\langle \text{ielements} \rangle] \ \mid \ [\langle \text{felements} \rangle] \\
\langle \text{selements} \rangle & \models & \langle \text{str} \rangle \ \mid \ \langle \text{str} \rangle, \langle \text{selements} \rangle \\
\langle \text{belements} \rangle & \models & \langle \text{boolean} \rangle \ \mid \ \langle \text{boolean} \rangle, \langle \text{belements} \rangle \\
\langle \text{nelements} \rangle & \models & \langle \text{number} \rangle \ \mid \ \langle \text{number} \rangle, \langle \text{nelements} \rangle \\
\langle \text{ielements} \rangle & \models & \langle \text{int} \rangle \ \mid \ \langle \text{int} \rangle, \langle \text{ielements} \rangle \\
\langle \text{felements} \rangle & \models & \langle \text{float} \rangle \ \mid \ \langle \text{float} \rangle, \langle \text{felements} \rangle \\
\langle \text{melements} \rangle & \models & \langle \text{elem} \rangle \ \mid \ \langle \text{elem} \rangle, \langle \text{melements} \rangle \\
\langle \text{elem} \rangle & \models & \langle \text{str} \rangle \ \mid \ \langle \text{num} \rangle \ \mid \ \langle \text{bool} \rangle \\
\langle \text{value} \rangle & \models & \langle \text{array} \rangle \ \mid \ \langle \text{element} \rangle \\
\langle \text{num} \rangle & \models & \langle \text{int} \rangle \ \mid \ \langle \text{float} \rangle \\
\langle \text{int} \rangle & \models & \langle \text{digit} \rangle \langle \text{int} \rangle \ \mid \ \langle \text{digit} \rangle \\
\langle \text{float} \rangle & \models & \langle \text{int} \rangle . \langle \text{int} \rangle \\
\langle \text{digit} \rangle & \models & 0 \ldots 9 \\
\langle \text{str} \rangle & \models & \text{"} \langle \text{chars} \rangle \text{"} \\
\langle \text{chars} \rangle & \models & \langle \text{char} \rangle \langle \text{chars} \rangle \ \mid \ \langle \text{char} \rangle \\
\langle \text{char} \rangle & \models & a \ldots z \ \mid \ A \ldots Z \ \mid \ \langle \text{digit} \rangle
\end{array}
$$

Figura 4.3: Gramática utilizada para generar datos, los caracteres $<$ y $>$ indican una regla de producción, los caracteres ... indican un rango de valores y se utiliza el caracter | para separar entre posibles valores de reglas de produccion.

generen errores si los valores esperados como entrada tienen un rango definido o son solo un subconjunto muy pequeño dentro de todos los valores posibles a generar. Un ejemplo es el uso de cadenas de texto o *String*, ya que, existe una cantidad inmensa de posibles valores para las cadenas de texto pero en general el contexto de uso de estas es acotado a un conjunto de posibles valores, determinar esos valores es importante para realizar pruebas específicas.

Una vez fijado el dominio de las entradas es posible generar diferentes combinaciones de estas entradas para cada estado que serán válidas en una ejecución real ya que los dominios son válidos para cada entrada.

Para construir este dominio específico existen dos técnicas de producción de valores, como se explica en la Sección 2.2.1, las basadas en mutación y en la generación. Este trabajo utiliza la técnica basada en generación para producir sus valores, más en específico, utiliza reglas de producción basadas en gramáticas específicas para generar los valores. Estas gramáticas deben adaptarse al contexto de cada software para obtener mejores resultados. En este trabajo

se utiliza la gramática descrita en la Figura 4.3 como gramática general. Para utilizar esta gramática es necesario tener una asociación entre el nombre del valor a generar y el tipo inicial de generación. El nombre y tipo de la variable dependerá de la entrada del estado a probar. En el Capítulo 5 se aborda como obtener estos valores metodológicamente. La gramática genera una cadena de caracteres que representa el valor sin interpretar de la entrada, por lo que debe ser un tipo de dato válido para que el lenguaje de programación pueda interpretar posteriormente. Por ejemplo, una entrada posible generada por la gramática es `"9"` que representará al número 9 luego de ser interpretada. Si se quiere generar el *string* 9, es necesario crear el que la gramática genere de salida `“‘9’”` utilizando una mezcla de comillas para ser interpretado como el *string* `‘9’`

El uso de una gramática general sirve para todos los casos donde no existe un dominio claro de valores a probar. Volviendo al ejemplo del *string*, es posible encontrar casos donde esperamos que el comportamiento reaccione bien frente a un numero finito de valores pero que también reaccione bien frente a los valores no definidos. Para los casos más específicos, donde es conocido el tipo de entrada o este puede tomar ciertos valores acotados, es necesario especificar un conjunto de reglas de producción específicas. Un ejemplo se puede ver en la Figura 4.4, donde se espera que el comportamiento reciba una cadena de caracteres pero **no** cualquier cadena es válida.

Para aquellos casos donde el tipo de dato a generar es correcto, pero aun así se obtienen errores relacionados a los valores de los datos es necesario realizar una inspección más detallada del estado y sus entradas. Este tipo de casos es difícil de detectar ya que podrían no generar un error en el software, pero si un comportamiento errado, como por ejemplo, que la salida del estado sea distinta de la esperada por el desarrollador.

Para las entradas externas, es posible utilizar la misma técnica de reglas de producción para generar entradas aleatorias válidas. Dado que los tipos de datos externos deben ser manejados por el software, estos deben tener una estructura fija antes de ser procesados, por lo que se puede utilizar esa estructura fija y conocida para generar diversos tipos de datos. Por ejemplo, la información que podria entregar un sensor tan básico como un botón: Posee dos posibles valores `presionado` o `libre`, lo cual puede ser transformado en software a un valor de verdad de `Verdadero` o `Falso`. Si el tipo de dato es alguna estructura específica mas compleja, es posible usar esa estructura para generar una gramática que se adapte a este tipo de dato y con ello realizar pruebas a sus estados. Para este caso particular se debe construir una forma de transformar desde la salida de la gramática (*string*) a la estructura deseada. Un parser podria generar los tipos de datos propios sabiendo las estructuras que deben ser construidas a partir de la generación que puede crear el *fuzzer* específico para esa entrada externa. También es posible adaptar la generación para que sea basada en modelos, similar a las técnicas utilizadas por los *fuzzer* basados en modelos [22] que resuelven el problema de transformar desde la salida de la gramática pero impidiendo hacer la exploración planteada en el Capítulo 5.

El uso de gramáticas en comparación con otras técnicas de generación de datos permite obtener valores que son válidos dentro del contexto utilizado, pero pueden ser muy diferentes entre sí, dando una mayor posibilidad a generar valores o conjuntos de entrada variados en cada ejecución. Además, permite la creación de estructuras recursivas o complejas sin

$$
\begin{array}{rcl}
\langle\text{list}\rangle & \models & [\langle\text{greetings}\rangle] \\
\langle\text{greetings}\rangle & \models & \langle\text{greeting}\rangle \mid \langle\text{greeting}\rangle, \langle\text{greetings}\rangle \\
\langle\text{greeting}\rangle & \models & \texttt{'Good Morning'} \mid \texttt{'Hi'} \mid \\
 & & \texttt{'Good Night'} \mid \texttt{'Hello'} \mid \\
 & & \texttt{'Good Evening'}
\end{array}
$$

Figura 4.4: Gramatica especifica para un estado.

necesidad de definir previamente los niveles de complejidad de las estructuras a generar o también se puede especificar la profundidad de la recursión a definir para evitar complicar tanto la estructura.

4.4. El Fuzzer

Este trabajo de tesis evaluó la técnica de pruebas con la construcción de un *fuzzer* que opera con SMACH (Explicado en la Sección 2.1.2). Gracias a la estructura de máquinas de estado jerárquicas de SMACH, el *fuzzer* es capaz de operar a nivel individual del estado (nivel fino de análisis y pruebas *white-box*) como también a nivel de la máquina de estados (nivel grueso de análisis y pruebas *black-box*). Cualquiera sea el nivel de las pruebas a realizar, es importante definir bien la gramática a utilizar y adaptarla según las necesidades de las pruebas.

4.4.1. Pruebas a nivel de estado

El *fuzzer* construido genera un *Userdata*, explicado en la Sección 2.1.2, y lo entrega a un estado particular, el cual es parte de una máquina de estados de múltiples estados. En este modo de pruebas el *fuzzer* estresa el comportamiento de un estado particular, buscando situaciones en la lógica interna del estado que puedan generar algún error en la ejecución al modificar los valores de entrada que recibe el estado.

Para generar apropiados *userdata* es crucial entender cuáles son los valores de entrada aceptados por el estado. Si bien el lenguaje empleado para la creación del *fuzzer* es Python, un lenguaje dinámicamente tipado, es necesario reconocer los tipos de los valores de entrada para evitar errores triviales con el tipo de entrada, esto ya que es posible entregar un valor *string* a un estado y que éste lo utilice tratándolo como un tipo distinto, por ejemplo, como una colección de elementos. A pesar de que el *string* sea una colección de caracteres, es posible que se necesite una colección de enteros para el comportamiento, por lo tanto, podrían existir comportamientos erróneos. Entregar un tipo de dato erróneo generará un error de tipo en tiempo de ejecución. En este caso el *fuzzer* detectará el error de tipo como un error del estado y no necesariamente representa un error de la estructura interna del comportamiento robótico. Aun así, es importante analizar la posibilidad de que llegue una entrada con un tipo erróneo al estado, lo cual podría suceder por estar presente en diferentes máquinas de estados y que su uso sea diferente en cada contexto, por lo que siempre es necesario descartarlo o aceptarlo como un error. Para ello es necesario revisar la sucesión de estados a los que está

conectado en la máquina de estados, si en ellos es posible generar la entrada con el tipo erróneo, entonces el error es real y no un problema de una mala entrada.

Para manejar este problema se propone el diseño de una técnica para determinar el tipo del valor esperado por el estado. La técnica consiste en monitorear los valores entregados para la ejecución inicial de un estado. El primer paso es entregar valores de entrada al estado que puedan o no ser conocidos sus tipos. Si sus tipos son conocidos entonces es simple seleccionar el tipo adecuado y ejecutar el estado con ese tipo de dato como entrada. Cuando el tipo de dato no es conocido, se puede utilizar la gramática general para inducir errores de tipo. Cada error de tipo inducido nos permite conocer cuales son los tipos de datos que no sirven como tipo de dato de entrada, sin necesidad de mirar el código fuente del estado. Este se logra almacenando cada ejecución con los valores de entrada que tienen las diferentes entradas del estado y el resultado de la ejecución. Para efectos de este trabajo, se define la *ejecución inicial de un estado* como todas aquellas empleadas en buscar el tipo de dato de los valores de entrada. Esta ejecución inicial permite conocer diferentes tipos de valores al mismo tiempo, sin necesidad de hacer un análisis o inspección visual del código fuente de los estados.

Luego de varias ejecuciones iniciales, es posible inferir el tipo de entrada de la información almacenada. Normalmente la información entregada a los *userdata* serán tipos de datos simples de Python, tales como cadenas de caracteres, números o valores de verdad. Tipos de datos o estructuras más complejas como instancias de objetos son raramente implementados para comportamientos robóticos. De igual manera se puede inferir dado que el tipo no se ajustará a ninguno de los tipos de datos genéricos de Python y será necesaria una inspección visual del código para determinar su tipo explícito. Como se explicó previamente, estos casos necesitarán una gramática específica para funcionar, especialmente una que contenga el tipo de dato utilizado y la gramática que genere ese tipo de datos.

Este tipo de pruebas asimila a un *fuzzer white-box* basado en gramáticas ya que se debe conocer la estructura interna del programa para poder ajustar tanto la gramática como los posibles valores que se requieren, pero a la vez su ajuste puede ser realizado de manera exploratoria descartando los posibles tipos de datos.

Para ello se construye un archivo de configuración en formato JSON[1] que contiene la información como se muestra en la Figura 4.5. Al asociarse un estado a una máquina de estados, esté se asocia a través de un nombre. Ese nombre se almacena en el archivo de configuración en la posición de `<state-name>` en la Figura 4.5. Cada uno de los estados posee dos campos dentro de la configuración: *params* y *data*. *params* se refiere a las entradas internas. *data* se refiere a las entradas externas. Ambas poseen la misma estructura interna *keys*, *types* y *grammar*:

- *keys* son los nombres de las entradas internas que serán utilizadas en el estado.
- *types* son los tipos de las entradas internas, referidos a la gramática y a las *keys*, es decir, se asocian por posición a las *keys* y representan un tipo de dato de la gramática a utilizar, ya sea la general o una específica. Este valor es desde donde la gramática comenzará a generar el dato pedido.

[1]https://www.ecma-international.org/publications-and-standards/standards/ecma-404/

```
<state-name>:{
  "params":{
    "keys":[],
    "types":[],
    "grammar":{}
  },
  "data":{
    "keys":[],
    "types":[],
    "grammar":{}
  }
}, ...
```

Figura 4.5: Estructura del archivo de configuración.

- *grammar* es un valor opcional, si no esta presente se utiliza la gramática general. Este campo permite escribir una gramática específica para el contexto del estado y así poder realizar un proceso de pruebas más específico.

La ejecución de un estado puede tener tres resultados: una ejecución normal, un error en la ejecución o una ejecución infinita. Cuando la ejecución es normal el estado tomará un tiempo en ejecutarse y luego retornar un *string* de salida. Si bien una ejecución normal nos dice que el estado se puede ejecutar bajo ese contexto, esto no necesariamente representa un correcto funcionamiento del estado. Si se da el caso de una ejecución normal, se abre la posibilidad a un comportamiento errado. Para detectar un comportamiento errado es necesario revisar las entradas utilizadas y el retorno esperado del estado. Esta inspección es la más costosa del proceso dado que requiere a una persona que conozca la lógica del estado.

Un error en la ejecución es un tipo de error interesante que el *fuzzer* puede encontrar. Su análisis es simple, se detecta por la generación de un error en tiempo de ejecución del estado. Se genera un *stack trace* con información relevante de la ejecución. Dado el almacenamiento de la información utilizada para ejecutar el estado, es simple definir bien el error: posee un *stack trace*, los parámetros de entrada, el estado que se estaba probando y tipo de error que se generó. Esto permite reproducir el error al volver a ejecutar el mismo estado con los mismo parámetros de entrada.

Para el caso de la ejecución infinita no es claro cuando considerarla infinita sin conocer el estado y que acciones realiza. Un ejemplo claro es un estado que espera cierta valor específico desde una de las entradas externas. Si ese valor no es el entregado, entonces el estado no avanzará. En este caso pueden darse dos posibilidades, (i) el *fuzzer* no es capaz de cambiar el valor esperado porque no fue previsto como un valor que cambiaría o (ii) el *fuzzer* no cuenta con el valor específico a generar. En ambos casos el problema es el mismo: un ciclo infinito que solo se puede terminar deteniendo la ejecución. Es importante entender que si se detecta un ciclo infinito la solución es simple, se debe agregar la entrada al *fuzzer* para que pueda modificar su valor o especificar la entrada. Para las pruebas automatizadas es necesario configurar un reloj que detenga la ejecución pasado cierto tiempo y notificar de

esta interrupción para su posterior revisión.

4.4.2. Pruebas a nivel de máquina

Naturalmente un robot debe interactuar con un ambiente donde el ruido en los sensores y las perturbaciones inesperadas son frecuentes. En las pruebas a nivel de la máquina el *fuzzer* estresa el comportamiento de una máquina de estados completa, en lugar de enfocarse en cada estado interno individualmente. Se puede decir que el este tipo de pruebas opera a nivel macro a diferencia de las pruebas de estados que operan a nivel fino al enfocarse en cada estado.

Las pruebas a nivel de máquina generan valores que representan los valores externos que son generados para el robot. Valores aleatorios son generados y entregados a la máquina de estados, simulando una conexión con los valores externos. Para este trabajo se utiliza un fuzzer del tipo *black-box* para simular los cambios en los datos externos que el robot obtiene que no son necesariamente valores válidos, aunque sean del tipo de datos esperado. Estos valores no dependen de qué estado se está probando, sino de qué máquina de estados se está probando. Si la máquina lee valores de un sensor, estos valores deben generarse en paralelo a la ejecución de este estado y deben ir modificándose en tiempo de ejecución.

Cuando se trabaja a nivel de máquina de estados es importante notar que pueden existir entradas para inicializar la máquina. Estas entradas igualmente deben ser generadas por el *fuzzer* para realizar las pruebas en profundidad. Una vez que se ejecuta una prueba completa de la máquina de estados es necesario reiniciar todos los valores iniciales de la máquina de estados y no volver a ejecutarla como tal con una entrada diferentes, por lo que se vuelve a instanciar la máquina de estados.

El alcance de esta tesis no abarca en su totalidad este tipo de pruebas, dada la complejidad de realizarlas de manera satisfactoria. Aún así, existe al menos un trabajo[2] complementario al enfoque presentado en este trabajo que permite generar datos para algunas de las entradas externas del robot, siendo posible emular algunos de los sensores que estos podrían presentar.

Es importante diferenciar entre estos dos tipos de pruebas ya que generan dos análisis completamente diferentes, las pruebas a nivel de estado pueden detectar errores con los datos de entrada del estado, pero no analizan si alguna ejecución podría llegar a generar ese valor de entrada al estado. Lo que permite también comprender el alcance del estado programado y definir el contexto de uso que podría tener al enlazarlo en una nueva máquina de estados. Por su parte, las pruebas a nivel de máquina permiten comprobar que valores definen caminos dentro de la máquina de estados, pero no permiten conocer si existe alguna combinación especifica que podría generar un error o un mal comportamiento al no tener un mayor control de los valores evaluados en la ejecución. Un análisis más profundo se puede generar al realizar pruebas a nivel de estados y luego realizar una prueba a nivel de máquina intentando generar los caminos para reproducir alguna combinación de las pruebas realizadas a los estados, por ejemplo, intentar llegar a un estado con ciertas entradas para ver si son valores válidos o no.

[2] fuzzer para ROS `https://github.com/aliasrobotics/ros1_fuzzer`

Capítulo 5

Experimentos

La hipótesis de este trabajo de tesis es que la técnica de *fuzz testing* puede detectar errores en comportamientos complejos de un robot. Para afirmar o refutar esta hipótesis se plantean tres preguntas en la Sección 1.2 que cubren la naturaleza de los problemas identificados (PI1), la dificultad de encontrar estos problemas (PI2) y la gravedad del error en el comportamiento robótico (PI3). Este capítulo describe el diseño experimental y la metodología utilizada para responder a las preguntas de investigación y verificar la hipótesis previamente planteada. La primera sección describe el contexto donde se realizó el experimento. La segunda sección describe el paso a paso de la metodología ocupada. Por último, la tercera sección describe el reporte utilizado para analizar las situaciones ecnontradas.

5.1. Contexto del experimento

Para el desarrollo de este trabajo de tesis se contó con el apoyo del equipo de robótica UChile Peppers de la Universidad de Chile. Este equipo de robótica utiliza un robot Pepper producido por la empresa SoftBank Robotics para mejorar la interacción humano-robots en ambientes domésticos[1].

El equipo ha recibido múltiples premios de la competencia RoboCup@Home[2]. RoboCup@Home es una competencia que se centra en la aplicación de robots autónomos en la vida diaria y en las interacciones humano-robot. Como ocurre en la mayoría de los equipos de robótica desarrollados en ambientes universitarios, gran parte del código fuente del software que utilizan está escrito por estudiantes de pregrado y postgrado durante el desarrollo de sus trabajos de tesis. Por lo tanto, es crucial probar cada comportamiento de manera rigurosa y exhaustiva para seguir siendo competitivo a nivel internacional. El equipo UChile Peppers permitió el acceso a sus repositorios de software que controlan el robot para realizar este trabajo de tesis. Es necesario destacar que al momento de comenzar este trabajo de tesis y durante todo el desarrollo, el autor no participó en el equipo UChile Peppers como colaborador directo, realizando consultas al equipo solo para entender algunos aspectos de la arquitectura del software escrito.

[1] https://uchile-robotics.github.io/bender-index.html
[2] https://athome.robocup.org/awards

Para maximizar la cantidad de comentarios sobre el experimento se restringió la evaluación del modelo a pruebas a nivel de estado para obtener una mejor descripción de los errores encontrados y poder revisarlos en un menor tiempo en comparación a hacer un análisis completo de pruebas a nivel de máquina y a nivel de estado de los comportamientos robóticos.

El equipo de desarrollo de UChile Peppers varía año a año, incorporando nuevos miembros, pero viendo como los de mayor permanencia se retiran. Durante este trabajo de tesis se vieron múltiples cambios tanto en la dirección del equipo como en sus miembros.

5.2. Metodología

Para realizar los experimentos del *fuzzer* y responder a las tres preguntas de investigación creadas se diseñó la metodología presentada en la Figura 5.1. Esta sección detalla cada paso de esta metodología.

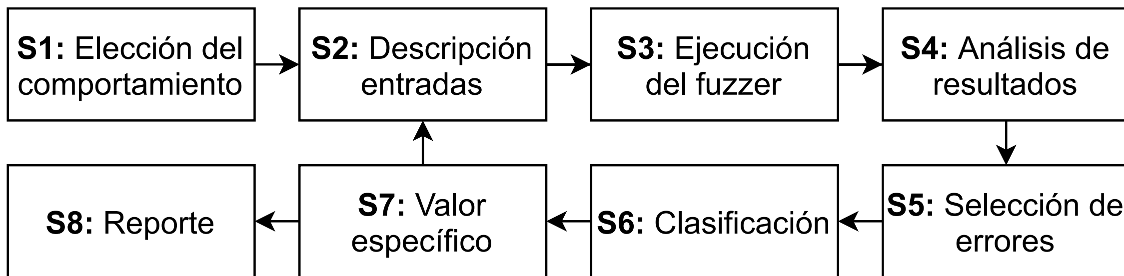

Figura 5.1: Pasos de la metodología para probar un comportamiento

S1: Elección del comportamiento. Un robot como el utilizado por el equipo UChile Peppers es complejo y permite un gran rango de comportamientos específicos diferentes. El primer paso de esta metodología involucra la identificación del conjunto específico de comportamientos que se probarán. Es importante identificar qué comportamiento específico se está probando y determinar claramente cuáles son las salidas esperadas de este conjunto. Como tal, es importante que los comportamientos específicos sean deterministas y puedan volver a ejecutarse una vez terminada su ejecución. Para este trabajo fue necesario que los comportamientos seleccionados contaran con una descripción precisa sobre los componentes del robot que se utilizan como pueden ser los sensores del robot.

S2: Descripción entradas. Las entradas válidas para el comportamiento deben ser adecuadamente descritas y caracterizadas para poder aplicar la técnica de *fuzz testing* a un estado específico o a la máquina de estados completa. La identificación involucra (i) conocer el nombre de las entradas, (ii) conocer el tipo de datos utilizado por el código del estado (se puede utilizar la técnica de monitoreo de estado descrita en la Sección 4.4.1) y (iii) si es una entrada externa o interna para realizar adecuadamente la generación de valores por el *fuzzer*. El resultado de este paso es una descripción clara e inequívoca de la entrada. El *fuzzer* también esta ajustado para producir entradas estructuradas correctamente mediante el diseño de la gramática mostrada en la Figura 4.3.

En caso de que el *fuzzer* produzca algunos valores de un tipo equivocado o mal definidos (por ejemplo, genere números positivos para una entrada que esperaba solo números negati-

vos), se necesitarán nuevas ejecuciones iniciales o ajustar la gramática para poder producir estos valores más específicos.

En esta etapa es donde se crea el archivo de configuración descrito en la Figura 4.5 con la información de los estados de la máquina a probar.

S3: Ejecución del fuzzer. El tercer paso consiste en ejecutar el *fuzzer* en cada estado de la máquina de estados seleccionada. Para ejecutar un estado, se deben generar los valores de sus entradas utilizando la gramática y la información de sus tipos descrita previamente. El *fuzzer* debe construir un *string* desde la gramática y luego interpretarlo como un tipo de dato de Python para poder entregar el *userdata* que corresponde con la entrada generada.

Cuantas más veces se prueba un estado (se ejecuta con diferentes valores generados), la probabilidad de encontrar una falla o un comportamiento errado incrementa. Una ejecución de un estado representa una unidad incremental de exploración del espacio de valores de entrada plausibles, aumentando así la probabilidad de desencadenar una falla o un comportamiento no esperado.

El número de ejecuciones de cada estado esta especificado mediante uno de los hiper parámetros asociados al proceso de *fuzzing*. En los experimentos realizados en este trabajo, se ejecutó cada estado 100 veces por cada ejecución del *fuzzer*. Para algunos comportamientos fué necesario realizar al rededor de 30 ejecuciones del *fuzzer* para obtener bien la información de los tipos de datos de sus entradas. Este valor arbitrario produjo buenos resultados en los experimentos, pero incrementándolo podría llegar a generar muchos más casos si existe alguna arquitectura con mayor poder de cálculo.

La salida de este pasos S3 es un archivo para cada estado que contiene el registro de cada ejecución del estado de la forma: (i) los valores de las entrada de la ejecución, (ii) la salida del estado, (iii) errores si existió alguno, (iv) tiempo de ejecución del estado, y (v) la pila de ejecución del error si existió. Esta información se guarda en un archivo JSON para poder inspeccionar visualmente. El archivo corresponde a una ejecución del *fuzzer* para el estado. El archivo se genera en una carpeta con el nombre del estado y se le asigna el nombre al archivo con el tiempo de cuando se crea, por lo que debería ser único si se reproduce múltiples veces el *fuzzer*. Cada una de las ejecuciones del estado se almacena como un objeto de JSON, separando cada uno de los diferentes campos como un parámetro con llaves previamente definidas.

S4: Análisis de resultados. Luego de haber ejecutado los estados, los registros generados deben ser revisados detalladamente para identificar anomalías en la ejecución del comportamiento. Este trabajo propone los siguientes métodos para detectar estas anomalías en la ejecución:

(i) Buscar los registros con errores y pilas de ejecución de los errores.

(ii) Comparar las salidas de los estados con ejecuciones que se conocen son correctas.

Para (i) es simple, solo basta buscar en el archivo creado el valor del campo `output` para conocer si la salida es un error o solo un *string*. Esta parte puede ser automatizada utilizando

expresiones regulares que caractericen la presencia de errores en la salida, por ejemplo, encontrar la palabra `Traceback` que puede significar la presencia de un error en la máquina de estados. Un ejemplo para (ii) es comparar aquellas salidas obtenidas en ejecuciones previas del comportamiento o pruebas realizadas con el robot real sobre el mismo comportamiento con las obtenidas por el *fuzzer* en estas ejecuciones.

De igual manera, mientras se ejecuta el *fuzzer*, es posible revisar la consola para detectar errores o advertencias que ROS o SMACH puedan generar. Esto es vital si SMACH encuentra un error que no es crítico pero que podria ser una mala práctica o un uso incorrecto de las propiedades de los estados. Si no es un fallo crítico que termine la ejecución del estado esta información no se verá reflejada en el reporte y quedará solamente en la consola.

S5: Selección de errores. En el paso anterior es posible que se generen muchas anomalías en los diferentes estados o comportamientos. Por lo tanto, es necesario filtrar estas situaciones para identificar cuáles verdaderamente se consideran un problema del software en sí. Un caso claro es que múltiples valores generen el mismo error, en este caso no son múltiples errores si no diferentes valores los que generan el mismo error.

Otro caso posible es que el estado tenga una error en la sintaxis de su código fuente y éste afecte a todos los estados escritos en el mismo archivo. Todos los estados registraran el mismo error, por lo que no será efectiva esa prueba para los estados afectados hasta que se solucione el error.

Para estos casos con la inspección de la salida del estado en el archivo de reporte y una mirada de manera superficial al código fuente del estado es posible determinar si el error es el mismo o no.

Para considerar una situación como error de software, se caracterizó la situación y se verificó si la situación influía en la ejecución del estado, modificando la salida o generando un error. Algunos de las situaciones que se detectaron no se reportaron como errores de software, ya que no influían en la ejecución del estado. Un caso particular se dio con una situación donde el estado declaraba ciertas entradas pero no las utilizaba, por lo que la situación fue reportada pero dejada fuera del experimento al no generar errores ni una mala ejecución del comportamiento.

S6: Clasificación. Luego de descartar todos los casos que no implican necesariamente un error en la ejecución del estado o un comportamiento inesperado, es necesario determinar qué causó ese comportamiento inesperado o generó el error. Para ello se debe determinar si las entradas eran las correctas o existe un error en el código fuente del estado del comportamiento.

Se clasificaron los errores detectados según la clasificación presentada por Zhao *et al.* [44], la cual cuenta con múltiples formas de categorizar un error de software. Adicionalmente se agregó una categoría que involucra el contexto robótico: errores de configuración del robot. Es importante notar que al ser un software, los errores son similares a los que pueden existir en cualquier software, pero el contexto robótico le añade el componente que puede generar nuevos tipos de errores. Aun así la mayoría de los errores que se buscan con el *fuzzer* son errores de software.

```
def execute(userdata):
    text = userdata.confirmation_text
    if 'yes' in text:
        return "yes"
    elif 'no' in text:
        return "no"
    else:
        return "aborted"
```

Figura 5.2: Ejemplo de la función *execute* de un estado de SMACH.

S7: Valor específico. Es posible que el análisis realizado sobre el tipo de parámetros de entrada no fuese suficiente para caracterizar bien la entrada. Existen casos donde el tipo de entrada no es suficiente para su caracterización, como posibles valores que no serán nunca generados dado el gran espectro de valores que la gramática puede generar o que la estructura de la entrada será siempre fija y no recibirá ningún otro valor. Ambos casos generan el mismo problema: errores que no son errores del todo pero que pueden pasar los filtros antes descritos porque podrían representar un verdadero error. Con la metodología de monitoreo de estado, si existe algún problema con el tipo de las entradas será necesario repetir desde el paso S2, para ello es necesario redefinir la gramática utilizada a una más específica o entregar el tipo correctamente.

Un ejemplo se puede ver en la Figura 5.2, un comportamiento que posee dos opciones de valores para el programador y cualquier otro valor retornará el valor por defecto. Si utilizamos una gramática general posiblemente obtendremos los resultados esperados, pero si generamos una gramática que genera con las letras `e, o, n, s,` y *strings* de 2-3 caracteres encontraremos algunos casos donde el comportamiento no realiza lo esperado por parte de los programadores. El ejemplo claro es con valores como `eno` o `noe` que cumplen el largo y los caracteres, pero el comportamiento responde similar al *string* `no`, mientras que lo esperado es que responda con un *string 'aborted'*.

Como el software que define un comportamiento robótico crece con el tiempo, es posible que el programador tenga poco conocimiento sobre que se necesita para utilizarlo. Este paso permite realizar pruebas sin mucho conocimiento del código o funcionamiento particular del estado. Se puede iterar sobre estos pasos una y otra vez hasta dejar de encontrar errores de tipos de datos y revisar si esos tipos generan algún problema en el código o no. Existe la posibilidad de que dado un tipo incorrecto el estado retorne una salida. Para estos casos se considera que el comportamiento posee una vulnerabilidad o no utiliza las entradas especificadas, lo cual podría deberse a que esa entrada va a ser requerida por un estado posterior o simplemente es una entrada que no es utilizada. Es importante detectar estos casos para un posterior análisis de la máquina completa y si se requiere, de las interconexiones en la misma máquina.

S8: Reporte. En esta última etapa el autor de este trabajo, como agente externo al equipo de UChile Peppers, crea un reporte sobre cada situación qué considera que es una posible vulnerabilidad o error del software para analizarlo con el equipo de desarrollo del robot. Esta

etapa consiste en resumir la información obtenida con el *fuzzer*, interpretarla para poder obtener un reporte detallado, e idealmente encontrar una posible solución a la situación. Los datos que se presentaron en este trabajo fueron: la información utilizada por el *fuzzer* para las entradas, el estado ejecutado, el error o salida obtenida y los registros generados en la consola de ejecución. Si el estado pertenecía a alguna máquina que pudiese dibujarse de manera simple, se realizó un pequeño bosquejo con la estructura simple de alguna de las máquinas donde estaba presente. La selección por parte del autor se realizó para evitar posibles conflictos de intereses de los participantes del experimento con una posible autoría del código fuente a revisar.

5.3. Reporte al equipo

Para analizar la hipótesis y las preguntas de investigación de la mejor manera se revisó cada situación identificada, en el paso 8 de la sección anterior, y se realizaron las siguientes preguntas al equipo de desarrollo de los comportamientos robóticos:

- *P1 - ¿Crees que esta situación es un problema de software?*
- *P2 - ¿Tienes familiaridad con la situación?*
- *P3 - ¿Crees que hubieras podido encontrar la situación por tu cuenta?*
- *P4 - ¿Has visto esta situación antes?*
- *P5 - ¿Qué tan difícil crees que es encontrar esta situación?* Respuesta en rango de 1 (fácil) a 5 (difícil)
- *P6 - ¿Es importante para la competencia RoboCup?* Respuesta en rango de 1 (no importante) a 5 (muy importante)

El cuestionario fue realizado a través de la plataforma Google Forms para almacenar las respuestas de manera centralizadas. El cuestionario se diseñó para recolectar opiniones sobre las situaciones encontradas de manera no sesgada. Cada miembro del equipo se encuestó de manera individual y no en grupo para evitar la influencia de otros miembros en sus respuestas. Las preguntas realizadas y el cuestionario en si nunca presenta las situaciones encontradas como errores para evitar influir en sus respuestas. En su lugar se utiliza el termino genérico de "situación"para darle neutralidad y no influir en las respuestas sobre los descubrimientos.

Cada participante del experimento recibió y evaluó tres situaciones. Se les indicó a cada participante la ubicación exacta del error en el código fuente, los valores de entrada del estado y la salida obtenida del estado o el error generado.

Para evitar que la fatiga o presión atendiendo el contexto de estudiantes de la mayoría del equipo, la cantidad de tiempo fue limitado a 20 minutos máximo por situación, generando una reunión de aproximadamente 1 hora con cada integrante para revisar los errores encontrados. El siguiente capítulo presenta los resultados obtenidos tanto por el *fuzzer* como las respuestas obtenidas de la aplicación del cuestionario.

Capítulo 6

Análisis Experimentos

En el presente capítulo se detallan los resultados de las sesiones de entrevista con los diferentes miembros del equipo UChile Peppers. Primero se muestra el proceso de *fuzzing* de los comportamientos, cuantos fueron probados, la cantidad de situaciones encontradas y como fueron revisadas por cada participante. Luego se responde a la primera pregunta de investigación con una caracterización de las situaciones encontradas. Posteriormente se responde a la segunda pregunta de investigación con un caso particular en que el fuzzer encontró una situación que se había generado en una ejecución real del comportamiento. Para finalizar con las preguntas de investigación se realiza un análisis sobre la tercera pregunta de investigación y como difiere el punto de vista del autor de este trabajo con el del equipo de desarrollo sobre la complejidad de las situaciones. El Capítulo finaliza con una sección dedicada a comentar la validez interna y externa del experimento realizado, haciendo una revisión de los participantes, los comportamientos, las sesiones, las situaciones e incluso del software analizado.

6.1. Datos del experimento

La propuesta metodológica presentada en la Sección 5.2 de este trabajo de tesis fue utilizada en el software desarrollado por el equipo UChile Peppers. El software probado con esta metodología está compuesto por 124 estados diferentes, los cuales componen 6 diferentes comportamientos robóticos utilizados en la competencia RoboCup. Se utilizó únicamente el software desarrollado por el equipo UChile Peppers por ser un software probado en competencias internacionales con destacadas participaciones y permitir un acceso sin restricciones para el desarrollo de esta tesis. La mayoría de los equipos que compiten en RoboCup mantienen en privado sus comportamientos al ser lo más importante para la competencia, ya que se compite sobre la misma plataforma robótica.

En total se utilizaron solo 6 comportamientos de los 17 comportamientos que poseía el software para realizar las pruebas debido al trabajo que había que hacer sobre cada estado para poder caracterizar las entradas. El objetivo es validar la metodología y obtener resultados que permitan responder la hipótesis de este trabajo.

ID Part.	Exp. [años]	Nivel	Área	Errores
P1	4	Est.	CC	E3, E4, E5
P2	1	Est.	IE	E1, E2, E6
P3	5	Est.	IM	E2, E3, E5
P4	3	Est.	IE	E1, E2, E4
P5	4	Est.	IE	E2, E3, E6
P6	5	PhD/Prof.	CC	E1, E3, E4
P7	6	Prof.	IE / CC	E2, E5, E6

Tabla 6.1: Información de los participantes (Est. = estudiante de pregrado, Prof. = profesional, CC = Ciencias de la Computación, IE = Ingeniería Eléctrica, IM = Ingeniería Mecánica)

El *fuzzer* operó de manera específica en 24 diferentes entradas internas utilizando gramáticas diseñadas para estos valores y en 9 entradas externas con diferentes valores de entrada. El resto de las entradas se utilizaron valores genéricos ya que algunos estados no utilizaban los valores de las entradas, siendo estados por los que el tipo de dato solo era enviado a un siguiente estado, sin analizar el contenido de estos valores.

En total se identificaron 6 errores de software que se consideraron anomalías en el comportamiento robótico o simplemente un error en tiempo de ejecución del comportamiento. Solo el autor de este trabajo revisó estas situaciones antes de mostrarlas a los miembros del equipo, por lo que no se descarta que existan algunos comportamientos errados en la ejecución de los estados u otros tipos de errores que no fueron detectados como tal, pero que el equipo podría considerar como una mala ejecución. Este análisis sería un aporte que permite detectar los errores más difíciles de encontrar que son aquellos que no generan errores de código. Sin embargo, debido al extenso trabajo que puede demandar del equipo de desarrollo y sumado al hecho de haber encontrado errores con el *fuzzer*, se prefirió no realizar el análisis de malas ejecuciones y enfocar el tiempo disponible del equipo de desarrollo en revisar las situaciones encontradas. Estos errores se presentaron a 5 miembros del equipo de desarrollo y a 2 ex-miembros que habían dejado el equipo luego de la edición previa de la competencia RoboCup.

Cada uno de los errores fue revisado por al menos 3 participantes diferentes para obtener diferentes percepciones sobre la misma situación. La Tabla 6.1 resume la distribución de los diferentes errores a cada uno de los participantes. Para facilitar el análisis y poder obtener comentarios no expresados en el formulario, se mantiene un registro de todas las sesiones. Por problemas técnicos, uno de los registros no presenta audio.

Si bien en la Sección 5.3 se habla de situaciones, en este capítulo se cambia esa referencia a errores ya que fueron confirmados y no representan sesgo en este punto para los participantes.

6.2. Caracterización del error

Dentro de los errores encontrado se aprecian algunas características interesantes y que permiten diferenciar entre ellos mismos a los errores. Una pregunta necesaria de responder como pregunta de investigación de este trabajo de tesis es:

PI1: *¿Cuáles son las características de los errores identificados utilizando fuzz testing?*

Los 6 errores encontrados produjeron (i) que el estado no se pudo ejecutar de manera completa o (ii) se generó algún error o advertencia a través de los registros de SMACH o ROS. Estos 6 errores se clasificaron en 4 categorías: *errores de sintaxis*, *errores en manejo de datos*, *errores lógicos* y *errores de configuración de la arquitectura*. Las primeras 3 categorías se adoptaron del trabajo de Zhao *et al.* [44] para clasificar soluciones a errores.

Errores de sintaxis. Se caracterizan como errores de sintaxis las secuencias de caracteres que no pueden ser interpretadas por Python. Los errores de sintaxis son comúnmente producidos por gente no experta desde (i) ambientes de programación para Python con un trabajo escaso en notificar al desarrollador sobre la presencia del error (por ejemplo, algunos editores de texto subrayan las partes donde existen errores de sintaxis), (ii) los programadores de comportamiento robóticos en general son personas ajenas al área de ingeniería de software y por lo tanto sin una habilidad en el uso de herramientas de análisis de software. Un error de sintaxis no ocurre en tiempo de ejecución sino al momento de intentar ejecutar la primera instrucción en el archivo con error. Considere el siguiente extracto de código en Python:

```
action_mapping = {"left": "I am going left"
    "right": "I am going right"}
```

El código contiene un error de sintaxis dado que falta una coma para separar las partes: `"I am going left"` y `"right"`. Las limitaciones del robot Pepper no permiten buenas prácticas para evitar estas situaciones, el código debe cargarse al robot a través del protocolo *ftp* y provoca que cualquier cambio deba realizarse tanto en los repositorios del equipo como en el robot. Dado el tiempo que toma realizar la carga y pruebas del código en el robot, algunas de estas situaciones son solucionadas directamente en el robot, pero no en el repositorio general. Poder detectar estas situaciones antes de cargar el código al robot le permite al equipo de desarrollo cargar código sin este tipo de errores, por lo tanto, reduce la necesidad de hacer modificaciones en el código del robot para solucionar problemas simples y reduce los problemas de sincronización entre el robot y el repositorio. El *fuzzer* identificó uno de estos errores, el cual no se detectó durante el desarrollo, pero si se detectó durante las pruebas realizadas en la competencia, la resolución de este problema se hizo directamente en el robot y no en el repositorio del software.

Errores en manejo de datos. Al utilizar un lenguaje dinámicamente tipado, Python no ofrece una seguridad para prevenir el uso incorrecto en tipos de datos primitivos. Es frecuente ver colecciones de elemento heterogéneos utilizados para programar, por ejemplo, la tupla `('move_left', 15)` puede ser la entrada de uno de los estados. El estado debe asumir que el primer elemento de la tupla es un *string* y el segundo es un entero. Si no son un *string* y un entero, el código probablemente generará un error por manejo de datos.

Los errores por manejo de datos ocurren cuando los datos no son manejados de manera adecuada por el estado. En este caso, el error puede estar al llamar al siguiente estado (por ejemplo, se entrega un valor de diferente tipo al estado siguiente), o en el estado que se está ejecutando (por ejemplo, los tipos de datos son correctos, pero no son manejados bien). Si se considera el siguiente código:

```
# Originalmente, list_objects es un arreglo de enteros.
# En una nueva version, list_objects es un nombre de una ubicacion.
a_list = userdata.list_objects # Si list_objects = "Door"
position = (a_list[0], a_list[1])
return position
# Retornara ("D","o") y es posible que retorne
# un error en el siguiente estado.
```

En una versión previa del código la variable `userdata.list_objects` contenía las coordenadas de la posición en un espacio de dos dimensiones. Luego de algunos cambios en el código, la variable ahora provee una etiqueta indicando un objeto en vez de coordenadas.

También se clasifican como error de manejo de datos situaciones donde un parámetro declarado no es accesible, o si algún campo no existente es tratado de ser accedido (por ejemplo, `a_list = userdata.non_existing_field`). Se encontraron 3 errores donde los datos no estaban bien definidos o el estado intentaba acceder a datos no existentes.

Los errores del tipo de manejo de datos pueden ser complejos de encontrar y de depurar ya que usualmente requieren un sólido conocimiento acerca del estado que genera el dato, el dato que se utiliza y el estado que utiliza esta información. Más aún si existe información que debe pasarse a través de varios estados, pero sin manipularse en ellos. El uso del *fuzzer* genera una ayuda inmediata si logra detectar alguno de estos casos.

Errores lógicos. Una incorrecta evaluación de una instrucción en un condicional o en un ciclo generar un error lógico. Durante la ejecución, un error lógico puede expresarse por la ejecución de una rama que no debería en un condicional (por ejemplo, que se ejecute la rama `else` en lugar de la rama `if`). Considere el siguiente código:

```
# text_confirmation='Yes'
if userdata.text_confirmation=='yes':
  return 'yes'
elif userdata.text_confirmation=='no':
  return 'no'
return 'aborted'
```

El código contiene un error lógico ya que la variable `text_confirmation` contiene el valor 'Yes' (con mayúscula), mientras que la primera condición se expresa 'yes' (con minúsculas). Como resultado de la ejecución el código retornará 'aborted' y probablemente el programador esperaba que retorne 'yes'. Si esta situación pasa en medio de una ejecución real, podría mal interpretar la confirmación y por lo tanto realizar un comportamiento no esperado.

Desde la experiencia obtenida con el análisis del software del equipo UChile Peppers, los errores del tipo lógicos son usualmente difíciles de identificar y de encontrar. Una de las razones de esta dificultad es el hecho de que un error lógico no generará una caída de la aplicación ni un error en tiempo de ejecución. En su lugar, en el experimento realizado, el error lógico se expresó como un comportamiento inesperado del robot. El ejemplo anterior se puede producir por muchos factores, uno de ellos puede ser que al decir oralmente 'yes' al

robot, este lo transcriba a 'Yes' por ser la primera palabra que escucha en lugar de 'yes' por el módulo de voz a texto. Esto genera claramente un comportamiento no esperado pero que tiene un error en la interpretación del valor recibido. Identificar este tipo de error involucra comparar las salidas de los estados con una correcta ejecución del estado con configuraciones controladas. Durante el experimento se encontró un caso de este tipo de error.

Errores de configuración de la arquitectura. Para realizar un comportamiento sofisticado como interactuar con una persona, tomar un pedido y luego entregárselo a la persona, el robot no solo debe tener el código específico de cómo realizar esta tarea, sino que también depende de diferentes componentes independientes de él, como por ejemplo un módulo de voz a texto y luego interpretar esta instrucción. Cuando un robot se inicializa se realizan las mismas operaciones que en un computador, una larga lista de componentes se inicializa para poner en funcionamiento el computador completo, más aun, cuando una aplicación se ejecuta se realizan algunas revisiones de componentes o módulos.

Durante el experimento se encontraron situaciones que no eran relativas a la máquina de estados en sí, sino a la arquitectura de la configuración del robot. Al generar la máquina de estados y el *mock* del robot, se requiere una serie de funcionalidades que el robot necesita activar. Dada una cierta configuración de estas funcionalidades, la inicialización generaba una larga lista de registros producidos tanto por Python como por ROS. Luego de haberse inicializado no se notaban anomalías ni efectos en base a estos registros. Esta situación (E6) fue asignada deliberadamente a revisión por uno de los ex-miembros del equipo con mayor conocimiento en el robot completo, las apreciaciones sobre esta situación son que algunos componentes inicializan algunas habilidades más de una vez, en circunstancias no obvias. Como resultado se generan varios registros que no causan ninguna perdida de funcionalidad en el robot. Si bien esta situación ocurrió con más de una configuración, se considera como una única situación al ser el mismo error con diferentes formas de generarlo. No se comprobó ni se descartó que no pueda generar errores a futuro ya que la cantidad de posibles combinaciones que generaban situaciones similares es compleja de analizar. La situación es generada en su totalidad por fallas en la construcción de la arquitectura del software del robot.

> ***Respondiendo a la PI1.*** En este experimento se encontraron cuatro tipos de errores: sintaxis, manejo de datos, lógicos y de configuración de arquitectura. Es posible que se puedan encontrar otros tipos de errores, por ejemplo, que involucren el incorrecto uso de alguna API o incluso errores con los drivers de los sensores si se ejecuta con un robot real. Sin embargo, en este experimento en particular no se encontraron ese tipo de situaciones.

6.3. Errores reales

Uno de los focos de esta investigación es determinar si es posible detectar situaciones o errores que podrían comprometer el normal funcionamiento de un comportamiento robótico antes de ejecutarlo. Para ello se realizó el cuestionario presentado en la Sección 5.3 a los integrantes del equipo UChile Peppers. Este cuestionario se generó para recolectar opiniones y *feedback* sobre las situaciones encontradas para evaluar, en parte, que tan reales eran las situaciones.

Por ello se definió la segunda pregunta de investigación:

PI2: *¿Puede el fuzz testing detectar problemas en comportamientos robóticos que sean realista y representativos?*

Error	Q1	Q2	Q3	Q4	Q5	Q6
E1	3/3	0/3	2/3	3/3	2.7	5
E2	5/5	2/5	5/5	3/5	3	4.8
E3	3/4	3/4	3/4	3/4	2.75	4.5
E4	2/3	1/3	3/3	3/3	1.7	4
E5	3/3	3/3	3/3	3/3	2.3	4.7
E6	3/3	1/3	1/3	2/3	2.3	4.5

Tabla 6.2: Resultados del experimento. Las preguntas están listadas en la Sección 5.2. En X/Y, X = número de "Si" e Y = número total de respuestas. Puntaje promedio es asignado a Q5 y Q6. Se utiliza Qx para no confundir con Px de la Tabla 6.1

Las respuestas al cuestionario presentado en la Sección 5.3 están presentes en la Tabla 6.2. Las situaciones mostradas a los participantes del cuestionario son identificadas en su gran mayoría como un problema de software (Q1). Los errores identificados eran conocidos por los participantes, pero aún no resueltos (Q4). El reporte de los errores se percibe como importante o muy importante (Q6).

El hecho de poder construir casos de prueba repetibles utilizando los valores generados por el *fuzzer* es percibido como algo valorable para los participantes. Por ejemplo, E2 es un error de manejo de datos que ocurrió cuando una variable no declarada se utilizó en una cláusula de *if-else*. El participante P3 conocía el error E2 dado que se generó durante la RoboCup 2018. El error se solucionó en la competencia, pero dada las condiciones de competencia y el problema con la arquitectura del robot Pepper, la solución nunca llego al repositorio. A pesar de eso, el fuzzer fue capaz de encontrar este error que previamente había aparecido en una ejecución real.

Un estudio reciente realizado por Afzal *et al.* [1] muestra como aún en la industria es difícil encontrar herramientas de pruebas que se adapten al contexto de la robótica por todas las componentes que involucran.

Respondiendo a la PI2. Es posible encontrar errores o situaciones donde el comportamiento robótico no se comporta como se espera utilizando la técnica de *fuzz testing*. Más aun, es posible encontrar problemas reales y que representan o representaron un problema. Es posible afirmar que, utilizando esta técnica, cualquier situación que se encuentre puede ser una posible situación donde el robot podría fallar a futuro si no se soluciona dada las condiciones especiales en las que se prueba.

6.4. Dificultad en encontrar los errores

Intuitivamente, un error fácil de encontrar debería ser fácil de resolver. De manera opuesta, un error que se encuentra escondido en el software y que no se ha detectado, puede llegar

a ser muy difícil de resolver ya que pueden existir pruebas y código que están construido sobre esta base con problemas. Entender la dificultad de los errores encontrados según los desarrolladores del robot es una tarea importante. La tercera pregunta de investigación se planteó como:

PI3: *¿Puede el fuzz testing detectar problemas difíciles de encontrar?*

El uso del *fuzzer* y el cuestionario muestran que, aunque los participantes no tengan una conexión particular con los errores (Q2), ellos creen que podrían haber encontrado los errores por su cuenta (Q3) sin un gran esfuerzo de tiempo (Q5). A pesar de ello, estos errores no fueron descubiertos o reportados antes del experimento salvo por el error E2, lo que hace indicar que no son errores tan simples de encontrar.

Una de las conclusiones que deja este experimento, probablemente intuitiva, es que la experiencia de los participantes contribuye directamente a la habilidad de identificar errores. Un participante con una gran experiencia en Python y el uso de *frameworks* en este lenguaje percibe los errores de sintaxis como algo muy fácil de encontrar con el uso de un *linter* o editores de texto. El experimento confirma esta pequeña hipótesis al mostrar que los participantes con mayor experiencia demostraban conocer y comprender los errores de manera más rápida que aquellos más nuevos. De igual manera un participante con experiencia está más familiarizado con herramientas de *debugging* e inspección de memoria, lo cual parece ser una clave para identificar y ubicar los errores de datos.

Los errores de tipo lógico parecen ser los tipos de errores más difíciles de encontrar e identificar. Sin generalizar el caso encontrado, algunos de los participantes indicaron que el error de tipo lógico es un error difícil de encontrar. Encontrar un error del tipo lógico no necesariamente implica un error en el código fuente, es decir, no se encuentra mirando la consola de ejecución, sino que analizando por qué el robot se comportó de una manera diferente de la esperada. Comprobar que este tipo de problema está solucionado puede tomar bastante tiempo considerando que los pasos a seguir son: (i) Detectar el problema en el comportamiento, (ii) encontrar donde se genera el problema, (iii) identificar que genera el error, (iv) corregir el error y (v) comprobar que el error se resolvió. Se requieren al menos 2 ejecuciones del comportamiento para encontrar y comprobar que se resolvió el problema. Como ha sido explicado anteriormente, la ejecución de un comportamiento es una tarea costosa en tiempo y recursos por lo que realizar dos ejecuciones cada vez que se encuentra un error de este tipo es considerado un problema.

Dado que un error difícil de detectar también es difícil de resolver, su comprobación podría ser también difícil de hacer. El principal problema para comprobar la resolución de un error es la reproducibilidad de una ejecución. Se deben reproducir tanto los valores internos como externos para poder determinar si la solución funciono o no.

> ***Respondiendo a la PI3.*** En general, es posible responder a la PI3 afirmando que el *fuzzer* identificó errores que se perciben como medianamente difíciles de encontrar, por lo tanto, no se consideran difíciles de detectar a pesar de estar presentes durante mucho tiempo y no haber sido reportados por el equipo.

6.5. Validez del experimento

Un punto importante a considerar en toda investigación que conlleve experimentación son posibles factores que afecten la validez del experimento, por ello es necesario realizar ciertas aclaraciones sobre estos factores para generar mayor credibilidad en el estudio.

La validez de un experimento se puede separar en dos partes: validez interna y validez externa. A continuación se analizan ambas por separado.

6.5.1. Validez interna

La validez interna se refiere a la confianza que se puede tener sobre cómo las variables independiente son responsables de cambios en las variables dependientes del experimento.

Participantes. Para reducir la posibilidad de que los participantes no fuesen capaces de revisar las situaciones presentadas, se seleccionaron solo miembros del equipo que poseían un mínimo de experiencia con maquinas de estado, es decir, que al menos hayan programado alguna máquina de estados para los comportamientos del robot. Estos participantes eran variados en su experiencia tanto con programación como en el tiempo que llevan en el equipo, produciendo una mezcla homogénea desde participantes expertos en el software, nuevos mentores del equipo y por último nuevos miembros del equipo. Las situaciones a revisar fueron distribuidas de manera aleatoria para evitar que el autor, de manera deliberada, asignara a participantes comportamientos donde fuesen autores directos. Por último una de las situaciones fue designada a la persona con mayor experiencia en el robot y sus comportamiento deliberadamente ya que ni los primeros revisores ni el autor pudieron encontrar una explicación para esa situación. Esta situación es explicada en el párrafo ***Errores de configuración de la arquitectura*** en la Sección 6.2.

Situaciones revisadas. Las situaciones presentadas fueron filtradas por el autor de este trabajo antes de presentarlas al equipo de desarrollo. Estos filtros se basaron en eliminar toda situación que fuese similar a los problemas seleccionados, dejando un error de cada tipo para evitar presentar situaciones similares en el mismo estudio. Los datos obtenidos de otras ejecuciones del *fuzzer* no fueron revisados por el equipo, pudiendo haber existido situaciones donde el comportamiento funciona pero no realiza lo que debía hacer. Estas posible situaciones no influyen negativamente en la investigación ya que, de existir, solo aumentarían los posibles errores o generarían nuevos tipos de errores, lo que se considera un efecto positivo en el estudio realizado.

6.5.2. Validez externa

La validez externa se refiere a la capacidad de otros investigadores de reproducir y generalizar el experimento.

Participantes. El grupo de participantes escogido es parte del mismo equipo de desarrollo por lo que sus prácticas de programación son muy similares, salvo un participante, todos cursan o cursaron su pregrado en la misma facultad de la misma universidad. Si bien existen otros equipos que utilizan comportamientos robóticos para la misma competencia, es difícil

acceder a realizar colaboraciones si existe un potencial vínculo con estudiantes del equipo de robótica de la universidad. A pesar de ello se ve posible un buen funcionamiento de esta estrategia en cualquier comportamiento que utilice el mecanismo de *userdata* de SMACH como mecanismo de envío de información. La cantidad de participantes fue la mayor cantidad de personas que trabajaron en el robot desarrollando comportamientos, a pesar de existir otros desarrolladores que trabajaban en módulos mas complejos u otros equipos de robótica, estos no fueron convocados dada su falta de experiencia con máquinas de estados. Se considera un grupo razonable ya que el fin del experimento fue revisar situaciones donde el *fuzzer* que potencialmente podrían constituir un problema en ejecuciones reales y no cuantificar la cantidad de problemas encontrados.

Comportamientos del robot. El software utilizado para realizar las pruebas es un software que ha sido desarrollado por único equipo. A pesar de ello, el software utiliza las estructuras presentes en SMACH que permiten realizar un proceso de *fuzzing* bastante general y aún así logró encontrar errores importantes y significativos. Dentro de los equipos de la misma competencia RoboCup, existe un equipo que libera el software de alto nivel utilizado, sin embargo, sus máquinas y ambiente son mucho mas complejos que los utilizados por UChile Peppers, siendo una gran barrera de entrada para realizar pruebas de manera exploratoria. A pesar de ello, es posible realizar las pruebas en esos comportamientos si el ambiente es configurado apropiadamente y es posible ejecutar el software. También se ve posible la ejecución en otro software distinto de SMACH pero que utilice los mismos principios de máquinas de estados que comparten información a través de los estados.

Tipos de errores. Por una parte es posible decir que los errores de lógica y datos pueden ser generalizables a cualquier aplicación al ser errores normales en el ambiente de software. Posiblemente la generalización no se pueda realizar para los errores de sintaxis ya que son simples de encontrar por herramientas automáticas y su solución es simple. Los errores de la arquitectura del software se deben a errores en la inicialización de los módulos del robot, entendiendo que un robot es una máquina compleja que se asemeja más a un sistema distribuido que a un solo computador. Los errores en la inicialización de módulos podrían ocurrir en cualquier robot tan o mas complejo que el robot Pepper, o incluso en algunos robots menos complejos que utilicen arquitecturas similares. Se consideraron como un elemento dentro de la validez externa ya que otros investigadores podrían llegar a clasificar errores similares dentro de otras categorías, algunos son casos ambiguos ya que se generan por ciertas condiciones pero producen condiciones que pertenecen a otras categorías. En este experimento se utiliza una clasificación creada por Zhao *et al.* [44] y una categoría adicional que es generada por el contexto robótico.

Capítulo 7

Conclusión

Este último capítulo resume el trabajo realizado en esta tesis. Se presentan las contribuciones realizadas por el trabajo y los principales conceptos abordados como lo son: El uso de *fuzz testing* en robótica y las dificultades de generar casos de pruebas en robótica. Finaliza el capítulo detallando mejoras a esta técnica que podrían ser consideradas para futuros trabajos en la línea de pruebas para comportamientos robóticos.

7.1. Contribuciones

El uso de técnicas de ingeniería de software en el campo de robótica es reciente en contraste con la cantidad de trabajo existente en las otras áreas que componen la robótica como visión computacional, planificación de caminos (*path planning*) o manipulación. Este trabajo contribuye a mejorar esta área al diseñar, aplicar y evaluar una forma de realizar pruebas generadas aleatoriamente a través del uso de gramáticas a comportamientos robóticos modelados como máquinas de estado jerárquicas.

El principal objetivo de este trabajo fue desarrollar una metodología de trabajo para poder realizar pruebas con un *fuzzer* que permitiera generar diferentes escenarios de prueba. Como se detalla en la Sección 5.2, este trabajo logró crear una metodología que permite realizar pruebas aleatorias generadas por una gramática para un comportamiento robótico modelado como una máquina de estado jerárquica, siguiendo los pasos de manera secuencial. También se creó una forma de evaluar si la situación encontrada es razonable o no de generar en una ejecución real a través de un cuestionario para conocedores del comportamiento. Si bien la evaluación de esta metodología se diseñó para la librería SMACH en específico, es posible extenderla a otros casos particulares que modelen los comportamientos como máquinas de estados.

De este trabajo se generó un repositorio público en GitHub[1] con el código base utilizado para generar las pruebas. Este código permite generar pruebas para comportamientos escritos en SMACH y si se sigue la metodología propuesta es posible probar otros comportamientos ajustando las gramáticas para cada caso.

[1]https://github.com/rdelgadov/fuzz_testing

Para el equipo UChile Peppers se generó el reporte de cada situación encontrada con una recomendación de parte del autor con una posible causa y solución. Se añadió una automatización al *mock* del robot, una interfaz para emular el robot programáticamente, que el equipo tenía, permitiendo ejecutar acciones del robot sin necesitar un operador que decida qué camino tomar. Por último, se reportaron algunas mejoras para la documentación del repositorio sobre la instalación del ambiente del equipo que se encontraron al configurar en un nuevo computador.

La hipótesis de este trabajo es:

> "*El uso de fuzz testing puede identificar automáticamente errores no triviales en el código de comportamientos robóticos modelados como máquinas de estados jerárquicas*".

Para poder afirmar o negar esta hipótesis se formularon 3 preguntas de investigación, las cuales se respondieron al final de la Sección 6.2, Sección 6.3 y Sección 6.4 respectivamente. Se puede concluir de esta investigación que:

Es posible utilizar técnicas de *fuzz testing* en comportamientos robóticos modelados como máquinas de estados jerárquicas para detectar errores no triviales de manera automática, en particular un *fuzzer* basado en gramáticas del estilo *white-box* junto a un *fuzzer* basado en gramáticas del estilo *black-box* ejecutar y encontrar errores reales en comportamientos robóticos.

Al revisar la bibliografía y los distintos proyectos en las comunidades de robótica es posible ver que si bien utilizar *fuzz testing* en robótica no es una novedad que presente este trabajo, el uso de un *fuzzer* que mezcla técnicas *black-box* y *white-box* con gramáticas enfocado en robótica no se había llevado a cabo antes. Tampoco existía alguna forma específica de probar un comportamiento robótico más allá de las pruebas que cada lenguaje o *framework* permita realizar, por lo que se considera como una contribución a la comunidad de robótica en particular a aquellos que utilizan la librería SMACH.

Este trabajo de tesis generó una publicación aceptada en la conferencia *International Conference on Robotics and Automation* (ICRA) 2021 titulada *Fuzz testing in behavior-based robotics* donde se detalla la metodología de pruebas de comportamientos robóticos desarrollada junto con los resultados de los experimentos.

7.2. Trabajo futuro

Al ser uno de los primeros trabajos en probar comportamientos robóticos de una manera específica, existen múltiples formas de continuar este trabajo, tanto por el lado de mejorar las pruebas como por automatizar la metodología. Algunas ideas para futuros trabajos pueden ser las siguientes:

7.2.1. Otros tipos de fuzzer

El paso más intuitivo si hablamos de *fuzz testing* es probar otros tipos de *fuzzer* para realizar pruebas a comportamientos robóticos. La revisión bibliográfica realizada hace suponer

que existen técnicas de *fuzz testing* que son mejores que otras para ciertos contextos, por ello resulta interesante determinar si un *fuzzer* basado en gramáticas resulta suficientes o existe alguna otra combinación de técnicas de *fuzz testing* que permitan generar mejores pruebas a un comportamiento robótico.

Es posible que el uso de un *fuzzer* basado en mutación para las entradas externas del comportamiento resulte en una mejora a la actual técnica, también supone nuevos desafíos como lo son la validez de la entrada generada y su coherencia en el contexto de uso. Por ejemplo, si pensamos en una entrada externa que recibe una imagen, ¿Qué tanto puede ser modificada esa imagen hasta que deje de ser realista?, ¿Se asemeja a un ruido en la imagen esa mutación o va más allá y cambia por completo a la imagen? o ¿Qué tipo de mutación se requiere hacer para que no pierda valor la entrada?

Se podrían realizar combinaciones de *fuzzer* como los descritos en este trabajo, donde se utilizaron dos *fuzzers* en paralelo para generar entradas para los comportamientos. Una posible mezcla de *fuzzers* podría ser utilizar uno basado en gramáticas con datos de ejemplo y que cada vez que genere un dato con la gramática, se busque en una colección de datos reales para rellenar las estructuras finales de la entrada, pudiendo utilizar datos reales en un contexto de prueba aleatorio.

7.2.2. Gramáticas probabilísticas

Un paso más allá del realizado en este trabajo de tesis es ajustar las probabilidades de la gramática a utilizar para generar datos. Este ajuste puede ser uno empírico y manual en su totalidad o mezclando con estadísticas.

Para el caso empírico, se pueden ir ajustando las probabilidades en cada ejecución, por ejemplo, se pueden utilizar pesos que van aumentando en caso de que alguna ejecución genere un error y se reajusten de manera aleatoria si no. Este ajuste podría ayudar a encontrar errores similares cada vez, pero con datos distintos, pudiendo encontrar toda una colección de entradas que genere errores y sea válida o, incluso, una gramática completa que genere entradas válidas que generen errores.

Otro acercamiento que podría ser relevante para ajustar las gramáticas es utilizar datos reales para generar modelos probabilísticos de cada tipo de entrada y con ello generar datos con estas probabilidades, esto permitiría generar datos válidos y con probabilidades similares a las reales, generando casos más verosímiles. El principal problema de esta idea es encontrar bancos de datos lo suficientemente extensos para generar buena cantidad de contextos y que no se vean parcialmente guiados por unos pocos casos.

7.2.3. Fuzzer generados por aprendizaje de máquinas

Así como el uso de inteligencia artificial (*IA*) y aprendizaje de máquinas (*ML*) es común en robótica se puede aprovechar de igual manera para generar pruebas generadas por *fuzzers*. Un claro ejemplo es EvoGFuzz [36], un *fuzzer* que utiliza gramáticas y técnicas de algoritmos evolutivos para elegir las probabilidades de generación de las entradas. Si bien esta técnica está diseñada para un lenguaje distinto al utilizado en este trabajo, propone una metodología interesante que podría aplicarse a comportamientos robóticos ajustando siempre el contexto

de la aplicación.

Si se requieren pruebas genéricas o la nula intervención de personas especialistas en los comportamientos robóticos a probar, un desarrollo de este tipo en comportamientos robóticos permitiría ejecutar un sin fin de veces cada comportamiento y verificar si es posible a través de algoritmos evolutivos o alguna otra técnica de inteligencia artificial y aprendizaje de máquinas generar casos de prueba interesante, que no solo generen errores de tipos de datos sino más bien errores complejos en el código al ajustar en cada paso las probabilidades de la gramática.

Por otro lado, si lo que se necesita es robustez en la prueba, se puede mezclar la técnica empleada en EvoGFuzz al generar la gramática base con el proceso mostrado en este trabajo de tesis, donde era necesaria una persona entendida en errores del lenguaje utilizado para diferenciar cuando el error era provocado por un mal tipo de dato en contra parte con un error generado por una mala ejecución en el código.

7.2.4. Automatización en la detección de tipos de datos

Este trabajo presenta una metodología que permite conocer el tipo de entradas a través de la prueba y el error con una corrección manual, un trabajo que podría mejorar este trabajo de tesis es la automatización de esa parte. La principal dificultad está en identificar aquellos tipos de datos que podría generar un error que no está directamente relacionado con los errores conocidos de tipo de datos.

Una buena definición de que es un error de tipo de dato podría ayudar a solucionar ese problema y con ello hacer más simple la automatización a través de un proceso de prueba y error. Este paso permitiría generar una gramática lo suficientemente acotada para realizar prueba, pero posiblemente se escape a los detalles más finos del programa y sus entradas como lo podrían ser las características propias de las entradas, por ejemplo, una lista que contenga al menos 10 elementos y sean solo números pares. Llegar a ese nivel de detalle en las pruebas de manera autónoma sería una gran contribución a las pruebas de código en general.

7.2.5. Probar partes de la máquina de estados

Este trabajo mostró como realizar pruebas a estados en particular podrían generar errores, para ello se llevaron a cabo algunos análisis que permitieron concluir que algunos tipos de datos si eran posibles dada las conexiones que la máquina de estados poseía. Realizar pruebas a cadenas de estados con sus transiciones podría reducir el tiempo empleado en las pruebas al realizar el análisis de manera automática sobre la secuencia de estados que previos al estado y poder entender si el tipo de dato entregado es posible o no.

Si bien en el trabajo se habla que las pruebas a un comportamiento completo consumen bastante tiempo, la idea de este posible trabajo futuro es analizar que largos de cadenas de estados son buenos para encontrar errores minimizando el tiempo de ejecución de los estados.

Capítulo 8

Bibliografía

[1] A. Afzal, C. L. Goues, M. Hilton, and C. S. Timperley, "A Study on Challenges of Testing Robotic Systems," *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pp. 96–107, 2020.

[2] A. Afzal, D. S. Katz, C. Le Goues, and C. S. Timperley, "Simulation for Robotics Test Automation: Developer Perspectives," in *International Conference on Software Testing, Validation and Verification*, ICST '21, Apr. 2021.

[3] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, "The Fuzzing Book," in *The Fuzzing Book*, Saarland University, 2019. Retrieved 2019-09-09 16:42:54+02:00.

[4] M. Wahde, "Introduction to autonomous robots," *Department of Applied Mechanics, Chalmers University of Technology, Goteborg, Sweden*, 2012.

[5] G. Qiao and B. A. Weiss, "Accuracy degradation analysis for industrial robot systems," in *International Manufacturing Science and Engineering Conference*, vol. 50749, p. V003T04A006, American Society of Mechanical Engineers, 2017.

[6] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of computer programming*, vol. 8, no. 3, pp. 231–274, 1987.

[7] O. Lebeltel, P. Bessière, J. Diard, and E. Mazer, "Bayesian robot programming," *Autonomous Robots*, vol. 16, no. 1, pp. 49–79, 2004.

[8] H. Lausen, J. Nielsen, M. Nielsen, and P. Lima, "Model and behavior-based robotic goalkeeper," in *Robot Soccer World Cup*, pp. 169–180, Springer, 2003.

[9] J. Bohren, R. B. Rusu, E. G. Jones, E. Marder-Eppstein, C. Pantofaru, M. Wise, L. Mösenlechner, W. Meeussen, and S. Holzer, "Towards autonomous robotic butlers: Lessons learned with the PR2," in *2011 IEEE International Conference on Robotics and Automation*, pp. 5568–5575, IEEE, 2011.

[10] H. Nguyen, M. Ciocarlie, K. Hsiao, and C. C. Kemp, "Ros commander (rosco): Behavior

creation for home robots," in *2013 IEEE International Conference on Robotics and Automation*, pp. 467–474, IEEE, 2013.

[11] J. Bohren and S. Cousins, "The SMACH high-level executive," *Robotics & Automation Magazine, IEEE*, vol. 17, pp. 18 – 20, 01 2011.

[12] P. Schillinger, S. Kohlbrecher, and O. von Stryk, "Human-Robot Collaborative High-Level Control with Application to Rescue Robotics," in *Proc. IEEE Int. Conf. on Robotics and Automation (ICRA)*, pp. 2796–2802, 2016.

[13] D. Riehle, "Composite design patterns," in *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pp. 218–228, 1997.

[14] M. Serrano, E. Gallesio, and F. Loitsch, "Hop: a language for programming the web 2.0.," pp. 975–985, 01 2006.

[15] J. Jackson, "Microsoft robotics studio: A technical introduction," *IEEE robotics & automation magazine*, vol. 14, no. 4, pp. 82–87, 2007.

[16] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source Robot Operating System," in *ICRA workshop on open source software*, vol. 3, p. 5, Kobe, Japan, 2009.

[17] E. Tsardoulias and P. Mitkas, "Robotic frameworks, architectures and middleware comparison," 11 2017.

[18] S. Bøgh, O. S. Nielsen, M. R. Pedersen, V. Krüger, and O. Madsen, "Does your robot have skills?," in *Proceedings of the 43rd international symposium on robotics*, vol. 6, pp. 1–6, Verlag, 2012.

[19] H. Herrero, A. A. Moughlbay, J. L. Outón, D. Sallé, and K. L. de Ipiña, "Skill based robot programming: Assembly, vision and Workspace Monitoring skill interaction," *Neurocomputing*, vol. 255, pp. 61 – 70, 2017. Bioinspired Intelligence for machine learning.

[20] M. R. Pedersen, L. Nalpantidis, R. S. Andersen, C. Schou, S. Bøgh, V. Krüger, and O. Madsen, "Robot skills for manufacturing: From concept to industrial deployment," *Robotics and Computer-Integrated Manufacturing*, vol. 37, pp. 282–291, 2016.

[21] M. Sutton, A. Greene, and P. Amini, "Fuzzing: Brute Force Vulnerability Discovery," 01 2007.

[22] V.-T. Pham, M. Böhme, and A. Roychoudhury, "Model-based whitebox fuzzing for program binaries," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pp. 543–553, 2016.

[23] D. Yang, Y. Zhang, and Q. Liu, "BlendFuzz: A Model-Based Framework for Fuzz Testing Programs with Grammatical Inputs," in *2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*, pp. 1070–1076, 2012.

[24] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: Grammar-aware greybox fuzzing," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 724–735, IEEE, 2019.

[25] R. Hodován, Á. Kiss, and T. Gyimóthy, "Grammarinator: a grammar-based open source fuzzer," in *Proceedings of the 9th ACM SIGSOFT international workshop on automating TEST case design, selection, and evaluation*, pp. 45–48, 2018.

[26] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna, "SNOOZE: Toward a Stateful Network Protocol FuzZEr," in *Proceedings of the 9th International Conference on Information Security*, ISC'06, (Berlin, Heidelberg), p. 343–358, Springer-Verlag, 2006.

[27] Z. Zhang, Q. Wen, and W. Tang, "An Efficient Mutation-Based Fuzz Testing Approach for Detecting Flaws of Network Protocol," in *2012 International Conference on Computer Science and Service System*, pp. 814–817, 2012.

[28] C. De la Higuera, *Grammatical inference: learning automata and grammars*. Cambridge University Press, 2010.

[29] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley, "Scheduling black-box mutational fuzzing," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pp. 511–522, 2013.

[30] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, 2019.

[31] S. Nidhra and J. Dondeti, "Black box and white box testing techniques-a literature review," *International Journal of Embedded Systems and Applications (IJESA)*, vol. 2, no. 2, pp. 29–50, 2012.

[32] R. Fan and Y. Chang, "Machine learning for black-box fuzzing of network protocols," in *International Conference on Information and Communications Security*, pp. 621–632, Springer, 2017.

[33] S. Kim, J. Cho, C. Lee, and T. Shon, "Smart seed selection-based effective black box fuzzing for IIoT protocol," *The Journal of Supercomputing*, pp. 1–15, 2020.

[34] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Transactions on Software Engineering*, vol. 45, no. 5, pp. 489–506, 2017.

[35] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 206–215, 2008.

[36] M. Eberlein, Y. Noller, T. Vogel, and L. Grunske, "Evolutionary Grammar-Based Fuzzing," in *International Symposium on Search Based Software Engineering*, pp. 105–120,

Springer, 2020.

[37] D. Brugali and M. Fayad, "Distributed computing in robotics and automation," *Robotics and Automation, IEEE Transactions on*, vol. 18, pp. 409 – 420, 09 2002.

[38] J. Laval, L. Fabresse, and N. Bouraqadi, "A methodology for testing mobile autonomous robots," in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 1842–1847, IEEE, 2013.

[39] A. Bihlmaier and H. Wörn, "Robot Unit Testing," in *Simulation, Modeling, and Programming for Autonomous Robots* (D. Brugali, J. F. Broenink, T. Kroeger, and B. A. MacDonald, eds.), (Cham), pp. 255–266, Springer International Publishing, 2014.

[40] D. Lee and M. Yannakakis, "Principles and methods of testing finite state machines-a survey," *Proceedings of the IEEE*, vol. 84, no. 8, pp. 1090–1123, 1996.

[41] W. Curran, C. Bowie, and W. D. Smart, "POMDPs for Risk-Aware Autonomy.," in *AAAI Fall Symposia*, 2016.

[42] D. Thomas and A. Hunt, "Mock objects," *IEEE Software*, vol. 19, no. 3, pp. 22–24, 2002.

[43] L. Pitonakova, M. Giuliani, A. Pipe, and A. Winfield, "Feature and performance comparison of the V-REP, Gazebo and ARGoS robot simulators," in *Annual Conference Towards Autonomous Robotic Systems*, pp. 357–368, Springer, 2018.

[44] Y. Zhao, H. Leung, Y. Yang, Y. Zhou, and B. Xu, "Towards an understanding of change types in bug fixing code," *Information and software technology*, vol. 86, pp. 37–53, 2017.