Técnicas basadas en Búsqueda y Aprendizaje para Inferencia de Especificaciones

por Lic. Facundo Molina

Presentado ante la Facultad de Matemática, Astronomía, Física y Computación como parte de los requisitos para la obtención del grado de Doctor en Ciencias de la Computación de la

UNIVERSIDAD NACIONAL DE CÓRDOBA

Febrero, 2022

© FaMAF - UNC 2022

Director: Dr. Nazareno Aguirre

Técnicas basadas en Búsqueda y Aprendizaje para Inferencia de Especificaciones por Facundo Molina se distribuye bajo una Licencia Creative Commons Atribución-NoComercial-CompartirIgual 4.0 Internacional

Técnicas basadas en Búsqueda y Aprendizaje para Inferencia de Especificaciones

por Lic. Facundo Molina

Resumen

La confiabilidad del software es una preocupación central en el proceso de construcción de software y, por lo tanto, una componente fundamental en la definición de la calidad del software. Analizar la confiabilidad del software requiere una *especificación* del comportamiento esperado del software bajo análisis. Desafortunadamente, el software muchas veces carece de tales especificaciones. Este problema afecta negativamente, y de manera sustancial, la capacidad de análisis del software con respecto a su confiabilidad. Por lo tanto, encontrar técnicas novedosas para capturar el comportamiento esperado del software en forma de especificaciones nos permitiría explotar a éstas en tareas de análisis de confiabilidad automático.

En esta tesis presentamos técnicas basadas en búsqueda y aprendizaje para inferir *especificaciones* que permitan distinguir los comportamientos correctos de los comportamientos incorrectos del software. En esta tesis, la referencia en relación a comportamiento correcto e incorrecto será el propio software, es decir, las especificaciones serán producidas a partir del comportamiento *real* del software. El objetivo final es el de utilizar las especificaciones obtenidas para mejorar tareas de análisis de programas y reducir el esfuerzo de los desarrolladores en la construcción de especificaciones. Los enfoques que presentamos, basados principalmente en técnicas de aprendizaje como redes neuronales y computación evolutiva, son capaces de producir especificaciones que pueden capturar más detalladamente comportamientos de software complejo, en comparación con técnicas relacionadas, y también lograr una mejora considerable en una serie de tareas de análisis de programas como bug finding, ejecución simbólica y verificación, entre otras.

Palabras clave: Especificaciones de Software, Inferencia de Especificaciones, Redes Neuronales, Computación Evolutiva, Fuzzing.

A mi familia

Agradecimientos

Durante los cinco años de mi doctorado, recibí la ayuda de muchas personas, y esta tesis no hubiera sido posible sin su apoyo. En este agradecimiento, haré lo mejor posible para expresar mi gratitud hacía cada uno de ustedes.

En primer lugar, me gustaría agradecer a todos mis mentores. En particular, a mi director, el Profesor Nazareno Aguirre. Conocí a Nazareno en los comienzos de mi carrera de grado en la Universidad Nacional de Río Cuarto, más puntualmente durante el 2014, cuando cursé la asignatura Diseño de Algoritmos. Ya hacia el final de mi carrera de grado, Nazareno notó en mi un creciente interés en las ciencias de la computación, lo que lo llevo a proponerme en 2016 la posibilidad de postularme a una beca EVC-CIN, destinada a iniciar a los estudiantes de grado en la investigación científica, y posteriormente a aplicar por una beca doctoral. Desde ese momento, y más principalmente desde 2017, cuando me convertí en estudiante de Nazareno, tuve la oportunidad de aprender innumerables lecciones que me llevarían a convertirme en el investigador que soy hoy. Una de las cosas que más agradecido estoy, es la posibilidad que Nazareno siempre me dió de expresar mis intereses personales de investigación, no solo escuchándome, sino también involucrando estos intereses en los proyectos de investigación que abordamos en nuestro grupo. Esta tesis y mi desempeño como estudiante de doctorado, no hubiera sido posible sin la ayuda de Nazareno, quién ha destinado incontables horas de reuniones conmigo, para analizar mi progreso, responder mis dudas, escribir artículos, cartas de recomendación, y muchas otras tareas. Por estas razones y muchas otras más, me gustaría expresar mi más profundo agradecimiento hacía Nazareno por todas las lecciones que he aprendido a lo largo de estos años, las cuáles espero poder propagar a mis futuros estudiantes.

Continuando con mis mentores, me gustaría agradecer también específicamente a Germán Regis, Renzo Degiovanni y Pablo Ponzio, quienes que han trabajado incansablemente en varios de los proyectos de investigación,

y de quienes he aprendido muchísimo. Germán participó activamente en la implementación de los prototipos que formarían parte de mis primeras publicaciones. Renzo, además de dirigir mi tesis de grado, también realizó aportes considerables en cuanto a la implementación de herramientas y escritura de artículos. Pablo, más hacía el final de mi doctorado, hizo increíbles aportes desde el area de test generation, que facilitaron considerablemente la implementación de los prototipos de inferencia de especificaciones que se presentan en esta tesis.

Me gustaría agradecer también a quienes fueron co-autores de los trabajos de esta tesis: Nazareno Aguirre, Pablo F. Castro, César Cornejo, Marcelo d'Amorim, Renzo Degiovanni, Marcelo F. Frías, Pablo Ponzio, y Germán Regis. De algún modo, cada uno de ellos me ha guiado a través de sus contribuciones, y largas discusiones, en cada trabajo que hemos compartido. Creo que cada una de las contribuciones de esta tesis es también parte del trabajo de ellos, con quienes me encantaría continuar colaborando en el futuro.

Adicionalmente, me gustaría dar las gracias a todos los miembros del Departamento de Computación de la Universidad Nacional de Río Cuarto, del cuál me he sentido parte desde el primer día, y más principalmente a los miembros de nuestro grupo de investigación MFIS (Métodos Formales e Ingeniería de Software). En particular, me gustaría agradecer a Nazareno Aguirre, Marcelo Arroyo, Francisco Babera, Valeria Bengolea, Simón Emmanuel Gutiérrez Brida, Pablo F. Castro, César Cornejo, Renzo Degiovanni, Cecilia Kilmurray, Agustín Nolasco, María Marta Novaira, Sonia Permigiani, Mariano Politano, Luciano Putruele y Gastón Scilingo. Muchas de las ideas presentadas en esta tesis fueron mejoradas a partir de presentaciones y discusiones que se dieron durante reuniones de nuestro grupo de investigación.

Los trabajos de investigación de esta tesis han sido publicados en varias conferencias y/o revistas: ICSE 2019 (Capítulo 3), SBMF 2016 (Capítulo 5), SBST@ICSE 2018 (Capítulo 5), SCP 2019 (Capítulo 5), ICSE 2021 (Capítulo 6) e ICSE 2022 (Capítulo 7). Me gustaría agradecer a todos los revisores anónimos que analizaron los artículos y brindaron comentarios para mejorar estos trabajos. Además, ninguno de estos trabajos hubiera sido posible sin el financiamiento de varias agencias, tales como el Consejo Interuniversitario Nacional (CiN); el Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET); la Agencia Nacional de Promoción de la Investigación, el Desarrollo Tecnológico y la Innovación (Agencia I+D+i) y Microsoft Research. Además, mi doctorado fue apoyado por una beca doctoral de CONICET y un Latin America PhD Award otorgado por Microsoft Research.

Finalmente, me gustaría agradecer a mi familia y amigos. En particular, me gustaría agradecer a mis padres Juan Molina y Andrea Girardi, por su cuidado y apoyo incondicional a lo largo de mi vida. Mis padres dedicaron muchísimo tiempo de su vida a cuidarme, y la razón por la que he dedicado tanto tiempo a mi educación y formación académica, es en gran parte gracias a ellos. También me gustaría agradecer a mi hermano Gonzalo Molina, con quién compartí muchos momentos viviendo juntos en los primeros años de mi doctorado, y me ha ayudado y acompañado. Me gustaría también agradecer a mi novia, Bettina Milanesio. Ella me ha brindado un enorme apoyo en los últimos días de mi doctorado, principalmente durante el tiempo de escritura de esta tesis. Miro con ansias los próximos capítulos de nuestras vidas juntos, y estoy muy agradecido de todo lo que ella hace por mí. Por último, me gustaría agradecer a mis amigos, quienes durante los últimos años me han brindado un apoyo increíble en cuanto a lo social y emocional. En particular, me gustaría agradecer a Agustín Bauer, Rocío Bonino, Aldana Correa, Alan Gonzáles, Joaquín Heredia, Maribel Lopez, Nicolás Pereyra Orcasitas, Luciano Putruele, Cindy Romero, María Jose Sabattini y Ezequiel Zensich.

Índice general

Capítulo 1

Introducción

La calidad de los sistemas de software se define normalmente teniendo en cuenta varias dimensiones, como la confiabilidad, la usabilidad, la eficiencia, etc. Entre éstas, la confiabilidad es considerada generalmente un atributo fundamental de la calidad del software y una preocupación principal durante el proceso de desarrollo de software [22, 38]. El análisis de la confiabilidad del software está fuertemente relacionado con la búsqueda de defectos en el software, es decir, comportamientos reales del software que divergen del comportamiento esperado. Descubrir tales defectos requiere que de algún modo cuál se indique el comportamiento esperado, en otras palabras, que se provea una *especificación* del comportamiento que el software debería exhibir. Por lo tanto, contar con algún tipo de especificaciones que acompañen el software no sólo mejora considerablemente el análisis de confiabilidad, sino que también habilita una serie de aplicaciones, entre las que se incluyen comprensión de programas, evolución y mantenimiento de software, búsqueda de errores [81], mejora de especificaciones [81, 37], entre otras.

Las especificaciones pueden aparecer de diversas maneras. A nivel de código fuente, cuando están presentes, generalmente se manifiestan como *comentarios*, es decir, descripciones informales en lenguaje natural de lo que se supone que debe hacer el software. También es posible encontrar especificaciones más formales a nivel de código fuente, como *aserciones de programas*, es decir, declaraciones (normalmente ejecutables) que capturan propiedades que el software debe satisfacer en ciertos puntos durante su ejecución. Si bien las especificaciones como comentarios son más comunes, tienen la desventaja de que no pueden utilizarse de manera directa para un análisis automático de confiabilidad. En cambio, las aserciones pueden utilizarse para un análisis

de programas de manera sencilla y directa, especialmente cuando se expresan como *contratos* [56], pero rara vez se encuentran disponibles acompañando el código fuente. Además, muchas veces las aserciones de programas establecen propiedades que deben cumplirse en un escenario específico, por ejemplo, declaraciones que solo expresan el comportamiento esperado del software para un caso de test en particular, a diferencia de aserciones más generales, y también significativamente más útiles, asociadas con elementos de contratos como son los invariantes y las pre/poscondiciones.

La situación descrita anteriormente disminuye la posibilidad de explotar las especificaciones para el análisis (automático) de la confiabilidad del software. Además, dado que hoy en día los notables avances en el análisis automático de programas nos permiten producir de manera eficiente grandes conjuntos de inputs para los programas, así como examinar conjuntos muy grandes de ejecuciones de programas, la falta de especificaciones hace más difícil determinar si estas ejecuciones exhiben un comportamiento correcto (esperado), o si son la manifestación de algún defecto del software. Por lo tanto, idear mecanismos novedosos y efectivos para distinguir las ejecuciones de software válidas de las inválidas, es decir, para abordar el llamado problema del oráculo [4], tiene un impacto enorme en la capacidad de las técnicas automáticas de análisis para detectar defectos en los programas.

1.1. El Problema del Oráculo

El Problema del Oráculo [4] se refiere al desafío de distinguir entre el comportamiento correcto (esperado) del software del comportamiento incorrecto (inesperado) del software. Aunque durante varias décadas los investigadores han estado abordando directa o indirectamente el problema, en particular desde la introducción de la noción de corrección de programas en las obras seminales de Hoare [34] y Floyd [14], abordar el problema de manera automática ha recibido una atención significativamente menor en comparación con otras tareas de análisis de programas, como la automatización de generación de tests. Por lo tanto, el problema del oráculo ha sido comparativamente menos tratado [4].

En general, el problema del oráculo se puede abordar utilizando diferentes estrategias [4]. Una alternativa es construir los oráculos a partir de información *implícita* (por ejemplo, problemas de buffer overflow o fallas de segmentación no deberían ocurrir nunca), un aspecto que se aplica a casi

todos los programas. Otra opción es *especificar* manualmente los oráculos a través de algún formalismo, un enfoque que puede permitir caracterizar con precisión el comportamiento esperado pero que requiere un esfuerzo humano significativo. Y finalmente, uno podría intentar *derivar* los oráculos a partir de elementos de software ya existentes como documentación, comentarios, ejecuciones, etc. A continuación describimos con más detalle este último enfoque al problema, que es el que adoptamos en todos los trabajos que forman parte de esta tesis.

1.1.1. Oráculos Derivados

Ante un software que carece de oráculos especificados, una situación muy común, las personas encargadas de evaluar el comportamiento del software pueden recurrir a oráculos derivados. La característica principal de estos oráculos es que intentan capturar la diferencia entre el comportamiento correcto e incorrecto del software basándose en información que es *derivada* de elementos del software existentes. Por ejemplo, si un comentario (entendido como un elemento del software) menciona que el resultado de cierta operación siempre devuelve un valor positivo, un mecanismo de derivación basado en comentarios podría usar esa información para derivar un oráculo que indique que el resultado de dicha operación debe ser siempre mayor que cero.

Entre los diferentes elementos de software que uno puede utilizar para derivar los oráculos, nosotros estamos particularmente interesados en dos de ellos: *oráculos existentes* y *ejecuciones del sistema*. En el primer caso, derivar un oráculo a partir de uno existente podría permitir no sólo mejorarlo (hacerlo más preciso) sino también producir uno equivalente (probablemente en un formalismo diferente) que podría ser más adecuado para una tarea de análisis en particular. En el segundo caso, dado que las ejecuciones reales del sistema son los elementos del software que mejor representan su comportamiento real, probablemente sean los elementos más adecuados para observar al momento de derivar un oráculo que caracterice el comportamiento. En este último caso cabe resaltar que el oráculo caracterizaría el comportamiento real del software, que puede por supuesto no coincidir con el comportamiento esperado (es decir, con la intención de los desarrolladores). Sin embargo, este tipo de oráculos tienen diversas utilidades prácticas, en contextos de análisis diferencial y de regresión, por ejemplo, y como forma de comprensión de programas.

Las técnicas para derivación de oráculos no sólo difieren en el elemento

de software que utilizan, sino también en el tipo de oráculos que producen. Por ejemplo, algunas herramientas de generación de tests (por ejemplo, EvoSuite [15], Randoop [68], JWalk [82]) incorporan mecanismos que producen oráculos para escenarios específicos, es decir, relacionados con secuencias específicas de operaciones. Otras herramientas (por ejemplo, Daikon [13], InvGen [31]) en cambio, generan oráculos que intentan capturar propiedades del software más generales, como invariantes y pre/poscondiciones. Todas las técnicas que presentamos en los siguientes capítulos están específicamente enfocadas en la derivación de este último tipo de oráculos más generales, y utilizan como elementos de software oráculos previamente definidos o ejecuciones del sistema.

1.2. Objetivos

La Figura 1.1 muestra la descripción general de nuestro enfoque al problema del oráculo: el uso de técnicas basadas en búsqueda y aprendizaje para inferir especificaciones (oráculos) a partir *(a)* de especificaciones existentes o *(b)* del comportamiento real del software. Más precisamente, nuestro objetivo es producir especificaciones que puedan usarse directamente para mejorar tareas de análisis de programas como búsqueda de errores [81], ejecución simbólica [44], y testing diferencial o de regresión [2]. En nuestros trabajos, nos concentramos en tres estrategias diferentes:

- el uso de modelos de Aprendizaje Automático [78] para capturar el comportamiento de especificaciones,
- el uso de Computación Evolutiva [40] para inferir especificaciones, y
- el uso de Fuzzing [54], también para inferir especificaciones.

Estas estrategias abordan el problema del oráculo mediante un flujo de trabajo similar. Comienzan desde un elemento específico del software, ya sea una especificación existente o un conjunto de ejecuciones del software, e inician un proceso de inferencia (aprendizaje, búsqueda) que permite generar una nueva especificación, consistente con el elemento en cuestión. Otra similitud, sólo compartida por las técnicas que utilizan ejecuciones del sistema como punto de partida, es el uso de mecanismos del estado del arte en generación automática de tests para producir automáticamente escenarios que ejerciten el software bajo análisis, y obtener así las ejecuciones del mismo.

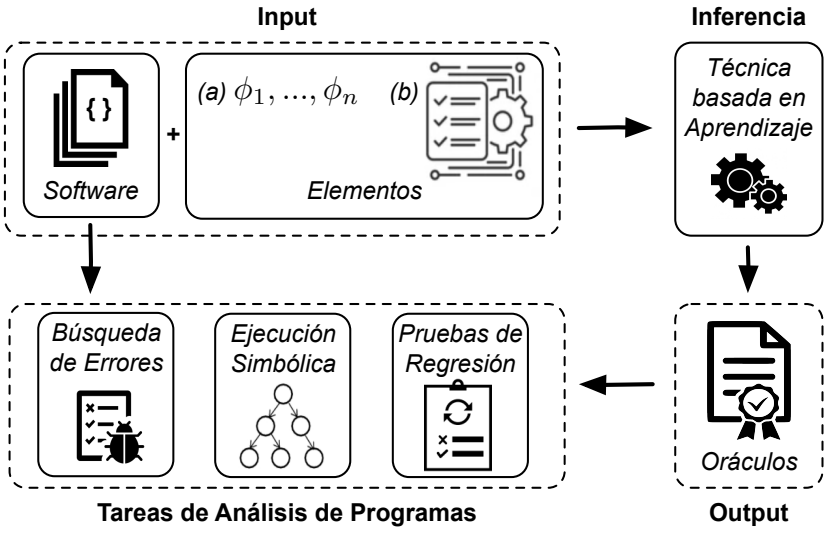


Figura 1.1: Descripción general de nuestro enfoque.

En esta tesis describimos cinco técnicas que implementan las estrategias mencionadas. Dos de éstas están relacionadas con la primera estrategia, utilizando redes neuronales (NNs) [78] como modelo de aprendizaje; otras dos están relacionadas con la segunda estrategia, usando Algoritmos Genéticos (GAs) [40] como mecanismo de computación evolutiva; y finalmente una de estas técnicas está relacionada a la tercera estrategia, basada en el uso de Fuzzing basado en gramáticas [54]. Estas técnicas, además de utilizar distintos elementos de software o distintos mecanismos de aprendizaje subyacentes, también difieren en el tipo de especificaciones que producen. Más precisamente, estas especificaciones difieren en un aspecto muy importante: la *interpretabilidad*. Por un lado, las técnicas que capturan comportamientos de especificaciones utilizando NNs, mantienen la falta de interpretabilidad inherente de tales modelos. Por otro lado, las técnicas basadas en el uso de GAs y de Fuzzing, representan las especificaciones usando un formalismo lógico bien definido, por lo que producen especificaciones que son totalmente interpretables. Como se describe más adelante en cada técnica en particular, la interpretabilidad de las especificaciones tiene implicaciones directas con respecto a las tareas para las que se pueden utilizar.

Especificar formalmente el comportamiento del software a través de orácu-

los puede resultar muy difícil y llevar mucho tiempo, incluso cuando el lenguaje utilizado para especificarlo es el mismo lenguaje de programación utilizado para la implementación (con el cual los desarrolladores están claramente familiarizados). Dado que tener tales oráculos puede tener un gran impacto en nuestra capacidad de detectar defectos del software, así como de aliviar el problema del oráculo, otro de nuestros objetivos de investigación es reducir el esfuerzo de los desarrolladores o ingenieros interesados en especificar el software mediante la provisión de mecanismos para la inferencia de especificaciones ejecutables. Entre las diversas aplicaciones que tienen estas especificaciones ejecutables, incluida la comprensión de programas, la evolución y mantenimiento de software, y la mejora de especificaciones [81, 37], estamos particularmente interesados en su uso en combinación con técnicas de análisis de programas para la búsqueda de errores [81]. Por ejemplo, algunos verificadores de aserciones en tiempo de ejecución, como el que acompaña el conjunto de herramientas de JML [11] o el incluido en la herramienta de generación de tests Randoop [68], pueden beneficiarse de estas especificaciones para mejorar considerablemente la búsqueda de errores.

1.3. Contribuciones

Concretamente, esta tesis realiza las siguientes contribuciones:

- Una técnica basada en NNs para capturar el comportamiento de un tipo particular de oráculos (invariantes de clase) [63], logrando muy buena precisión y permitiendo mejorar tareas de búsqueda de errores.
- Una técnica basada en NNs que mejora la detección de caminos inviables en ejecución simbólica generalizada, proporcionando así una forma de acelerar los tiempos de ejecución simbólica.
- Una técnica que utiliza un GA diseñado para traducir oráculos entre diferentes estilos de especificación [61], habilitando la realización de tareas de análisis de programas que requieren oráculos especificados en un estilo particular.
- EvoSpex [64], un algoritmo evolutivo desarrollado específicamente para inferir poscondiciones de métodos a partir de comportamientos del software, y que es capaz de inferir oráculos que son más precisos que los producidos por técnicas relacionadas.

- SpecFuzzer, una técnica para la inferencia de especificaciones de clase que utiliza una combinación de análisis estático, fuzzing basado en gramáticas, y análisis de mutación, logrando mayor precisión que técnicas existentes.

Cuatro de estas cinco contribuciones ya han sido publicadas en conferencias de Ingeniería de Software, mientras que la restante aún está en evaluación en una conferencia del área. En nuestra investigación adoptamos una política de ciencia abierta, por lo que cada una de nuestras publicaciones tiene asociado un paquete de replicación que está disponible públicamente y se puede utilizar para reproducir todos los experimentos que hemos realizado.

1.4. Esquema

Los siguientes capítulos están organizados de la siguiente manera. El Capítulo 2 proporciona los conceptos preliminares, incluidas las definiciones de los diferentes tipos de oráculos que producen nuestras técnicas, los detalles de los mecanismos de aprendizaje o búsqueda involucrados y también las situaciones específicas en las que se pueden explotar los oráculos para análisis de programas. El Capítulo 3 presenta los detalles y la evaluación experimental de la técnica basada en NNs para capturar el comportamiento de invariantes de clase. El Capítulo 4 describe la técnica basada en NNs enfocada en aprender a podar caminos inviables en ejecución simbólica generalizada, proporcionando también resultados experimentales. El Capítulo 5 presenta los detalles técnicos del GA diseñado para traducir especificaciones a través de diferentes estilos y su evaluación. El Capítulo 6 se centra en los detalles y la evaluación experimental de EvoSpex, el algoritmo evolutivo para la inferencia de poscondiciones de métodos. El Capítulo 7 presenta SpecFuzzer, nuestro enfoque que utiliza fuzzing basado en gramáticas para la inferencia de especificaciones de clase a partir de ejecuciones del software, incluyendo también una evaluación experimental. Finalmente, el Capítulo 8 revisa los principales aportes e implicaciones de cada una de las técnicas presentadas, y concluye esta tesis resaltando la importancia de las principales contribuciones, junto con una descripción de las líneas abiertas para trabajos futuros.

Capítulo 2

Preliminares

Este capítulo proporciona los conceptos preliminares necesarios para la comprensión de las técnicas presentadas en esta tesis. El capítulo está organizado de la siguiente manera. La Sección 2.1 introduce una formalización del problema abordado, que será utilizada a lo largo de la tesis. La Sección 2.2 describe las características principales de las especificaciones de software (que denominaremos indistintamente como *oráculos*) en las que se centran nuestras técnicas. La Sección 2.3 presenta las redes neuronales (NNs), los algoritmos genéticos (GAs) y fuzzing, los tres mecanismos subyacentes a nuestras técnicas de inferencia de oráculos. Finalmente, la Sección 2.4 presenta diferentes escenarios en los que se pueden explotar los oráculos derivados, principalmente para mejorar tareas de análisis de programas.

2.1. Formalización del Problema

Como mencionamos anteriormente, el objetivo de esta tesis es la derivación automática de especificaciones que puedan ser utilizadas como oráculos. En particular, lidiamos con dos instancias del problema de derivación de oráculos: *(i)* inferencia (o síntesis) a partir de especificaciones formales existentes y *(ii)* inferencia (o síntesis) a partir del comportamiento actual del software. Podemos definir las instancias del problema que abordamos, de manera más formal, como lo hacemos a continuación.

Más formalmente, sea $\mathcal{P}$ un programa dado, ρ un punto del programa $\mathcal{P}$ y $\mathcal{S}_\rho$ el conjunto de estados alcanzables en el punto ρ luego de ejecutar $\mathcal{P}$. Las instancias del problema que abordamos pueden definirse de la siguiente

manera:

Definición 1. *Sea $\mathcal{P}$ un programa, ρ un punto del programa $\mathcal{P}$, $\mathcal{S}$ el conjunto de estados posibles de $\mathcal{P}$, y $\mathcal{S}_\rho$ el conjunto de estados alcanzables de $\mathcal{P}$ en el punto ρ, correspondientes a todas las ejecuciones de $\mathcal{P}$. Sea además $\mathcal{L}_1$ un lenguaje y $\phi_\rho^{\mathcal{L}_1}$ una sentencia de $\mathcal{L}_1$ que constituye una* especificación *para el programa $\mathcal{P}$ en el punto ρ, es decir, $\forall s \in \mathcal{S}_\rho : \phi_\rho^{\mathcal{L}_1}[s]$. Dado un lenguaje adicional $\mathcal{L}_2$, el problema de la inferencia de un especificación en $\mathcal{L}_2$ equivalente a la especificación existente $\phi_\rho^{\mathcal{L}_1}$ consiste en construir una sentencia $\phi_\rho^{\mathcal{L}_2}$ de $\mathcal{L}_2$ tal que $\forall s \in S \cdot \phi_\rho^{\mathcal{L}_1}[s] \leftrightarrow \phi_\rho^{\mathcal{L}_2}[s]$.*

Definición 2. *Sea $\mathcal{P}$ un programa, ρ un punto del programa $\mathcal{P}$, y $\mathcal{S}$ el conjunto de estados posibles de $\mathcal{P}$. Sea además $\mathcal{T}$ una test suite que ejercita el programa $\mathcal{P}$, y $\mathcal{S}_\rho^{\mathcal{T}}$ el subconjunto de $\mathcal{S}$ correspondiente a los estados de programa en el punto ρ, observados en las ejecuciones $\mathcal{T}$ de $\mathcal{P}$. Dado un lenguaje $\mathcal{L}$, el problema de la inferencia de una especificación en $\mathcal{L}$ a partir de $\mathcal{T}$ para el punto de programa ρ consiste en encontrar una sentencia $\phi_\rho^{\mathcal{L}}$ de $\mathcal{L}$ tal que $\forall s \in S \cdot s \in \mathcal{S}_\rho^{\mathcal{T}} \leftrightarrow \phi_\rho^{\mathcal{L}}[s]$.*

La Definición 1 captura el problema de derivar una especificación a partir de otra ya existente. Esta tarea, además de permitir mejorar una especificación (por ejemplo, encontrando una equivalente y más compacta), puede utilizarse para producir una especificación equivalente a otra, pero en un formalismo diferente, y así mejorar alguna tarea de análisis. Por ejemplo, como veremos en el Capítulo 5, traducir una especificación escrita en un estilo *imperativo* a otra equivalente escrita en un estilo *declarativo*, permite mejorar considerablemente ciertas tareas de verificación.

La Definición 2 define el problema más general de generar una especificación que captura el comportamiento actual de un programa dado. A diferencia del caso anterior, en este problema no contamos con una especificación previa, sino que buscamos inferirla mediante observaciones de ejecuciones del programa. Este problema es el que más abordaremos en esta tesis, y en el cual se enfocan la mayoría de las técnicas presentadas en los capítulos posteriores.

2.2. Especificaciones de Software

Las especificaciones de software son descripciones abstractas que capturan el comportamiento previsto del software, y sirven para dos propósitos

principales: definir explícitamente las necesidades del usuario y verificar la conformidad de la implementación [22]. Mientras que las especificaciones típicamente aparecen de manera *informal*, a través de documentación o como comentarios en lenguaje natural, son mucho más útiles cuando se las expresa de manera *formal*, mediante restricciones conocidas comúnmente como *contratos* [57, 80]. Particularmente, en esta tesis, estamos interesados en la derivación de dos tipos de especificaciones formales: *invariantes de clase* y *poscondiciones*. Los invariantes de clase capturan el comportamiento esperado de los objetos de una clase mediante aserciones que se deben mantener verdaderas a lo largo de la vida de un objeto, independientemente de los métodos invocados en el mismo, y el orden de las invocaciones. Las poscondiciones, en cambio, están asociadas a un método particular (o más generalmente, una rutina particular), y son aserciones que capturan una propiedad de los estados alcanzados luego de cada ejecución legal del método.

2.2.1. Invariantes de Clase

Una de las claves en el diseño Orientado a Objetos (OO) es el énfasis que este paradigma pone en la *abstracción de datos* [48]. Particularmente, el concepto de *clase* es un mecanismo útil y directo para definir *nuevos* tipos de datos, que amplían el conjunto de tipos predefinidos de un lenguaje de programación. Una clase C define un tipo de datos abstracto y proporciona una implementación para él, definiendo la representación interna del tipo y las operaciones para su manipulación. Por ejemplo, la Figura 2.1 muestra una implementación orientada a objetos, en Java, de secuencias ordenadas de números enteros a través de la clase `SortedList`.

Estas implementaciones de nuevas abstracciones de datos a menudo van acompañadas de una serie de *supuestos* sobre cómo se debe manipular la estructura de datos, los cuáles capturan la intención del desarrollador en la representación elegida. Los *invariantes de clase* o *invariantes de representación* son justamente uno de estos supuestos. Un invariante de representación para una clase C, usualmente llamado `repOk`, es una propiedad que es satisfecha por todos los objetos legítimos de la clase, es decir, objetos que representan objetos abstractos válidos. Más precisamente, es un predicado $inv : C \rightarrow bool$ que es verdadero para cada estado de cada objeto legítimo. Por ejemplo, si consideramos nuestra clase `SortedList`, un invariante de representación para la clase podría verificar que una lista dada es acíclica y que está ordenada de manera ascendente, tal como lo hace el método `repOk` que se muestra en

```java
public class SortedList {
    private int elem;
    private SortedList next;
    private static final int SENTINEL = Integer.MAX_VALUE;

    /** Constructors */
    public SortedList() { this(SENTINEL, null); }
    private SortedList(int elem, SortedList next) {
        this.elem = elem;
        this.next = next;
    }

    /** Inserts the element maintaining the ascending order. */
    public void insert(int data) {
        if (data > elem) {
            next.insert(data);
        } else {
            next = new SortedList(elem, next);
            elem = data;
        }
    }
}
```

Figura 2.1: Implementación de secuencias ordenadas de enteros utilizando listas enlazadas. El campo `elem` mantiene el valor de un nodo de la lista, y el campo `next` la referencia al siguiente nodo. El campo `SENTINEL` almacena un valor especial al final de la lista. El constructor por defecto crea un nodo marcando el final, mientras que el método `insert` inserta el entero `data` en la posición correcta manteniendo el orden.

la Figura 2.2.

En esta tesis, nos enfocamos en el problema de inferir invariantes de clase utilizando dos enfoques diferentes. Por un lado, trabajamos en capturar el comportamiento de los invariantes de clase utilizando modelos de aprendizaje automático a partir de observaciones del comportamiento actual de la clase (Capítulo 3 y Capítulo 4). Por otro lado, investigamos la traducción automática de invariantes de clase escritos en un lenguaje imperativo a otro invariante equivalente pero en un lenguaje declarativo, utilizando computación evolutiva (Capítulo 5).

```
public boolean repOk() {
    Set<SortedList> visited = new HashSet<SortedList>();
    List current = this;
    while (current.elem != SENTINEL) {
        // The list should be acyclic
        if (!visited.add(current))
            return false;
        // The list should be sorted
        if (current.elem > current.next.elem)
            return false;
        current = current.next;
    }
    return true;
}
```

Figura 2.2: Invariante de representación de la clase `SortedList`.

2.2.2. Pre/Poscondiciones

La inclusión de pre y poscondiciones como especificaciones formales está íntimamente relacionada con el uso de aserciones de programas. Si bien el uso de aserciones como especificaciones se remonta a los trabajos de Hoare [33] y Floyd [14], en el contexto de verificación de programas y asociado con el concepto de corrección de programas, pronto se comenzaron a utilizar en lenguajes de programación y metodologías de programación (informales). Más recientemente, han jugado un papel fundamental en la definición de metodologías para el diseño de software, en particular *diseño por contratos* [57]. La gran mayoría de los lenguajes de programación imperativos y orientados a objetos modernos soportan aserciones, ya sea de manera integrada en el lenguaje [58] o a través de bibliotecas maduras como Code Contracts [3] y JML [11].

Técnicamente, una aserción es un predicado sobre estados de un programa, que puede ser usado para capturar *propiedades supuestas*, como las precondiciones, o *propiedades esperadas*, como las poscondiciones. Se dice que un programa $\mathcal{P}$ acompañado de una precondición *pre* y una poscondición *pos* es (parcialmente) correcto con respecto a su especificación, si cada ejecución de $\mathcal{P}$ que comience en un estado que satisface *pre*, si termina, lo hace en un estado que satisface *pos* [33]. Es decir, toda ejecución de $\mathcal{P}$ que termine y sea válida (satisface los requerimientos definidos en la precondición) debe

```
assert(data != SENTINEL); // precondition
insert(data);
assert(repOk() && contains(data)) // postcondition
```

Figura 2.3: Pre y poscondición del método `insert`.

terminar en un estado que satisface la poscondición. A modo de ejemplo, la Figura 2.3 muestra una posible pre y poscondición para el método `insert` de nuestra clase `SortedList`. La precondición requiere que el elemento a insertar no sea el utilizado para representar el fin de la lista, mientras que la poscondición garantiza que la operación de inserción preserva el invariante de representación y además el nuevo elemento fue efectivamente agregado. La incorporación de estos contratos permiten garantizar que el comportamiento del programa sea el esperado.

Con respecto a estos tipos de especificaciones, en esta tesis nos centramos en inferirlas en forma de aserciones. Más precisamente, en el Capítulo 6 describimos una técnica basada en computación evolutiva que, dado un método Java, automáticamente produce una poscondición del comportamiento actual del método. Similarmente, en el Capítulo 7 presentamos una técnica que hace uso de fuzzing basado en gramáticas para inferir especificaciones de clase (incluidas las pre/poscondiciones de los métodos de la clase). En ambos casos, las especificaciones son inferidas en forma de aserciones que pueden ser directamente explotadas para tareas de análisis.

2.3. Técnicas Subyacentes

Esta sección describe las principales técnicas subyacentes a los mecanismos basados en aprendizaje y búsqueda/optimización para la inferencia de especificaciones, que se presentan en los capítulos posteriores.

2.3.1. Redes Neuronales

Las Redes Neuronales (NNs, del inglés *Neural Nets*) son uno de los modelos de aprendizaje automático más tradicionales y exitosos. Hoy en día son de gran importancia entre los profesionales del aprendizaje automático, y forman las bases de muchas aplicaciones comerciales importantes, desde

reconocimiento de objetos en imágenes hasta aplicaciones de procesamiento de lenguaje natural. Por lo general, en cada una de estas tareas se utilizan distintos tipos de NNs. En nuestro caso en particular, estamos interesados en el uso de los modelos conocidos como redes *feed-forward* [27]. La principal razón es que, como explicamos a continuación, el tipo de problema que abordamos en esta tesis puede ser pensado como un problema de *clasificación*, y este tipo de redes son las que mejor se adaptan a tales problemas.

Las tareas de clasificación son una de las más comunes en aprendizaje automático. En este tipo de tarea, el objetivo es que un programa determine a qué categoría pertenece un input dado. Como generalmente las entradas son representadas por vectores n dimensionales y cada categoría con un código numérico [27], resolver esta tarea con algún modelo de aprendizaje implica que el modelo produzca una función $f : R^n \to \{1, \ldots, k\}$. Cuando $y = f(X)$, el modelo asigna la categoría identificada con y al input representado por el vector X de dimensión n. Si pensamos el problema que abordamos en esta tesis, un oráculo puede ser visto como una función que toma un estado de un programa y determina si es correcto o no. Por lo tanto, el problema de derivar automáticamente un oráculo ϕ para un conjunto de estados $\mathcal{S}$ puede ser planteado como una tarea de clasificación: producir una función $f : \mathcal{R}^n \to \{0, 1\}$ que capture el comportamiento de ϕ:

$$\forall s \in \mathcal{S} : \phi[s] \leftrightarrow f(encode(s)) = 1$$

donde $encode : \mathcal{S} \to \mathcal{R}^n$ codifica los estados del programa en vectores. Es decir, para cada vector de $\mathcal{R}^n$ representando un estado de un programa, la función f debe devolver 1 si es un estado aceptado por ϕ, o 0 si es un estado rechazado por ϕ.

Las redes feed-forward [27] son uno de los modelos de aprendizaje por excelencia, principalmente por su notable habilidad para modelar relaciones complejas y no-lineales en los datos, que de otra manera se vuelven muy difíciles de capturar. El objetivo de una red feed-forward es aproximar alguna función f'. Particularmente, para una tarea de clasificación, $y = f'(X)$ mapea una input X a una categoría y. Una red feed-forward que intenta aproximar f' define un mapeo $y = f(X, \theta)$ y *aprende* los valores de los parámetros θ que resultan en la mejor aproximación. Estos parámetros capturan esencialmente coeficientes de la función f.

Una red neuronal puede ser vista como un conjunto de nodos, llamados *neuronas*, que están conectadas entre sí por enlaces dirigidos y ponderados. Cada neurona es una unidad computacional simple que computa una suma

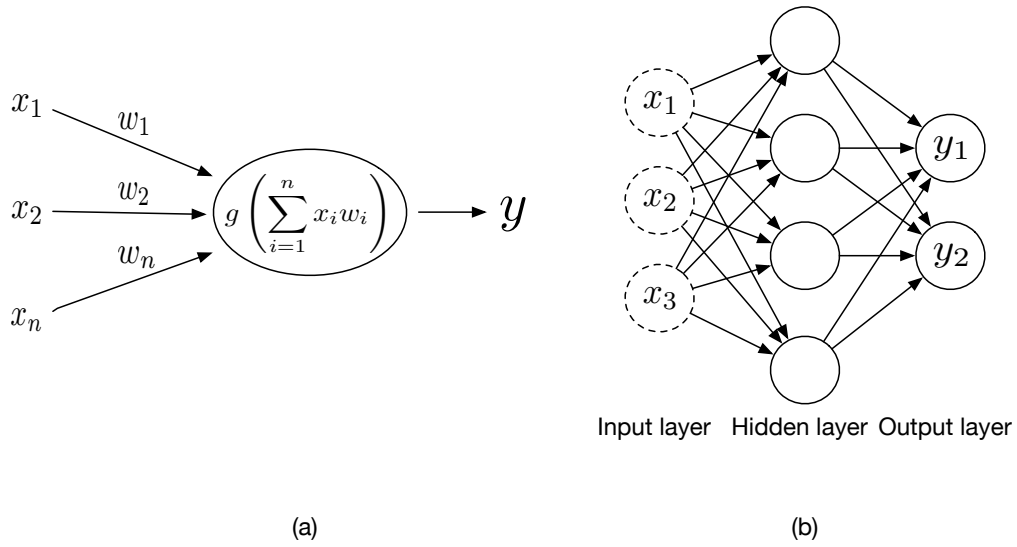


Figura 2.4: Una neurona, y una red feed-forward con una única capa intermedia.

ponderada de sus inputs, y luego aplica alguna función de activación g para producir una salida, como se ilustra en la Figura 2.4(a). Para formar una red, estas neuronas pueden organizarse respetando diversas arquitecturas. En particular, en una red feed-forward, las neuronas típicamente se organizan en capas y la información fluye a través de estas capas en una única dirección, desde la entrada de la red a la salida. En cada capa, una neurona está conectada con cada neurona de la capa siguiente, formando una estructura de grafo acíclico dirigido. La primera capa es la capa de entrada (input layer), y cada una de sus neuronas recibe un componente del vector X de entrada, y simplemente replican el valor recibido a través de sus múltiples salidas, hacia la siguiente capa. La última capa es la capa de salida (output layer) y sus neuronas producen la salida y, obtenida mediante los cómputos intermedios de la red. Además de estas dos capas, puede haber cualquier número de capas intermedias, llamadas capas ocultas (hidden layers). En muchos casos, las redes son utilizadas con una arquitectura de una única capa intermedia, como se ilustra en la Figura 2.4(b), ya que una capa es suficiente para aproximar muchas funciones continuas [78].

El comportamiento de una red, y por lo tanto el de la función f capturada

por ella, puede ser alterado de manera dinámica, mediante la actualización de los parámetros θ (pesos asociados a los enlaces dirigidos de la red), o mediante la modificación de alguno de los llamados *hiperparámetros* de la red: cantidad de capas intermedias, cantidad de neuronas de cada capa, etc. Si estamos aproximando una función f' para la que contamos con un conjunto de inputs y sus respectivas salidas deseadas (es decir, un conjunto de pares input-output conocidos para f'), se puede *entrenar* una NN para que capture una función f que aproxime la función objetivo f'. Esto se puede lograr analizando la diferencia entre la salida esperada y y la salida obtenida de la red para una entrada X dada, y produciendo cambios (leves) en los parámetros θ de modo que, si la red fuera alimentada nuevamente con X, la salida estaría más "cerca" del valor y esperado [78]. Un mecanismo usualmente empleado para esta tarea es *backpropagation* [27].

Como mencionamos previamente, en esta tesis nos enfocamos en mecanismos que usan redes neuronales feed-forward para aproximar el comportamiento de especificaciones de programas (Capítulos 3 y 4), más precisamente invariantes de clase (Subsección 2.2.1).

2.3.2. Algoritmos Genéticos

Los algoritmos genéticos [26] (GAs, del inglés *Genetic Algorithms*) son una forma de búsqueda heurística, que propone resolver problemas de búsqueda (con frecuencia, problemas de optimización) imitando la evolución natural. En general, los GAs requieren que uno provea una codificación del espacio de búsqueda de las soluciones, y mediante un proceso evolutivo, basado fuertemente en elecciones aleatorias guiadas, el algoritmo explora este espacio con el objetivo de encontrar la solución que mejor resuelve el problema. Mientras que la aleatoriedad es una característica notable de los GA, no son una exploración aleatoria simple. Estos algoritmos explotan eficientemente información histórica para especular nuevos puntos de búsqueda con la esperanza de mejorar el desempeño. Más aún, se han demostrado teórica y empíricamente capaces de proveer una búsqueda robusta en espacios complejos [26].

Dado un problema de búsqueda donde la solución es algún elemento de un conjunto $\mathcal{S}$, para abordarlo mediante un GA deberíamos contar con los siguientes componentes:

Codificación. Cada solución candidata $s \in \mathcal{S}$, debería poder representarse como un individuo, comúnmente llamado *cromosoma*, de la población $\mathcal{P}$ del GA, donde $\mathcal{P} \subseteq \mathcal{C}$ y $\mathcal{C}$ es el conjunto de todos los cromosomas posibles. Por

lo tanto, necesitamos una función biyectiva de codificación $encode : \mathcal{S} \to \mathcal{C}$. Usualmente, los cromosomas están formados por *genes* dispuestos en una sucesión lineal, donde cada uno representa una cierta característica del individuo. Luego, cada cromosoma es implementado como un arreglo de n genes, de modo que para cada posible solución s tenemos:

$$encode(s) = c_s = [g_1, g_2, \ldots, g_n]$$

Población inicial. La forma de inicializar la población $\mathcal{P}$ es totalmente dependiente del problema, pero normalmente $\mathcal{P}$ es un conjunto de cromosomas codificando soluciones candidatas, y cuyos genes son seleccionados aleatoriamente. El número de cromosomas iniciales suele ser definido a priori, y mantenido a lo largo de cada generación.

Función de fitness. La función de fitness es quizás el componente más importante del GA ya que tiene un rol central en la exploración del espacio de búsqueda. La función de fitness es una función $f : \mathcal{C} \to \mathbb{R}$, y esencialmente mide la aptitud de un cromosoma c dado, es decir, cuán bien resuelve el problema la solución candidata representada por c. Durante la evolución, el GA tiene como objetivo central optimizar la función f, es decir, descubrir el individuo de la población que maximiza (o minimiza) el valor de f.

Operadores genéticos. Si bien existe un amplio conjunto de operadores genéticos, los GA clásicos utilizan tres operadores: *mutación*, *crossover* y *selección*. Mutación y crossover son dos operadores que permiten obtener nuevos individuos, ayudando así a explorar el espacio de búsqueda, mientras que la selección permite decidir cuáles son los individuos que se preservan en cada generación. La mutación se logra mediante un operador $mut : \mathcal{C} \to \mathcal{C}$, el cual se aplica a un único cromosoma, y produce uno nuevo a través de una alteración aleatoria en el original. El crossover se implementa con otro operador $crossover : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$, el cual toma dos cromosomas $c1$ y $c2$ y produce un nuevo cromosoma combinando la información genética de $c1$ y $c2$. Finalmente, la operación de selección esta dada por un operador $select : P(\mathcal{C}) \to P(\mathcal{C})$ el cual, a partir de una población actual, determina esencialmente cuáles son los cromosomas que formarán parte de la población $\mathcal{P}$ en la próxima generación. Esta última operación hace uso de la función de fitness f y, en su implementación más simple, generalmente ordena a los individuos por su valor de fitness y selecciona los mejores N, donde N es el tamaño máximo de población definido.

Parámetros. El GA recibe una serie de parámetros para su ejecución entre los que se encuentran el tamaño máximo de población max_p, la máxima

Algorithm 1: Un GA clásico.

Input: Tamaño máximo de población max_p, máximo número de generaciones max_g, probabilidad de mutación p_m, y probabilidad de crossover p_c.

Output: El *mejor* individuo encontrado en max_g generaciones.

```
1 Function CLASSIC-GA(max_g, p_m, p_c):
2     gens ← 0;
3     P ← initializePopulation(max_p);
4     while gens ≤ max_g do
5         applyMutation(P, p_m);
6         applyCrossover(P, p_c);
7         P ← applySelection(P);
8         gens + +;
9     return getFittest(P);
```

cantidad de generaciones max_g, la probabilidad de mutación de un individuo p_m, y la probabilidad de aplicar crossover a dos individuos p_c. Generalmente, son necesarios varios experimentos con el algoritmo para poder definir los valores más adecuados de estos parámetros.

A partir de estos elementos, la mecánica de un GA clásico es sorprendentemente simple. La búsqueda comienza a partir de una población inicial $\mathcal{P}$, construida aleatoriamente, y el GA evoluciona esta población de cromosomas de manera iterativa mediante la aplicación de los operadores genéticos *mut* y *crossover*, los cuales permiten producir nuevos individuos mediante alteraciones aleatorias y combinación de otros individuos, respectivamente. Cada iteración representa una generación del proceso de evolución, y en cada una de ellas el operador *select* permite seleccionar los individuos más aptos para ser promovidos a la siguiente generación. De esta manera, el algoritmo va explorando el espacio de búsqueda hasta encontrar el individuo adecuado o cumplir algún criterio de terminación (máxima cantidad de generaciones, timeout, etc.). El Algoritmo 1 ilustra el funcionamiento de un GA clásico.

En las últimas décadas, estos algoritmos, y computación evolutiva en general, han sido ampliamente aplicados en diversas tareas de análisis de programas como generación automática de tests [15], reproducción automática de crashes [83], evaluación y mejora de especificaciones [37], etc. Particularmente, en esta tesis nos enfocamos en el uso de GAs para traducción de

especificaciones entre distintos formalismos (Capítulo 5) y para la inferencia de poscondiciones de métodos Java (Capítulo 6). Para más detalles sobre los algoritmos genéticos y la computación evolutiva en general, referimos al lector interesado a [40].

2.3.3. Fuzzing

Fuzzing es un tópico muy activo tanto en la investigación[1] como en la práctica [29, 30, 28]. Fuzzing es una técnica para producir automáticamente grandes conjuntos de datos (a menudo estructurados), para testear un programa objetivo. El proceso de generación típicamente involucra aleatoriedad y la razón es que testear con (grandes conjuntos de) datos casi válidos puede revelar errores sutiles, tales como inputs mal manejados o casos borde. Un caso de uso muy bien conocido de Fuzzing es la detección de vulnerabilidades de seguridad, como los buffer overflows [60, 54].

Existen diferentes estrategias para fuzzing [54], en particular, el fuzzing basado en gramáticas. Esta técnica utiliza una gramática para producir inputs sintácticamente válidos visitando las reglas de producción de la gramática. En su forma más simple, el proceso de generación de inputs puede ser implementado como expansión incremental de una cadena comenzando del símbolo inicial de la gramática, y reemplazan los símbolos no terminales por la aplicación de una regla de producción de los no terminales correspondientes elegida aleatoriamente, hasta que la cadena sólo contenga símbolos terminales; una cota en el número de no terminales permite a este proceso manejar la recursión, que en otro caso podría llevar a loops infinitos. A modo de ejemplo, consideremos un escenario donde el programa a testear toma como input una fórmula en lógica proposicional (LP), caracterizada por la gramática de la Figura 2.5. Para generar datos para testing, se le puede proveer la gramática LP a un fuzzer basado en gramáticas (por ejemplo, Grammarinator [35]) para obtener eficientemente un conjunto grande de datos de test bien formados (en este caso, fórmulas en LP). Para generar inputs, el fuzzer explora caminos inducidos por las reglas de producción de la gramática. Por ejemplo, el input `neg(p and q)` puede ser obtenido a partir de la siguiente derivación: *start* $\rightsquigarrow$ *formula* $\rightsquigarrow$ `neg` *formula* $\rightsquigarrow$ `neg` (*formula* `and` *formula*) $\rightsquigarrow$ `neg (p` `and` *formula*) $\rightsquigarrow$ `neg (p and q)`. Vale la pena mencionar que una gran ventaja de fuzzing en este caso es que el lenguaje puede ser fácilmente adaptado

[1] https://fuzzing-survey.org

$$
\begin{array}{rcl}
\langle \text{start} \rangle & ::= & \langle \text{formula} \rangle \\
\langle \text{formula} \rangle & ::= & \langle \text{atomic} \rangle \mid \mathit{neg}\ \langle \text{formula} \rangle \mid (\langle \text{formula} \rangle\ \mathit{and}\ \langle \text{formula} \rangle) \\
\langle \text{atomic} \rangle & ::= & \mathit{true} \mid \mathit{false} \mid p \mid q \mid r \mid \ldots
\end{array}
$$

Figura 2.5: Gramática de la Lógica Proposicional.

modificando la gramática. Por ejemplo, el fuzzer podría generar fórmulas con disyunciones si agregamos la regla de producción correspondiente al símbolo no terminal *formula*.

Aunque, como mencionamos previamente, fuzzing es comúnmente utilizado para generación eficiente de inputs para test, en este trabajo lo utilizamos con un objetivo diferente. Más precisamente, en el Capítulo 7, presentamos SpecFuzzer, una técnica para generar especificaciones probables mediante *fuzzing* de potenciales especificaciones asociadas con una clase dada. SpecFuzzer utiliza fuzzing basado en gramáticas para generar automáticamente qué especificaciones puedan ser utilizadas como candidatas por una herramienta de detección de invariantes (como Daikon [13]). La simplicidad con la que la gramática puede ser adaptada o extendida es una de las ventajas del enfoque, en comparación con técnicas relacionadas.

2.4. Especificaciones en Tareas de Análisis

Como mencionamos previamente, contar con especificaciones formales (y ejecutables) del comportamiento del software permite mejorar considerablemente diversas tareas de análisis, proporcionando una mayor confiabilidad. Generalmente, estas especificaciones se utilizan en combinación con técnicas de análisis de programas que realizan alguna tarea específica, como por ejemplo *bug finding* [81], con el objetivo de lograr una mayor efectividad en la tarea. Esta sección describe algunas de las tareas que abordamos a lo largo de estas tesis, utilizando las especificaciones producidas por nuestras técnicas de inferencia.

2.4.1. Bug Finding

La tarea de bug finding se refiere esencialmente a idear mecanismos que permitan detectar, preferentemente de manera automática, defectos en el software. Hoy en día, el objetivo final de la mayoría de las tareas de análisis de programas es tratar de revelar defectos en el software que revelen algún aspecto no deseado. Estos defectos pueden ser funcionales, es decir, aquellos que revelan que el comportamiento exhibido por el software difiere del esperado; de performance, cuando están relacionados a la velocidad de respuesta, la estabilidad, el uso de recursos, etc; de seguridad, cuando el software es vulnerable a ataques que pongan en riesgo la integridad de los datos manipulados por el mismo; entre otros.

Usualmente, la búsqueda de defectos en el software es abordada mediante dos enfoques generales: *análisis estático* y *análisis dinámico*. El análisis estático se enfoca en analizar el software sin ejecutarlo, es decir, mediante la inspección del código fuente; mientras que el análisis dinámico analiza el software a partir de ejecuciones del mismo. Por ejemplo, Infer[2] es una herramienta de análisis estático que analiza el código de los programas con el objetivo de detectar problemas causados por *null dereference* y *memory leaks* mediante una técnica basada en *separation logic* [76] y *bi-abduction* [10]. Por otro lado, herramientas de generación de tests como Randoop[3], además de generar tests que ejercitan un programa dado, permiten incorporar especificaciones para *runtime checking*, lidiando de este modo con la tarea de bug finding desde un enfoque dinámico: aquellos puntos alcanzados durante la ejecución y que no satisfacen con las especificaciones provistas, son considerados defectos.

En esta tesis, abordaremos la tarea de bug finding desde un enfoque puramente dinámico, y enfocamos en la detección de defectos funcionales. Más precisamente, en el Capítulo 3 mostraremos como mejorar tareas de bug finding mediante la provisión de invariantes de clase (capturados por NNs) para runtime checking, tal y como lo permiten herramientas como Randoop.

2.4.2. Ejecución Simbólica Generalizada

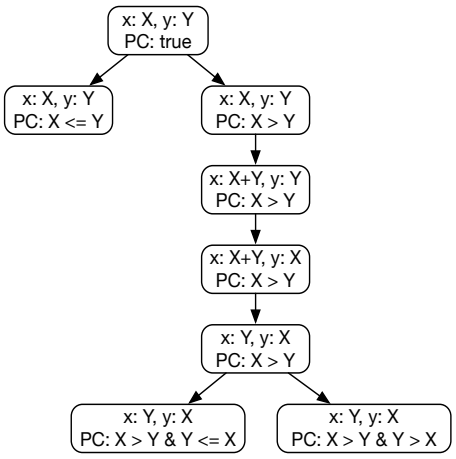
La ejecución simbólica [44] es una tarea de análisis de programas muy popular, que permite explorar sistemáticamente distintas ejecuciones de un

[2]https://fbinfer.com/

[3]https://randoop.github.io/randoop/

```
int x, y;
if (x > y) {
    x = x + y;
    y = x - y;
    x = x - y;
    if (x > y)
        assert(false);
}
```

(a) Programa objetivo.



(b) Árbol de ejecución simbólica.

Figura 2.6: Un programa simple y su árbol de ejecución simbólica.

programa bajo análisis, y aprovechar estas exploraciones para diversas tareas como generación automática de tests [7, 25, 43], verificación de programas [43, 69], estimación de tiempos de ejecución [9, 52], entre otras. En una ejecución concreta, un programa se ejecuta a partir de una entrada específica, y por lo tanto explora un único camino del grafo de control de flujo del programa. En cambio, en una ejecución simbólica, se puede explorar simultáneamente múltiples caminos que un programa podría tomar bajo diferentes entradas. Esto se logra mediante un motor de ejecución simbólica, el cuál ejecuta el programa utilizando inputs *simbólicos*, en lugar de concretos, y manteniendo, para cada camino, la siguiente información: (i) una fórmula booleana de primer orden que captura las condiciones satisfechas por las ramas tomadas a lo largo de ese camino, y (ii) una *memoria simbólica* que asigna variables a expresiones o valores simbólicos. La ejecución de una rama actualiza la fórmula, mientras que las asignaciones actualizan la memoria simbólica.

Consideremos el fragmento de código que se muestra en la Figura 2.6a, el cuál intercambia el valor de las variables `x` e `y` cuando `x` es mayor que `y`. La Figura 2.6b también muestra el árbol de ejecución simbólica correspondiente. Al ejecutar simbólicamente este programa con un motor de ejecución simbólica, inicialmente `PC = true` (la condición de camino) y `x` e `y` tienen los valores simbólicos `X` e `Y`, respectivamente. En cada punto, la `PC` es actualizada con

suposiciones sobre los inputs, para poder elegir entre caminos alternativos. Por ejemplo, luego de la ejecución de la primer sentencia, ambas alternativas `then` y `else` de la sentencia `if` son posibles, por lo que la `PC` es actualizada de la manera correspondiente. Si la condición de camino se vuelve `false`, es decir, no existe un conjunto de entradas que la satisfaga, significa que el estado simbólico no es alcanzable, y la ejecución simbólica descarta ese camino. Por ejemplo, la sentencia `assert(false)` no es alcanzable.

Cuando los programas manipulan variables de tipos de datos básicos o tipos estructurados simples, la resolución de las condiciones de caminos durante ejecución simbólica puede ser realizarse recurriendo a tecnologías estándares de constraint solving como SMT [65] solving. Pero cuando los programas involucran datos alocados en el heap, los SMT solvers no pueden manejar de manera sencilla restricciones sobre estos tipos de datos, y por lo tanto, es necesario algún enfoque complementario. La ejecución simbólica generalizada, introducida en [43], propone abordar este problema mediante una doble generalización de la ejecución simbólica tradicional. Primero, se propone una traducción de código a código para instrumentar un programa, de modo que se pueda hacer ejecución simbólica del programa utilizando un model checker estándar sin tener que construir una herramienta específica. Segundo, se presenta un algoritmo de ejecución simbólica novedoso que manipula datos alojados en el heap, precondiciones de métodos, datos y concurrencia. Debajo describimos el funcionamiento de este algoritmo, conocido como *lazy initialization* [43].

Lazy Initialization

El algoritmo de lazy initialization (Algoritmo 2) propone iniciar la ejecución a partir de un estado en el que cada variable basada en referencia es un "objeto" simbólico, el cuál será parcialmente concretizado cada vez que sus campos son accedidos. La concretización parcial tendrá un número de caminos alternativos, que deberán ser explorados exhaustivamente. Un objeto simbólico puede ser concretizado de tres maneras posibles: *(i)* como `null`; *(ii)* como una referencia a un objeto previamente observado y del mismo tipo, es decir, un objeto compatible que ya está presente en el heap parcialmente simbólico (notar que esta alternativa en realidad corresponde a tantos casos como objetos compatibles existan en el heap); o *(iii)* como una referencia a un objeto *nuevo* (no observado previamente), cuyos campos son todos simbólicos. Notar que en este contexto generalizado, la ejecución

Algorithm 2: Inicialización de campos en Lazy Initialization

Input: Un campo f a ser inicializado durante la ejecución simbólica.

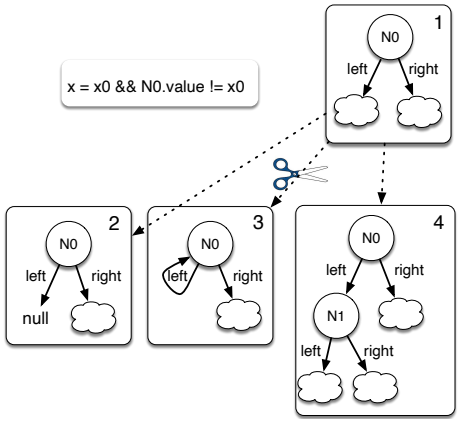
```
1  Function LAZY-INITIALIZATION(f):
2      if f is uninitialized then
3          if f is reference field of type T then
4              nondeterministically do:;
5              1. o.f = null;
6              2. o.f = newSymbolicObjectOfType(T);
7              3. o.f = getExistingObjectOfType(T);
8              if method precondition is violated then
9                  backtrack();
10         if f is primitive (or string) field then
11             o.f = newSymbolicValueForField(f)
```

simbólica mantendrá tanto la condición de camino como un heap parcialmente simbólico, y todavía es necesario determinar la viabilidad de los caminos, teniendo en cuenta la condición de camino en conjunto con el heap parcialmente simbólico, sobre el cuál habrá normalmente alguna precondición asumida (por ejemplo, un invariante de representación asumido sobre una estructura de datos alojada en el heap).

A modo de ejemplo, consideremos una implementación simple de árboles binarios de enteros, y un método de búsqueda, que dado un nodo del árbol y un valor a buscar, devuelve `true` si y solo si el valor pertenece al sub-árbol cuya raíz es el nodo dado. El método se muestra en la Figura 2.7a, y presenta una precondición, la cuál asume que el nodo es la raíz de un árbol binario (es decir, una estructura acíclica, con cada nodo alcanzable, excepto la raíz, teniendo exactamente un nodo padre). La estructura (1) de la Figura 2.7b muestra un heap parcialmente simbólico, junto con su condición de camino, que correspondería a evaluar la sentencia `if` en la primera línea del método `search`, en la rama `else`. Ahora, cuando la condición de la segunda sentencia `if` es simbólicamente evaluada, tres caminos son generados, de acuerdo a lazy initialization: uno para el caso en que `this.left` es `null` (estructura (2)); uno en el que `this.left` apunta al único nodo previamente creado (estructura (3)); y uno en el que `this.left` apunta a un nuevo nodo, con todos sus campos simbólicos (estructura (4); el campo `value` no se muestra

```
public boolean search(int x) {
  if (this.value == x) return true;
  if (this.left != null) {
    boolean inL =
        this.left.search(x);
    if (inL) return true;
    if (this.right != null)
      return this.right.search(x);
  }
  return false;
}
```

(a) Método `search`.



(b) Árbol de ejecución simbólica.

Figura 2.7: Un método de `BinaryTree`, y su ejecución simbólica utilizando lazy initialization.

por simplicidad). Notar como el segundo camino debería ser podado, dado que no satisface la precondición, indicando que `this` debería representar un árbol válido (más detalles de la implementación de esta precondición se presentan más abajo).

Del mismo modo que debe definirse un límite a la profundidad para la exploración exhaustiva de caminos, el tamaño del heap también debe limitarse. Una forma habitual es utilizar, por ejemplo, un número máximo de objetos, de modo que cuando se alcance ese número, las alternativas de lazy initialization solo serán los dos primeros casos descritos previamente (es decir, instanciar un objeto simbólico con `null` o con un objeto previamente observado). Algunas veces no referiremos a este límite como el *scope* del análisis. Para más detalles en relación a ejecución simbólica generalizada y al algoritmo de lazy initialization, remitimos al lector a [43].

El análisis de programas mediante ejecución simbólica brinda una ventaja significativa en comparación con las ejecuciones concretas debido esencialmente a dos razones principales: el colapso de múltiples ejecuciones concretas en una única ejecución simbólica lleva a una explosión de ejecuciones significativamente más pequeña; y la satisfacibilidad de las condiciones recolectadas de los caminos puede ser verificada utilizando *constraint solvers*, normalmente SAT o SMT solvers [65], de modo que caminos no satisfacibles pueden ser

podados en etapas tempranas del análisis simbólico. Como veremos en el Capítulo 4, en esta tesis nos enfocamos en mejorar la detección de caminos inviables durante ejecución simbólica generalizada [43] mediante el uso de NNs.

Capítulo 3

Aproximando Invariantes de Clase con Redes Neuronales

Este capítulo presenta la primera técnica basada en aprendizaje: el uso de NNs para capturar el comportamiento de invariantes de clase (Subsección 2.2.1). La idea general de la técnica es observar el comportamiento exhibido por la implementación actual de una clase y, a partir de estas observaciones, entrenar una NN de modo que aproxime el invariante de representación de la clase actual, es decir, que aprenda a distinguir los objetos válidos de la clase de los objetos inválidos. El objetivo es entonces que los clasificadores obtenidos puedan luego ser utilizados (como invariantes de clase) al momento de realizar alguna tarea de análisis de programas, como bug finding. El resto del capítulo está organizado de la siguiente manera. La Sección 3.1 introduce la técnica a través de un ejemplo ilustrativo. La Sección 3.2 describe cada uno de los pasos de nuestro mecanismo, incluyendo como son construidas y entrenadas las NNs. La Sección 3.3 muestra los resultados experimentales obtenidos en relación a la efectividad de las NNs aproximando invariantes de clase y en el uso de estas redes en tareas de bug finding. Finalmente, la Sección 3.4 describe las conclusiones de este trabajo y algunas líneas de investigación futuras.

3.1. Un Ejemplo Ilustrativo

Consideremos nuevamente la implementación de listas ordenadas de enteros a través de la clase `SortedList`, ilustrada en la Figura 3.1. En un

```java
public class SortedList {
    private int elem;
    private SortedList next;
    private static final int SENTINEL = Integer.MAX_VALUE;

    /** Constructors */
    public SortedList() { this(SENTINEL, null); }
    private SortedList(int elem, SortedList next) {
        this.elem = elem;
        this.next = next;
    }

    /** Inserts the element maintaining the ascending order. */
    public void insert(int data) {
        if (data > elem) {
            next.insert(data);
        } else {
            next = new SortedList(elem, next);
            elem = data;
        }
    }
}
```

Figura 3.1: Implementación de secuencias ordenadas de enteros utilizando listas enlazadas. El campo `elem` mantiene el valor de un nodo de la lista, y el campo `next` la referencia al siguiente nodo. El campo `SENTINEL` almacena un valor especial al final de la lista. El constructor por defecto crea un nodo marcando el final, mientras que el método `insert` inserta el entero `data` en la posición correcta manteniendo el orden.

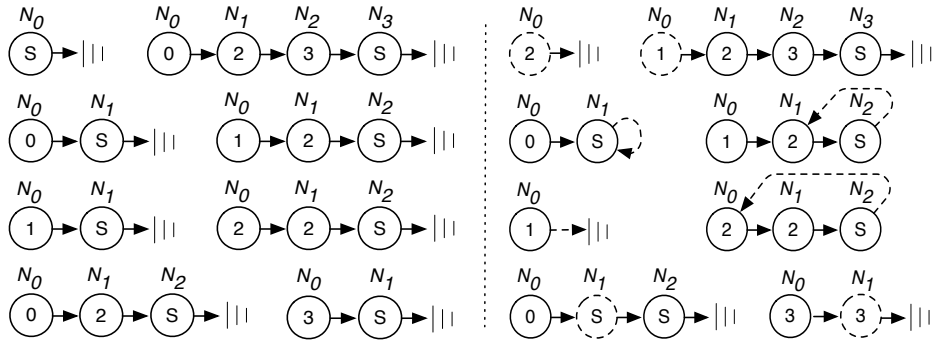
escenario típico de análisis, podríamos estar interesados en verificar que esta implementación se comporta como se espera. Esto incluye garantizar que todos los constructores públicos de la clase permiten construir objetos que satisfacen el invariante de clase, y que los métodos públicos de la clase (incluyendo métodos para inserción, eliminación y recuperación de elementos) preservan este invariante. Es decir, considerando el invariante ilustrado en la Figura 3.2), todas las operaciones deberían mantener la aciclicidad y el orden ascendente de la lista. Si tuviéramos este invariante especificado formalmente, podríamos verificar que efectivamente se preserva con la ayuda de algunas herramientas de análisis, por ejemplo, el verificador de aserciones

```
public boolean repOk() {
    Set<SortedList> visited = new HashSet<SortedList>();
    List current = this;
    while (current.elem != SENTINEL) {
        // The list should be acyclic
        if (!visited.add(current))
            return false;
        // The list should be sorted
        if (current.elem > current.next.elem)
            return false;
        current = current.next;
    }
    return true;
}
```

Figura 3.2: Invariante de representación de la clase `SortedList`.

para *runtime checking* como el incluido en JML [11], o una herramienta de generación de tests como Randoop [68]. El problema es que obtener estos invariantes correctamente, y especificarlos en algún lenguaje apropiado, incluso utilizando el mismo lenguaje que el utilizado para la implementación, es una tarea muy difícil y que demanda mucho tiempo; por lo que por lo general estos invariantes no se encuentran acompañando las implementaciones.

Ante esta situación, nuestra idea es, dada una clase C, aproximar el invariante de clase $inv : C \rightarrow bool$ mediante una NN nn_c. Para lograr esto, necesitamos entrenar la NN con un conjunto de objetos de la clase C para los cuáles sepamos la salida correcta. En otras palabras, necesitamos entrenar nn_c con un conjunto de instancias válidas de la clase C, es decir, objetos que satisfacen el invariante (o, equivalentemente, para los que el invariante debería retornar `true`), así como un conjunto de instancias inválidas, es decir, objetos que no satisfacen el invariante (para los que el invariante debería retornar `false`). Para producir estos conjuntos, nuestra técnica require que el usuario provea un subconjunto de los métodos de la clase, que serán utilizados como constructores y cuya implementación se asume correcta, es decir, son métodos que nos permiten construir instancias válidas. Por ejemplo, para la clase `SortedList` el usuario puede confiar en la implementación del constructor y del método `insert`, y por lo tanto, se asume que todos los objetos producidos con tales métodos son correctos. Con estos métodos podemos construir instancias que se asumen correctas, utilizando, por ejem-



(a) Instancias válidas. (b) Instancias inválidas.

Figura 3.3: Instancias de la clase `SortedList`. Cada objeto N_i representa un nodo de la lista, y contiene su elemento (campo `elem`) y una referencia al siguiente (campo `next`). El elemento S representa el valor de `SENTINEL`.

plo, una herramienta de generación de tests como Randoop. Un conjunto particular de instancias válidas que podemos obtener mediante este proceso podría ser el que se muestra en la Figura 3.3a.

Crear instancias inválidas es más difícil. Podemos pedirle al usuario que manualmente provea tales casos, pero el número de objetos necesarios para poder entrenar la red nn_c de manera apropiada sería demasiado grande, lo que sería totalmente impráctico. También podemos pedirle al usuario que proporcione métodos para construir objetos incorrectos, pero esto significaría trabajo extra (no es algo que el usuario disponga de antemano), y proveer tales métodos no es, en principio, una tarea sencilla. En cambio, nuestra técnica se basa en el uso de *extensiones de campos*[1], las cuáles se pueden computar a partir del conjunto de instancias válidas. Al computar estas extensiones de campos a partir de instancias válidas, lo que capturamos es el conjunto de valores admisibles para los campos, es decir, al menos un objeto observado admite cada valor de la extensión correspondiente. Por ejemplo, las extensiones para los campos `elem` y `next` de nuestro ejemplo, considerando las

[1]Estas extensiones, computadas a partir de un conjunto de objetos de una clase, capturan esencialmente, para cada campo de la clase, todos los valores que han sido observados en ese campo en al menos un objeto del conjunto.

instancias válidas de la Figura 3.3a, son las siguientes:

$$\begin{aligned}
\texttt{elem} &= \{\langle N_0, S\rangle, \langle N_0, 0\rangle, \langle N_0, 1\rangle, \langle N_0, 2\rangle, \langle N_0, 3\rangle, \langle N_1, S\rangle, \langle N_1, 2\rangle, \\
&\quad \langle N_2, S\rangle, \langle N_3, S\rangle, \langle N_3, S\rangle\} \\
\texttt{next} &= \{\langle N_0, null\rangle, \langle N_0, N_1\rangle, \langle N_1, null\rangle, \langle N_1, N_2\rangle, \langle N_2, null\rangle, \\
&\quad \langle N_2, N_3\rangle, \langle N_3, null\rangle\}
\end{aligned}$$

Entonces, la idea es construir instancias *potencialmente* inválidas cambiando valores de los campos en instancias válidas. Al cambiar el valor de un campo en un objeto dado, tenemos dos posibilidades: lo podemos cambiar por un valor "por fuera" de las extensiones, es decir, uno que no ha sido observado en ninguna de las instancias válidas; o por un valor "por dentro" de las extensiones, es decir, uno que es diferente al valor original pero dentro de los observados para el campo en cuestión. En el primer caso, necesitamos definir un *scope*, de modo que la noción de estar "por fuera" de las extensiones pueda definirse precisamente. Asumamos, por simplicidad, que definimos arbitrariamente el siguiente *scope*: exactamente 4 nodos (N_0, N_1, N_2, N_3) para el campo `next`, y `elem` en el rango S,0,...,3. Ahora, podemos construir instancias (supuestamente) inválidas cambiando, para cada instancia válida, el valor de algún campo por otro diferente, por fuera de la extensión correspondiente, o por dentro pero diferente del valor original. En la Figura 3.3b mostramos algunas instancias potencialmente inválidas obtenidas usando este mecanismo a partir de las válidas, representando la modificación con una línea punteada. Por ejemplo, la instancia inválida superior izquierda, cambia el valor S en el campo `elem` del nodo N_0 por el valor 2 que está dentro de las extensiones. Similarmente, la instancia que se encuentra debajo cambia el valor *null* en el campo `next` del nodo N_1 por el valor N_1, el cuál está fuera de las extensiones.

A partir de estos ejemplos, podemos observar algunos problemas. Primero, no hay garantía de que este proceso nos permita producir instancias verdaderamente inválidas. De hecho, la instancia de arriba a la derecha en la Figura 3.3b es en realidad válida. Sin embargo, las NNs son bastante tolerantes al "ruido" durante el aprendizaje, por lo que siempre que el número de objetos inválidos mal categorizados sea bajo, el aprendizaje todavía puede funcionar relativamente bien. Segundo, lo que podemos construir como instancias potencialmente inválidas depende en gran medida de lo que producimos como válidas y de lo que definimos como *scope*. Ambos problemas son críticos, y son abordados con más detalle en la próxima sección. Tercero,

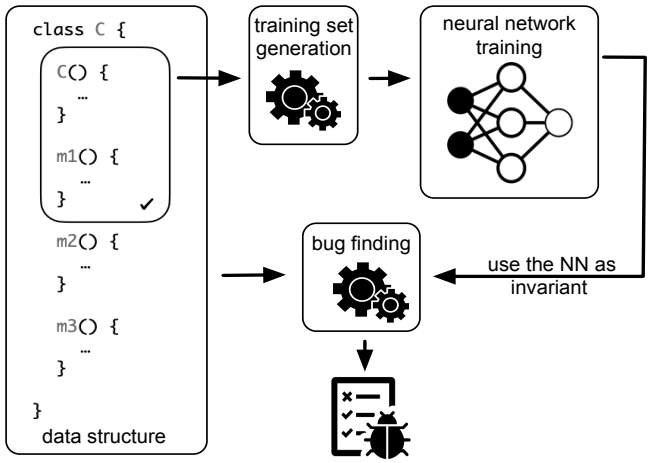


Figura 3.4: Una vista general de nuestro enfoque.

tampoco especificamos el mecanismo para elegir un valor dentro de la extensión correspondiente o fuera de ella. Uno podría elegir aleatoriamente en que dirección ir, y con que proporción caer dentro o fuera de las extensiones. Intuitivamente, salir "fuera" de las extensiones tiene mejores chances de producir instancias inválidas.

3.2. Aproximando Invariantes de Clase

Presentemos ahora en más detalle nuestra técnica basada en NNs para la aproximación de invariantes de clase. La técnica, ilustrada en la Figura 3.4 tiene tres pasos principales: *(i)* la generación automática de instancias válidas e inválidas que formaran el conjunto de entrenamiento, *(ii)* la representación de esas instancias como vectores numéricos (para poder proveerlos a una NN), y *(iii)* la construcción y entrenamiento de una NN para aprender a clasificar instancias de clases como válidas e inválidas, aproximando así el comportamiento del invariante.

3.2.1. Generación de los Datos de Entrenamiento

Dada una clase C para la cuál queremos aproximar su invariante, en otras palabras, aprender a clasificar instancias válidas vs. inválidas, nuestro primer paso consiste en obtener los datos de entrenamiento. Para ello, generamos un conjunto de instancias $\mathcal{V}$ y un conjunto de instancias $\mathcal{I}$ que deben y no deben satisfacer el invariante *pretendido*, respectivamente. Como mencionamos anteriormente, nuestra primera suposición es que contamos con un conjunto $\mathcal{M}_c$ de métodos identificados cómo los *constructores* de la clase, es decir, un conjunto de métodos cuyas implementaciones se asumen correctas, y por lo tanto pueden ser utilizados para construir instancias válidas. De hecho, esta suposición es bastante común en tareas de verificación y generación de tests en las que es necesario construir instancias a partir de la interfaz pública de una clase (ver, por ejemplo, [43, 66, 55]). Nuestra segunda suposición es que además se provee una noción de *scope* (ver la sección previa para intuición de este requerimiento). El scope provee un dominio definido para cada campo de cada clase involucrada en el análisis y, por lo tanto, proporciona un dominio donde buscar valores de campos al momento de construir instancias potencialmente inválidas a partir de las válidas. El scope no solo es relevante para definir un universo finito para calcular el complemento de las extensiones de campo; sino que también acota las instancias a ser consideradas, lo que nos permite caracterizarlas como vectores de tamaño fijo (ver Subsección 3.2.2).

Generación de Instancias Válidas

Para construir el conjunto $\mathcal{V}$ de instancias válidas, cualquier técnica de generación de inputs que pueda producir objetos a partir de la API de la clase es adecuada, incluyendo, por ejemplo, las basadas en model checking [43, 24] y en generación aleatoria [68, 59]. En nuestro caso particular, lo construimos utilizando Randoop, una herramienta de generación aleatoria muy popular. Más precisamente, durante la generación limitamos Randoop para que genere secuencias de métodos que solo utilicen métodos del conjunto $\mathcal{M}_c$, es decir, métodos que han sido identificados como constructores. Luego, el conjunto $\mathcal{V}$ es construido colectando las instancias obtenidas luego de ejecutar cada una de las secuencias de test producidas por Randoop.

Generación de Instancias Inválidas

El conjunto de instancias inválidas $\mathcal{I}$ es generado a partir del conjunto de instancias válidas. Para ello, computamos las extensiones de campos a partir de $\mathcal{V}$ y generamos el conjunto $\mathcal{I}$ de instancias potencialmente inválidas de la siguiente manera: dada una instancia válida $v \in \mathcal{V}$, un objeto o alcanzable desde v y un campo f en o, cambiamos el valor de $o.f$ ya sea a un valor dentro de la extensión de f, o fuera de ella pero dentro del scope. En nuestra implementación, favorecemos el segundo caso dado que aprovecha las extensiones de los campos garantizando que un nuevo objeto es construido. A partir de cada instancia válida, producimos tantos objetos inválidos como campos alcanzables, utilizando el procedimiento anterior para cambiar un solo campo del objeto en cada caso. La elección de ir por dentro o fuera de las extensiones puede hacerse de manera aleatoria. En nuestros experimentos, cambiamos un campo de un objeto dentro de las extensiones con una probabilidad de 0.2, y fuera de las extensiones con una probabilidad de 0.8. La justificación de esta elección se basa en probar experimentalmente, para scopes pequeños, diferentes probabilidades (y las elegidas fueron las que exhibieron mejor desempeño). Finalmente, descartamos cualquier objeto potencialmente inválido que aparezca dentro del conjunto de objetos válidos, de modo que $\mathcal{V} \cap \mathcal{I} = \emptyset$.

Es importante remarcar que nuestra técnica de generación de instancias válidas está relacionada con cómo se elige el scope. Algunas técnicas de generación requieren el scope *a priori* (por ejemplo, [24]), mientras que en otras el scope puede ser derivado a partir de las instancias generadas, por ejemplo, mirando el objeto producido más grande, o el rango de valores producidos. Intuitivamente, el scope debe ser como mucho "ligeramente suelto" con respecto a las extensiones de campo correspondientes a las instancias válidas generadas, en el sentido que, al construir objetos inválidos, debe prevalecer la formación de asociaciones de campo que no formen parte de cualquier objeto válido, pero que involucran valores que son parte de objetos válidos. En nuestros experimentos, elegimos el scope a priori, y descartamos cualquier objeto generado aleatoriamente que vaya fuera del scope.

3.2.2. Representación Vectorial

Las redes neuronales reciben como input vectores de valores, que en general están restringidos a valores de tipo *numérico*. Mientras que para algunos tipos de datos la codificación es directa (por ejemplo, caracteres, tipos enu-

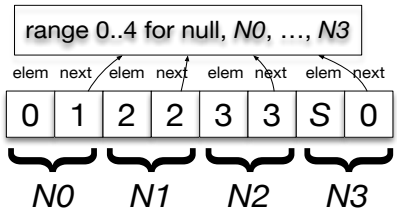


Figura 3.5: Codificación vectorial para una instancia de `SortedList`.

merados, tuplas de tipos básicos, y strings con una longitud máxima), para instancias de una clase arbitraria C es mucho menos sencillo. En este trabajo, para codificar instancias como vectores, adoptamos el formato de vectores candidatos propuesto en Korat [6]. Dado un scope k, el cuál define rangos para tipos numéricos y un número de instancias máximo para referencias, cualquier instancia o de la clase C dentro del scope k puede ser representada mediante un vector conteniendo una celda por cada campo f de cada objeto o' alcanzable a partir de o. El dominio de cada celda es un rango de números naturales, donde cada valor identifica unívocamente un objeto/valor dentro del scope, para el campo correspondiente. Por ejemplo, siguiendo la clase `SortedList`, dado un scope de 4 nodos y elementos en el rango S, 0, ..., 3, la lista de más arriba a la derecha de la Figura 3.3a es representada por el vector candidato de la Figura 3.5.

Esta representación implica que el tamaño máximo considerado para los tipos de datos basados en referencia debe establecerse de antemano. Por lo tanto, en principio, no podemos utilizar una red entrenada para instancias de un tamaño máximo k con instancias de mayor tamaño, ya que estas últimas serían capturadas con vectores diferentes (de mayor tamaño). Además, es importante remarcar que nuestro mecanismo para codificación de instancias es determinista. Las instancias se representan canónicamente asignando identificadores a cada nodo a partir de su orden en *breadth-first* (utilizando orden de campos arbitrario, pero fijo). Además, los campos con valores de punto flotante son compatibles con nuestra técnica, pero se ignoran al momento de generar vectores no válidos (es decir, nunca construimos instancias "inválidas" modificando estos campos).

3.2.3. Arquitectura y Entrenamiento de la Red

Dada una clase C para la cuál hemos construido los conjuntos $\mathcal{V}$ e $\mathcal{I}$, conteniendo las instancias válidas e inválidas, respectivamente, el conjunto de vectores representando cada una de estas instancias forman el *conjunto de entrenamiento* que utilizamos para entrenar una NN. La red que construimos con el objetivo de aprender a distinguir entre instancias válidas e inválidas es una red *feed-forward* (ver Subsección 2.3.1). En primer lugar, asumiendo que el tamaño de los vectores actuales es n, la capa de entrada contendrá n neuronas, cada una recibiendo el valor de una posición del vector, y la capa de salida siempre tendrá 1 neurona ya que nuestro problema de clasificación involucra solo dos clases diferentes. Además, utilizamos solo una capa intermedia. El número de neuronas de esta capa intermedia es un hiperparámetro cuyo valor puede impactar en la efectividad de la red, y puede buscarse de manera automática. Algunos algoritmos conocidos para determinar automáticamente los valores de los hiperparámetros son *grid search* and *random search*; nosotros utilizamos random search debido a su capacidad de reducir el error en el conjunto de validación de manera más rápida que grid search [5]. El parámetro para el número de neuronas intermedias toma valores en el rango $[2, 100]$. Otro hiperparámetro que usualmente es considerado, y que nosotros también consideramos en este trabajo, es el término de regularización (un coeficiente de penalización que afecta el proceso de aprendizaje de la NN); los valores para este parámetro son tomados en el rango $[-5, 3]$, espaciados uniformemente en una escala logarítmica, como es habitual en varios dominios. Para determinar los valores de los hiperparámetros y así la arquitectura final de la NN, entrenamos la red con 10 combinaciones aleatorias de valores (tomados de los rangos correspondientes) y seleccionamos la combinación que exhibe la mejor performance. Como mostramos en la siguiente evaluación experimental, el nivel de precisión logrado no exigió un ajuste adicional de los hiperparámetros de la red.

Un prototipo de nuestra técnica, basado en la implementación de perceptrones multi-capa disponible en el paquete python scikit-learn[2], se encuentra públicamente accesible para su descarga en el sitio que acompaña el artículo en el que presentamos nuestro enfoque [63]:

`https://sites.google.com/site/learninginvariants`

[2]https://scikit-learn.org/

3.3. Evaluación Experimental

En esta sección presentamos la evaluación experimental de nuestra técnica de aprendizaje, la cuál evaluamos de varias maneras. Primero, evaluamos cuan adecuado es nuestro mecanismo para generar instancias analizando cuantas de las instancias potencialmente inválidas que construimos son verdaderamente inválidas (falsifican un invariante provisto). Segundo, tomamos un benchmark de clases que implementan estructuras de datos equipadas con invariantes de clase y constructores (incluyendo rutinas de inserción); generamos los clasificadores de instancias válidas e inválidas mediante nuestra técnica, y evaluamos las métricas de *precision* y *recall* frente a los invariantes correspondientes, como es habitual en el contexto de aprendizaje automático [78]. En este contexto, comparamos nuestra técnica con Daikon [13], una herramienta dinámica para inferencia de invariantes. Finalmente, también comparamos nuestros clasificadores con invariantes producidos por Daikon, en tareas de bug finding a través de generación de tests, para varios casos de estudio de la literatura: `scheduler` del repositorio SIR [12], una implementación de árboles n-arios en el generador de parsers ANTLR[3], una implementación de conjuntos de enteros utilizando red-black trees introducida en [93], de binary search trees y binomial heaps de la evaluación realizada en [20], y de fibonacci heaps de la librería graphmaker[4].

La evaluación de nuestra técnica está guiada por las siguientes preguntas de investigación:

RQ1 *Nuestro mecanismo de generación de instancias potencialmente inválidas, es adecuado para esta tarea?*

RQ2 *Cuán precisa es la NN obtenida en la tarea de clasificación de instancias válidas e inválidas?*

RQ3 *Los clasificadores aprendidos, capturan información relevante sobre el comportamiento de las clases de modo que puedan utilizarse para mejorar tareas de bug finding?*

Cuadro 3.1: Instancias inválidas e incorrectas generadas.

Clase	k	Instancias			
		Total	Válidas	Inválidas	Inválidas inc.(%)
Singly Linked List	6	15731	2486	13245	0 (0%)
	7	41323	7867	33456	0 (0%)
	8	81199	16416	64783	0 (0%)
Singly Sorted List	6	10546	240	10306	239 (2%)
	7	34779	830	33949	760 (2%)
	8	85708	2577	83131	1966 (2%)
Doubly Linked List	6	48190	5155	43035	0 (0%)
	7	85955	9381	76574	0 (0%)
	8	136893	14801	122092	0 (0%)
Binary Tree	6	7255	193	7062	0 (0%)
	7	21647	567	21080	0 (0%)
	8	55953	1401	54552	0 (0%)
Binary Search Tree	6	19137	692	18445	365 (2%)
	7	53006	2198	50808	840 (2%)
	8	112031	5368	106663	1700 (2%)
Red Black Tree	6	8700	165	8535	131 (2%)
	7	27358	464	26894	313 (1%)
	8	73422	1323	72099	690 (1%)
Binomial Heap	6	3243	407	2836	70 (2%)
	7	2830	380	2450	58 (2%)
	8	124563	12889	111674	5213 (5%)

3.3.1. Efectividad generando Instancias Inválidas

Para evaluar la RQ1, necesitamos verificar para cada instancia potencialmente inválida, construida con nuestra técnica basada en extensiones de campos, si es verdaderamente inválida. Este experimento fue realizado de la siguiente manera. Tomamos todos los casos de estudio que acompañan la distribución[5] de Korat [6], que involucra estructuras de datos de distintas complejidades y para las que se cuenta con invariantes de clases (notar que Korat require estos invariantes expresados como predicados en Java para generación de inputs de tests [6]). Luego, extendimos cada clase con un conjunto de métodos constructores (incluyendo rutinas de inserción), y utilizamos

[3]https://www.antlr.org/
[4]https://github.com/nlfiedler/graphmaker
[5]http://korat.sourceforge.net

Randoop [68] para producir instancias válidas a partir de estos métodos para distintos scopes, y sin tener en cuenta el invariante provisto en Korat. Para cada caso, ejecutamos Randoop con 100 seeds diferentes, 10 segundos cada ejecución, y colectamos todas las instancias producidas (un total de más de 16 minutos de generación por caso de estudio). Finalmente, utilizamos nuestra técnica basada en las extensiones de campo para producir las instancias potencialmente inválidas, y verificamos su falsedad utilizando el invariante correspondiente incluido en la distribución de Korat. En estos experimentos, los métodos constructores fueron fáciles de seleccionar (mayormente el constructor de la clase y las rutinas de inserción), pero seleccionar un conjunto suficiente y al mismo tiempo pequeño de constructores puede ser una tarea sutil.

Los resultados de este experimento se resumen en el Cuadro 3.1. Este cuadro muestra, para cada caso de estudio y varios scopes (k), el número total de instancias producidos (Total), distinguiendo entre válidas (Válidas) e inválidas (Inválidas), y para estas últimas el número de inválidas incorrectas (Inválidas inc.(%)), es decir, instancias (construidas mediante nuestro mecanismo basado en extensiones de campos) que satisfacen el invariante de clase original provisto en Korat, cuando en realidad no deberían satisfacerlo.

3.3.2. Performance en la Clasificación de Instancias

Para poder evaluar la RQ2, primero realizamos el siguiente experimento. Tomamos nuevamente los casos de estudio de Korat equipados ya con nuestros métodos constructores y ejecutamos, para cada clase C, nuestra técnica de aprendizaje, obteniendo un clasificador de instancias I'_c para cada clase. En este paso, solo utilizamos los métodos constructores y sin considerar los invariantes provistos. Luego, para cada clase C, utilizamos el invariante I_c incluido en Korat para generar instancias válidas dentro de un scope k dado (es decir, *todas* las instancias válidas de tamaño menor o igual a k) usando la herramienta Korat [6]. Más aún, también colectamos las instancias producidas por Korat durante la búsqueda y que fueron descartadas debido a que no satisfacían el invariante I_c. Esta colección de instancias válidas e inválidas fue utilizada para medir precision y recall durante la clasificación de instancias. Este experimento fue realizado para valores de scopes cada vez más grandes, siempre que el número de instancias no superara el valor de 1 millón. Los resultados se resumen en el Cuadro 3.2. Para cada caso de estudio y scope, reportamos: el número de instancias válidas e inválidas

Cuadro 3.2: Precision y recall de las NNs obtenidas, en la tarea de clasificación de instancias de estructuras de datos complejas.

k	Instancias			Validación Válidas					Validación Inválidas				
	$\#\mathcal{V}$	$\#\mathcal{I}$	t	*total*	*tp*	*fp*	*P.*	*R.*	*total*	*tp*	*fp*	*P.*	*R.*
Singly Linked List													
6	2486	13245	3,9	3906 (36,3)	3906	5	99,8	100	42778 (88,8)	42773	0	100	99,9
7	7867	33456	26,6	55987 (85,9)	55897	7	99,9	99,8	725973 (98,3)	725966	0	100	99,9
8	16416	64783	75,6	960800 (98,2)	960800	1	99,9	100	10M (99,9)	9999999	0	100	99,9
Singly Sorted List													
6	240	10306	24,5	252 (4,7)	246	5	98,0	97,6	96820 (97,9)	96815	6	99,9	99,9
7	830	33949	133	924 (10,1)	873	2	99,7	94,4	1617109 (99,6)	1617109	51	99,9	100
8	2577	83131	453,1	3432 (24,9)	3142	57	98,2	91,5	10M (99,8)	9999943	290	99,9	99,9
Doubly Linked List													
6	5155	43035	18,3	55987 (90,7)	55987	8	99,9	100	465917 (98,3)	465909	0	100	99,9
7	9381	76574	240,6	960800 (99)	960800	8	99,9	100	8914750 (99,9)	8914742	0	100	99,9
8	14801	122092	978,3	1M (99,9)	999999	1	99,9	99,9	10M (99,9)	9999999	0	100	99,9
Binary Tree													
6	193	7062	12,8	197 (2)	197	0	100	100	4638 (79,4)	4638	0	100	100
7	567	21080	162,4	626 (9,4)	524	3	99,4	83,7	17848 (86,2)	17845	102	99,4	99,9
8	1401	54552	94	2056 (31,8)	2054	7	99,6	99,9	68810 (91,8)	68803	2	99,9	99,9
Binary Search Tree													
6	692	18445	99,1	731 (5,3)	720	1621	30,7	98,4	61219 (96,9)	59598	11	99,9	97,3
7	2198	50808	437,4	2950 (25,4)	2395	6105	28,1	81,1	468758 (99,4)	462653	555	99,8	98,6
8	5368	106663	574,2	12235 (56,1)	7185	109544	6,1	58,7	3613742 (97,1)	3504198	5050	99,8	96,9
Red Black Tree													
6	165	8535	29,3	327 (49,5)	231	52	81,6	70,6	25611 (98,7)	25559	96	99,6	99,7
7	464	26894	118,2	911 (49)	679	267	71,7	74,5	111101 (99,3)	110834	232	99,7	99,7
8	1323	72099	235,9	2489 (46,8)	1709	2123	44,5	68,6	493546 (99,7)	491423	699	99,8	99,5
Binomial Heap													
6	3013	31141	242,4	7602 (60,3)	6864	31	99,5	90,2	35213 (99,6)	35182	738	97,9	99,9
7	7973	70053	401,4	107416 (92,5)	100301	456	99,5	93,3	154372 (99,9)	153916	7115	95,5	99,7
8	12889	111674	289,4	603744 (97,8)	562354	34756	94,1	93,1	719450 (99,9)	684694	41390	94,2	95,1

utilizadas para entrenamiento, así como el tiempo de entrenamiento (notar que cada conjunto de entrenamiento se generó con Randoop, teniendo en cuenta los métodos constructores, y con el mecanismo basado en extensiones de campos); el tamaño de la muestra utilizada para medir precision/recall, proporcionado como el número total de instancias válidas/inválidas (notar que estas fueron generadas utilizando Korat); el número de instancias que fueron clasificadas correcta e incorrectamente (tp/tn para *true positive/negative*, fp/fn para *false positive/negative*), y los porcentajes de precision/recall

Cuadro 3.3: Precision y recall de los invariantes generados por Daikon, en la tarea de clasificación de instancias de estructuras de datos complejas.

k	Instancias		Validación Válidas					Validación Inválidas				
	$\#\mathcal{V}$	$\#\mathcal{I}$	*total*	*tp*	*fp*	*P.*	*R.*	*total*	*tp*	*fp*	*P.*	*R.*
Singly Linked List												
6	2486	13245	3906	3906	42770	8,3	100	42778	8	0	100	0
7	7867	33456	55987	55987	725964	7,1	100	725973	9	0	100	0
8	16416	64783	960800	960800	9999997	8,7	100	10M	3	0	100	0
Singly Sorted List												
6	240	10306	252	252	713	26,1	100	911	198	0	100	21,7
7	830	33949	924	924	3421	21,2	100	3978	557	0	100	14,0
8	2577	83131	3432	3432	15944	17,7	100	17781	1837	0	100	10,3
Doubly Linked List												
6	5155	43035	55987	55987	345246	13,9	100	465917	120671	0	100	25,8
7	9381	76574	960800	960800	6862849	12,2	100	8914750	2051901	0	100	23,0
8	14801	122092	1M	1M	7930490	11,1	100	10M	1069510	0	100	10,6
Binary Tree												
6	193	7062	197	197	4634	4,0	100	4638	4	0	100	0
7	567	21080	626	626	17844	3,3	100	17848	4	0	100	0
8	1401	54552	2056	2056	68806	2,9	100	68810	4	0	100	0
Binary Search Tree												
6	692	18445	731	731	40157	1,7	100	61219	21062	0	100	34,4
7	2198	50808	2950	2950	330636	0,8	100	468758	138122	0	100	29,4
8	5368	106663	12235	12235	2644744	0,4	100	3613742	968998	0	100	26,8
Red Black Tree												
6	165	8535	327	327	4122	7,3	100	25611	21489	0	100	83,9
7	464	26894	911	911	20796	4,1	100	111101	90305	0	100	81,2
8	1323	72099	2489	2489	94296	2,5	100	493546	399250	0	100	80,8
Binomial Heap												
6	3013	31141	7602	7602	13699	35,6	100	35213	21514	0	100	61,0
7	7973	70053	107416	107416	58791	64,6	100	154372	95580	0	100	61,9
8	12889	111674	603744	603744	297057	67,0	100	719450	422393	0	100	58,7

correspondientes. Tengamos en cuenta que, dado que los conjuntos de entrenamiento y de validación son generados de manera independiente, algunas instancias utilizadas para entrenar podrían también aparecer en el conjunto de validación. Hemos indicado entre paréntesis el porcentaje de instancias válidas e inválidas nuevas, es decir, aquellas que fueron utilizadas para la validación pero no fueron parte del conjunto de entrenamiento correspondiente.

Para tener una referencia de la precisión de nuestro enfoque, comparamos los clasificadores de instancias obtenidos con invariantes generados utilizan-

do Daikon [13]. El proceso que seguimos para producir estos invariantes con Daikon fue el siguiente. Para cada caso de estudio, tomamos los mismos tests que utilizamos como punto de partida en nuestra técnica (generados con Randoop), y ejecutamos Daikon a partir de ellos. Daikon produjo una lista l de invariantes probables, la cuál en todos los casos incluía propiedades inválidas (es decir, propiedades que eran verdaderas considerando los tests pero no lo eran en el caso general de la estructura correspondiente). A partir de l, generamos una lista l' filtrando manualmente las propiedades inválidas (es decir, propiedades que no valen para todas las instancias). Finalmente, evaluamos las métricas de precision/recall de los invariantes obtenidos por Daikon, para las mismas instancias que utilizamos para evaluar nuestros clasificadores. Los resultados se muestran en el Cuadro 3.3.

3.3.3. Mejoras en Bug Finding

RQ3 es la única pregunta de investigación que no requiere que las clases estén equipadas con invariantes. Para evaluar esta pregunta, tomamos de la literatura 6 implementaciones de estructuras de datos que contenían bugs: la implementación `scheduler` del repositorio SIR [12], una implementación de árboles n-arios que es parte del generador de parsers ANTLR[6], implementaciones de rutinas sobre conjuntos de enteros utilizando red-black trees, con bugs introducidos, introducidas en [93], binary search trees y binomial heaps de la evaluación empírica de [20] conteniendo un bug real cada una, y una implementación de fibonacci heaps de la librería graphmaker[7], conteniendo también un bug real. Para cada uno de estos casos de estudio, tomamos un conjunto de métodos constructores (ya provistos en las implementaciones); generamos tests utilizando Randoop a partir de los cuáles generamos un clasificador de instancias utilizando nuestra técnica, para un scope relativamente pequeño (5 en todos los casos); e inferimos además invariantes con Daikon, de la misma manera que para RQ2. Luego, realizamos una tarea de *runtime checking* con Randoop deshabilitando el chequeo de invariantes, y con Randoop habilitando el chequeo de invariantes usando: *(i)* la NN obtenida con nuestra técnica, y *(ii)* el invariante obtenido con Daikon y "filtrado" (solo manteniendo las propiedades válidas), para verificar la habilidad de bug finding en cada caso. Cada ejecución de Randoop para generación de instancias

[6]https://www.antlr.org/

[7]https://github.com/nlfiedler/graphmaker

Cuadro 3.4: Efectividad de los clasificadores en bug finding.

Caso de estudio	#bugs	Bugs detectados		
		Sin inv.	NN	Daikon
ANTLR*	1	0	1	0
Scheduler	8	3	4	3
IntTreeSet	5	0	5	3
BinTree	1	0	1	0
BinHeap	1	0	1	0
FibHeap	1	0	1	0
TOTAL	17	3	13	6

fue hecha de la misma manera que para la RQ2 (es decir, con un scope de 5), excepto para `BinHeap`, donde usamos un scope de 13 dado que un bug conocido se revela por primera vez en estructuras de ese tamaño. Para la tarea de bug finding, ejecutamos Randoop con un timeout de 10 minutos. Los resultados se muestran en el Cuadro 3.4. En el caso de ANTLR, Randoop no es capaz de detectar el bug con ninguna configuración. La razón es que, debido al mecanismo que Randoop usa para construir los tests de manera incremental, no puede generar la situación de *aliasing* que es necesaria para revelar el bug (básicamente, agregar un nodo como hijo de si mismo, una situación que la interfaz de la clase bajo análisis permite, pero que Randoop no puede producir). Sin embargo, si construimos el escenario manualmente, el clasificador basado en la NN detecta la anomalía, mientras que tanto sin utilizar invariante como utilizando el invariante producido por Daikon no es detectada. En el cuadro, el caso ANTLR está marcado con * para reflejar esta particularidad.

3.3.4. Discusión

Analicemos ahora los resultados de nuestra evaluación. Con respecto a la RQ1, nuestra técnica para producir instancias inválidas mediante la utilización de las extensiones de campo funcionó razonablemente bien. En general, menos del 5 % de las instancias supuestamente inválidas eran en realidad válidas, con una efectividad creciente para scopes más grandes. Una mirada más cercana a esos casos muestra que la mayoría de las instancias inválidas

incorrectas tienen que ver con producir cambios en campos de datos, que la NN identifica como anómalos pero que el invariante correspondiente admite. Es decir, cuando un cambio se produce en un campo de una instancia, por lo general lleva a una instancia inválida. En otras palabras, las extensiones de campos parecen resumir con precisión la admisibilidad de los valores en los campos. Un aspecto que podría afectar esta efectividad es el tiempo configurado para generar instancias válidas (y colectar las extensiones de campos). En nuestros experimentos, las múltiples ejecuciones de Randoop realizadas con diferentes seeds produjeron muestras suficientemente grandes de instancias válidas, pero, ante otros casos de estudio, el tiempo destinado a esta tarea podría ser extendido para obtener extensiones de campo más precisas.

En relación a la RQ2, nuestros resultados experimentales muestran que las NN pueden aproximar el comportamiento de invariantes de clase notablemente. De hecho, nuestra técnica produce clasificadores que logran valores de precision/recall muy altos para instancias inválidas, y valores de precision/recall significativamente mejor para instancias válidas, en comparación con las técnicas relacionadas. En otras palabras, es significativamente más probable que los casos mal clasificados sean instancias inválidas clasificadas como válidas, en lugar de lo opuesto. Este es un hecho positivo para detección de bugs, ya que confirma que los clasificadores tienden a *sobre-aproximar* los invariantes de clase (en el sentido que producirán menos falsos negativos). Los casos de estudio donde obtuvimos una menor precisión para las instancias válidas fueron Binary Search Tree y Red Black Tree. En estos casos, varias instancias inválidas son clasificadas como válidas. Analizando estas instancias mal clasificadas, confirmamos que, en estos casos de estudio, estas limitaciones en el aprendizaje tienen que ver con la complejidad de los invariantes en relación a los datos, más precisamente la propiedad de *orden*; de hecho, todos los casos inválidos que fueron incorrectamente clasificados como válidos eran correctos desde un punto de vista estructural, pero no cumplían con el orden requerido. Más experimentos utilizando NNs con arquitecturas más complejas (por ejemplo, mayor número de neuronas) podría mostrar un mejor rendimiento en estos casos. Aún así, la precisión de nuestra técnica completamente automática es significativamente mejor que los invariantes filtrados a partir de Daikon.

Con respecto a la RQ3, observemos que nuestra comparación es entre nuestra técnica, que es totalmente automática, y una instancia de Daikon con filtrado manual. En particular para esta pregunta de investigación, el esfuerzo requerido para producir la versión filtrada de los invariantes generados

por Daikon es significativa, ya que en la mayoría de los casos no teníamos invariantes como referencia para poder comparar, y por lo tanto cada invariante tuvo que ser examinado cuidadosamente para decidir si era válido o inválido. Por ejemplo, para el caso de estudio Binomial Heap, el invariante resultante luego de filtrar tiene 31 líneas e involucró observar 26 invariantes candidatos, difíciles de razonar (es una estructura de datos bastante sofisticada). Nuestros resultados muestran que logramos una capacidad de bug finding significativamente mejor en comparación con "sin invariante" y con los invariantes de Daikon "filtrado", ya que nuestros clasificadores pueden detectar 13 de los 17 bugs analizados, mientras que sin utilizar invariante solo 3 son detectados, y el invariante filtrado de Daikon solo encuentra 6 de los 17. Este es un resultado muy importante, teniendo en cuenta el esfuerzo requerido por el usuario para producir los invariantes de Daikon, y el hecho de que nuestra técnica es completamente automática.

Los 3 bugs del caso `Scheduler` que pueden ser encontrados con el chequeo de invariantes deshabilitado, lanzan excepciones, y por lo tanto no necesitan una especificación para ser detectados. Los 5 bugs restantes no producen cambios de estado, y por lo tanto no pueden ser capturados mediante el chequeo de invariantes. El bug adicional que encontramos es de hecho un bug que no está indicado explícitamente en el repositorio. Este noveno bug fue encontrado en la versión supuestamente correcta. Es un bug en el método `upgrade_process_prio`, el cuál mueve un proceso a una cola de mayor prioridad, pero no actualiza la prioridad del proceso de manera adecuada. La Figura 3.6 muestra la sentencia original donde se produce el bug y el fix correspondiente.

```
proc.setPriority(prio);
```

(a) Sentencia original.

```
proc.setPriority(prio+1);
```

(b) Sentencia correcta.

Figura 3.6: Bug en el caso `Scheduler`.

El repositorio SIR incluye también otra implementación (`Scheduler2`). Este caso de estudio no fue incluido en nuestra evaluación debido a que todos los bugs introducidos corresponden a rutinas que no producen cambios de estado, y por lo tanto no pueden ser capturados mediante chequeo de preservación de invariantes. Los bugs introducidos en la implementación Red Black Tree tomada de [93] corresponden al método `insert`. En este caso,

entrenamos nuestra NN utilizando una versión correcta del método, y la utilizamos para tratar de detectar los bugs introducidos.

3.3.5. Amenazas a la validez

Nuestra evaluación experimental se limita a implementaciones de estructuras de datos. Estas son buenas representantes de datos caracterizados por invariantes complejos, que están más allá de las técnicas de inferencia de invariantes como Daikon. Del amplio dominio de estructuras de datos, hemos seleccionado un largo conjunto para los cuáles los invariantes habían estaban presentes, dado que RQ1 y RQ2 requerían la provisión de invariantes. Este benchmark incluye estructuras de datos de variadas complejidades, incluyendo propiedades como ciclicidad, aciclicidad, balance, orden, etc. Uno podría argumentar que restringir el análisis a estos casos de estudio podría favorecer nuestros resultados. Si bien una evaluación exhaustiva de clases con restricciones complejas no es factible, consideramos que la complejidad de los invariantes (especialmente para invariantes cuya expresión va más allá de restricciones simples como comparaciones lineales) es un aspecto crucial al que queremos que se dirija nuestro enfoque, y diseñamos los experimentos teniendo en cuenta este tema. Las estructuras evaluadas corresponden a un amplio rango de complejidad, yendo desde estructuras lineales simples a otras con forma de árbol y balanceadas. Para la RQ3, no necesitamos clases que estuvieran equipadas con invariantes. Elegimos analizar estructuras de datos con bugs tomadas de la literatura, en lugar de evaluar con fallas introducidas por nosotros, para evitar sesgos no intencionales.

La evaluación se basa en gran medida en implementaciones tomadas de la literatura. Sin embargo, los casos de estudio de Korat tuvieron que ser extendidos con métodos constructores. Nuestras implementaciones fueron examinadas cuidadosamente, para asegurar que respetaban el invariante correspondiente, y para remover posibles defectos que pudieran afectar nuestros experimentos. No verificamos formalmente nuestras implementaciones, pero errores en estas hubieran implicado que instancias inválidas fueran generadas como válidas, afectando así la salida de todo el proceso de aprendizaje. Esto significa que errores en nuestras implementaciones hubieran derivado en menos precisión, es decir, obstaculizarían nuestros resultados en lugar de favorecerlos.

3.4. Conclusión

Las especificaciones en el software juegan un papel central en varias etapas del desarrollo de software, como la ingeniería de requisitos, la verificación y mantenimiento del software, entre otras. En el contexto de análisis de programas, existe una creciente disponibilidad de técnicas poderosas, incluida la generación de tests [68, 59, 1], bug finding [20, 55], localización de fallas [95, 92] y reparación de programas [91, 46, 71], para las cuáles la necesidad de especificaciones de programas se vuelve crucial. Si bien muchas de estas herramientas recurren a tests como especificaciones, en general se beneficiarían enormemente con la disponibilidad de especificaciones más generales y más sólidas, como las que proporcionan los invariantes de clase. Los invariantes se están volviendo más comunes en el desarrollo de programas, con metodologías que los incorporan [57, 48], y herramientas de análisis que pueden aprovecharlos significativamente cuando están disponibles.

En este trabajo, desarrollamos una técnica basada en NNs para inferir clasificadores de instancias que puedan ser utilizados en lugar de invariantes de clase. La técnica esta relacionada a otros enfoques [13, 82] similarmente motivados, en el sentido de que explora el comportamiento del software para la inferencia, pero también incorpora una técnica para producir instancias *inválidas*, permitiendo el entrenamiento de las NNs. Hemos analizado el uso de NNs para aprender clasificadores de instancias, y mostrado que el proceso de aprendizaje logra una alta precisión comparado con técnicas relacionadas, que nuestro mecanismo para construir instancias supuestamente inválidas es efectivo, y que los clasificadores de instancias obtenidos mejoran la detección de bugs, como lo demuestran los experimentos en un benchmark de estructuras de datos de diversa complejidad.

Además, a partir de este trabajo se abren varias líneas para trabajos futuros. Nuestras NNs son construidas con parámetros bastante estándares; por lo que ajustar variables cómo el número de capas intermedias, funciones de activación, etc., podría ser necesario, especialmente al escalar a dominios más grandes. El rendimiento de las NNs también podría mejorarse mediante *feature engineering* [32], un mecanismo que aún no hemos explorado. Nuestros experimentos se basaron en el uso de generación aleatoria para producir instancias válidas, durante la etapa inicial de la técnica. Utilizar enfoques de generación alternativos como model checking y ejecución simbólica, podría conducir a resultados diferentes, y posiblemente más precisos.

Capítulo 4

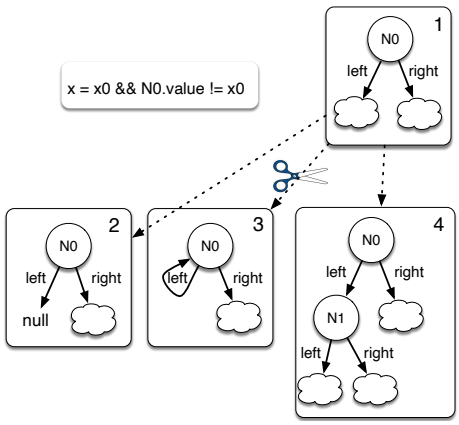
Aprendiendo a Podar Caminos en Ejecución Simbólica

La segunda técnica basada en aprendizaje que estudiamos es el uso de NNs para aprender a podar caminos en una instancia particular de ejecución simbólica, conocida como *ejecución simbólica generalizada* (Subsección 2.4.2). En esta generalización, especialmente diseñada para ejecutar simbólicamente programas que manipulan datos alojados en el heap, se asume que las ejecuciones comienzan en un estado que satisface alguna precondición o invariante, y la ejecución simbólica no solo lleva las condiciones de caminos, si no que también necesita mantener heaps parcialmente concretos. Estos heaps parcialmente concretos se van concretizando cada vez más a medida que progresa la ejecución simbólica, y su factibilidad (es decir, determinar si pueden extenderse a estructuras completamente concretas que satisfagan la precondición) necesita ser determinada, para considerar o descartar el camino actual. Esta última tarea generalmente requiere la provisión manual de rutinas para chequear la factibilidad de estructuras parcialmente concretas, que a menudo son imprecisas (no detectan todas las estructuras inviables), y aumentan el costo de la ejecución simbólica.

En este capítulo, presentamos una técnica que busca mejorar la situación anterior mediante el entrenamiento de NNs para que aprendan a determinar cuando un heap parcialmente simbólico puede ser extendido a una estructura concreta que satisfaga un invariante dado. El objetivo es entonces que estas NNs puedan ser incorporadas en un motor de ejecución simbólica y ayudar en la detección de la factibilidad de los caminos. El resto del capítulo está organizado de la siguiente manera. La Sección 4.1 introduce nuestra técnica

```
public boolean search(int x) {
  if (this.value == x) return true;
  if (this.left != null) {
    boolean inL =
        this.left.search(x);
    if (inL) return true;
    if (this.right != null)
      return this.right.search(x);
  }
  return false;
}
```

(a) Método `search`.



(b) Árbol de ejecución simbólica.

Figura 4.1: Un método de `BinaryTree`, y su ejecución simbólica utilizando lazy initialization.

a través de un ejemplo motivador. La Sección 4.2 describe todas las etapas de nuestra técnica, desde la generación de heaps parcialmente simbólicos para entrenamiento hasta la construcción y generación de las NNs. La Sección 4.3 presenta los resultados experimentales que obtuvimos al realizar ejecución simbólica incluyendo las NNs como un mecanismo de poda de caminos inviables. Y por último, la Sección 4.4 analiza las principales conclusiones de este trabajo y posible trabajo futuro.

4.1. Un Ejemplo Motivador

Consideremos nuevamente el ejemplo basado en la clase `BinaryTree`, una implementación simple de árboles binarios, introducido en la Subsección 2.4.2 e ilustrado nuevamente en la Figura 4.1 para una fácil referencia. Analicemos en más detalle la precondición del método `search`, la cuál asume que el nodo del que se comienza la búsqueda es la raíz de un árbol binario (es decir, una estructura acíclica, con cada nodo alcanzable, excepto la raíz, teniendo exactamente un nodo padre). Una implementación típica de tal precondición puede ser dada de una manera *operacional*, con código que chequea aciclicidad, como en `isBinTree()` en la Figura 4.2. Claramente, este código está diseñado para ejecutarse sobre estructuras totalmente concretas, de

```
public boolean isBinTree() {
    Set visited = new HashSet();
    visited.add(this);
    LinkedList workList = new LinkedList();
    workList.add(this);
    while (!workList.isEmpty()) {
        Node curr = workList.removeFirst();
        if (curr.left != null) {
            if (!visited.add(curr.left))
                return false;
            workList.add(curr.left);
        }
        if (curr.right != null) {
            if (!visited.add(curr.right))
                return false;
            workList.add(curr.right);
        }
    }
    return true;
}
```

Figura 4.2: Una implementación operacional de `isBinTree`.

modo que no puede ser ejecutado directamente en estructuras parcialmente simbólicas (como las ilustradas en la Figura 4.1b), ya que estas incluyen tanto referencias simbólicas como valores simbólicos de tipos básicos. Sin embargo, esta especificación operacional se puede utilizar sobre estructuras parcialmente simbólicas de la siguiente manera: intentar ejecutar el código en una estructura dada, y si es suficientemente concreta para que el código pueda ejecutarse sin caer en un valor o referencia simbólica, retornar el valor correspondiente; y si no, simplemente retornar `true`.

Utilizando este simple enfoque, le estructura parcialmente simbólica (3) en la Figura 4.1(b) puede ser considerada inviable; pero su espejo (sub-árbol izquierdo simbólico, sub-árbol derecho con un loop) sería considerado válido. El siguiente mecanismo más general resuelve este problema: la idea es comenzar ejecutando el código en una estructura parcialmente simbólica; y si es suficientemente concreta de modo que el código alcanza un estado final sin caer en una referencia o valor simbólico, devolver `true` o `false` según corresponda; y si en cambio un valor simbólico es encontrado, retroceder y continuar la ejecución. En oposición al caso previo, este caso no necesaria-

```
public boolean hybridIsBinTree() {
    Set visited = new HashSet();
    visited.add(this);
    LinkedList workList = new LinkedList();
    workList.add(this);
    while (!workList.isEmpty()) {
        Node curr = workList.removeFirst();
        if (curr.left != null && curr.left != symb) {
            if (!visited.add(curr.left))
                return false;
            workList.add(curr.left);
        }
        if (curr.right != null && curr.right != symb) {
            if (!visited.add(curr.right))
                return false;
            workList.add(curr.right);
        }
    }
    return true;
}
```

Figura 4.3: Implementación híbrida de `isBinTree`.

mente sobre-aproxima la precondición, ya que dependerá en como el código de la precondición está estructurado, y de que tipo es su valor de retorno. Pero esta aproximación tiene la ventaja de que puede ser producida sistemáticamente a partir del código de la precondición, es decir, no necesitamos pedirle al desarrollador que provea una precondición "híbrida" (una que sea capaz de operar tanto en estructuras concretas como en parcialmente simbólicas), la obtenemos a partir de la provista. La versión "híbrida" `hybridIsBinTree()` de `isBinTree()` se muestra en la Figura 4.3.

La situación descrita es el mejor escenario del enfoque que convierte sistemáticamente una precondición en su versión híbrida, ya que la función obtenida caracteriza perfectamente estructuras parcialmente simbólicas que pueden ser extendidas a estructuras completamente concretas que satisfacen la precondición original. Es decir, la precondición híbrida devuelve `true` si y solo si la estructura es completamente concreta y satisface la precondición, o si es parcialmente simbólica y puede ser extendida en una completamente concreta que satisface la precondición. Sin embargo, este escenario es extremadamente raro. Consideremos, continuando con nuestro ejemplo, una

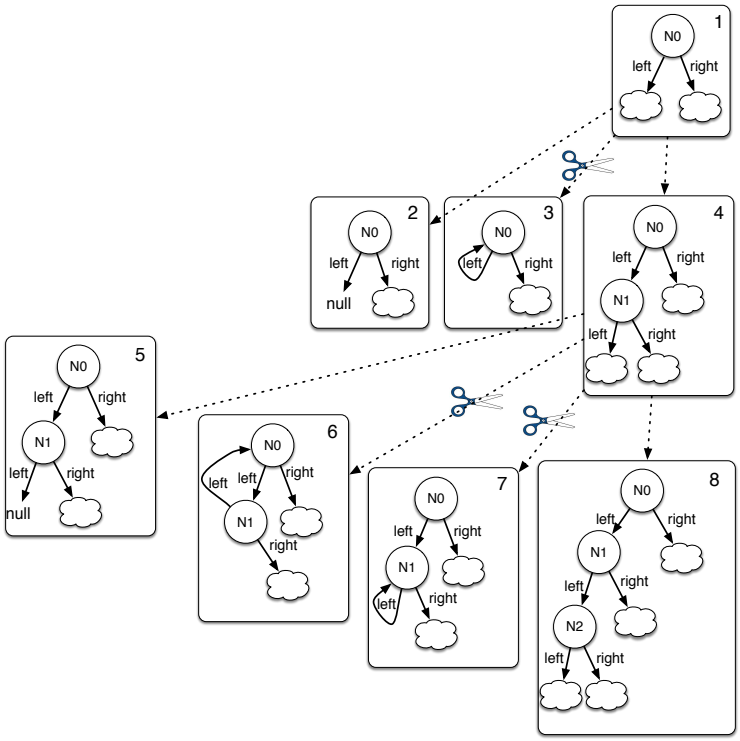


Figura 4.4: Ejecución simbólica utilizando lazy initialization a partir de un `BinaryTree` balanceado.

pequeña modificación en la que también se asume que el árbol binario es *balanceado*. Asumamos además que el método que implementa la precondición chequea primero si la estructura es de hecho un árbol binario, y luego si está balanceado, utilizando el procedimiento recursivo usual que primero chequea balanceo de los dos sub-árboles, y luego compara sus alturas. Más aún, consideremos las estructuras de la Figura 4.4, que extiende la ejecución simbólica de la Figura 4.1b. Las estructuras 6 y 7 pueden ser podadas utilizando la precondición híbrida. Pero la factibilidad de la estructura 8, por otro lado, depende de varios factores. En particular, depende del límite puesto sobre el heap o equivalentemente el límite de tamaño máximo a considerar en los caminos acotados. De hecho, notar que si el número máximo de nodos es 3, entonces la estructura debería ser considerada inviable, ya que, con los recursos disponibles, no hay manera de concretizar los nodos simbólicos de modo

que el árbol quede balanceado. Poder podar tantos caminos inviables como sea posible, y tan pronto como sea posible, es crucial para la eficiencia y la efectividad de la ejecución simbólica. Esto se debe al hecho de que continuar explorando caminos inviables no solo hace más lento el proceso de ejecución simbólica, sino que también lleva a producir resultados erróneos en el análisis, como violaciones de las precondiciones o inputs inválidos para testing (inputs que en realidad no satisfacen la precondición correspondiente).

Nuestro objetivo es contribuir exactamente en esos casos en los que la precondición híbrida no es lo suficientemente buena para la detección de caminos inviables. Básicamente, queremos construir algún tipo de "oráculo", capaz de determinar cuando una estructura parcialmente simbólica puede ser concretizada de modo que satisfaga una precondición dada. Nuestro enfoque se centra en *invariantes* de la estructura, en lugar de precondiciones, ya que estos son un núcleo en común de las precondiciones de todos los métodos de una estructura dada, y por lo tanto el costo de generar estos oráculos se puede amortizar a través del análisis de todos estos métodos. En este trabajo, los oráculos para determinar la validez o invalidez de las estructuras parcialmente simbólicas son generados utilizando NNs. Esto implica que tenemos que tolerar cierta imprecisión en estos oráculos. La imprecisión que corresponde a aceptar estructuras inválidas (es decir, falsos positivos, considerando caminos inviables del programa como factibles) ya es un problema en ejecución simbólica, por ejemplo, debido a las limitaciones en tecnologías para constraint solving (usualmente, cuando resolver un constraint está más allá de las capacidades del solver siendo utilizado o la resolución supera cierto límite de tiempo, el camino es directamente considerado como viable). Por otro lado, la imprecisión que corresponde a podar incorrectamente caminos factibles (falsos negativos), puede ser considerada más seria, ya que previene que la ejecución simbólica sea un análisis (acotado) *conservador*: una técnica de análisis que puede reportar violaciones inviables, pero que, cuando ninguna violación es reportada, garantiza la ausencia de violaciones (al menos en el límite propuesto para el análisis). Sin embargo, debemos tener en cuenta que esto solo es importante cuando la ejecución simbólica es utilizada para *verificación*; otras aplicaciones de ejecución simbólica, incluyendo generación de inputs para tests y la estimación de los peores tiempos de ejecución, no requieren conservadurismo.

La clave está entonces en cuán (im)preciso es el mecanismo de poda resultante: si es muy impreciso (dejando pasar muchos casos inválidos y rechazando muchos válidos), entonces el mecanismo no tiene valor; pero si una

NN logra buena precisión, entonces tiene el potencial de mejorar la escalabilidad de la ejecución simbólica (haciendo que sea más eficiente, permitiendo que escale a scopes más grandes, etc.), perdiendo un número relativamente pequeño de estructuras válidas y, por lo tanto, afectando levemente análisis como generación de tests o estimación de tiempo de ejecución. Cómo mostraremos luego, nuestras NNs logran una muy buena precisión para un número de implementaciones de estructuras de datos alojadas en el heap. Poder escalar a scopes más grandes es altamente relevante, ya que ciertos defectos solo aparecen en scopes suficientemente grandes. También, en algunos casos, las estimaciones del comportamiento del software se pueden determinar con precisión si se analizan en scopes lo suficientemente grandes. Por ejemplo, en estructuras que se auto-balancean, los mecanismos que se encargan de esta operación solo se ejecutan cuando hay un número de elementos lo suficientemente grande. En una implementación de binomial heaps muy estudiada, que se pensaba era correcta, un bug apareció solo cuando se puedo realizar un análisis con un scope de 13 (binomial heaps con 13 nodos) [20]. En el contexto de estimación de tiempo de ejecución en el peor caso como se propone en [51], lo lejos que la ejecución simbólica "exhaustiva" hasta un scope dado puede llegar afecta la política computada que luego es usada para scopes más grandes, y por lo tanto la precisión general en la estimación; por ejemplo, utilizar un scope de 8 lleva a una estimación cuadrática incorrecta para el método `TreeSet.add`, y scope 9 resulta en errores de out of memory.

Por supuesto, para entrenar una NN necesitamos un mecanismo para producir estructuras parcialmente simbólicas válidas e inválidas. Para ello, en este trabajo proveemos dos mecanismos para la generación del conjunto de entrenamiento. El primero consiste en la producción de estructuras parcialmente simbólicas válidas e inválidas mediante la ejecución simbólica, utilizando lazy initialization, del invariante de clase de la implementación bajo análisis. El segundo utiliza generación en *runtime*: ejecuta rutinas cuya implementación se asume correcta para construir estructuras válidas, y luego las muta para producir inválidas. En ambos casos, un mecanismo de *abstracción/concretización* es aplicado: la construcción de estructuras parcialmente simbólicas válidas adicionales a partir de las válidas, haciendo simbólicas algunas partes concretas de la estructura; y la generación de estructuras parcialmente simbólicas inválidas adicionales mediante la concretización de partes de las ya generadas. Utilizando estos mecanismos, podemos generar conjuntos de entrenamiento lo suficientemente grandes para nuestra técnica.

4.2. Podando Caminos con Redes Neuronales

En esta sección presentamos los detalles de nuestro enfoque basado en la definición y entrenamiento de una NN *feed-forward*, para aprender a distinguir las estructuras parcialmente simbólicas *factibles* (es decir, aquellas que pueden extenderse a estructuras completamente concretas que satisfacen un invariante dado) de las *inviables*, y luego utilizar esta NN como un mecanismo de poda durante ejecución simbólica generalizada. Una vista general de nuestra técnica se muestra en la Figura 4.5. Como mencionamos previamente, para poder utilizar una NN necesitamos definir un mecanismo que nos permita obtener estructuras parcialmente simbólicas que puedan ser utilizadas como *conjunto de entrenamiento*, proveer un método para representar estas estructuras como vectores, y determinar una arquitectura para la NN.

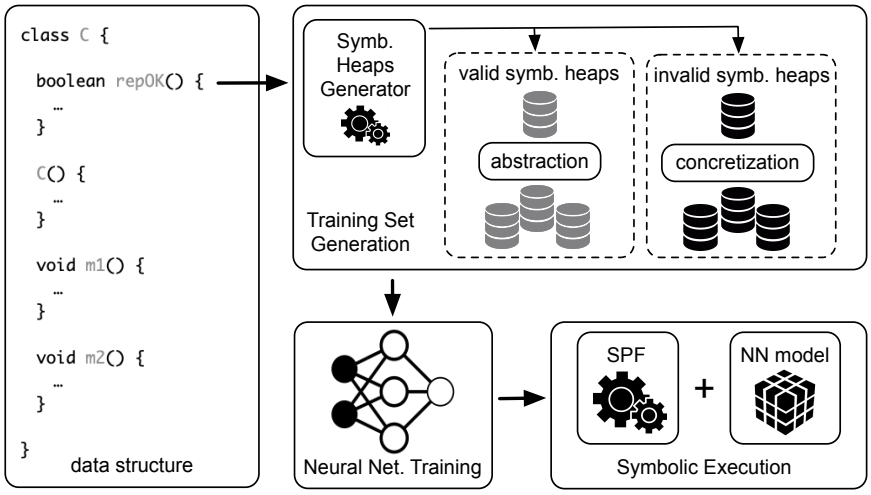


Figura 4.5: Una vista general de nuestro enfoque.

4.2.1. Generación del Conjunto de Entrenamiento

Asumamos que queremos ejecutar simbólicamente métodos de una clase C dada, para la cuál se provee un invariante de representación[48], o *repOK*, inv_c. Para entrenar una NN, primero generamos un conjunto $\mathcal{S}_f$ de estructuras parcialmente simbólicas *factibles* y otro conjunto $\mathcal{S}_i$ de estructuras parcialmente simbólicas *inviables* de la clase C, utilizando dos procedimientos

Algorithm 3: Generación del conjunto de entrenamiento basada en repOK.

```
   Input: repOK of class C.
   Output: the sets S_f and S_i.
 1 Function GEN-TRAINING-SET(repOK):
 2     S_f ← ∅;
 3     S_i ← ∅;
 4     Paths ← SYMB-EXEC(repOK);
 5     for p in Paths do
 6         s ← GET-SYMBOLIC-HEAP(p);
 7         b ← GET-OUTPUT(p);
 8         if b then
 9             S_f ← S_f ∪ {s} ∪ ABSTRACT(s);
10         else
11             S_i ← S_i ∪ {s} ∪ CONCRETIZE(s);
12     return ⟨S_f, S_i⟩
```

alternativos. Intuitivamente, en ambos casos, las estructuras *factibles* son aquellas para las cuáles existe una asignación de valores concretos a cada campo simbólico, de modo que la estructura final es consistente con el inv_c; por otro lado, las estructuras *inviables*, son aquellas para las cuáles no existe tal asignación. Para ello, implementamos un generador de heaps parcialmente simbólicos con dos mecanismos diferentes. Como explicamos debajo, el primer mecanismo genera las estructuras utilizando el repOK provisto, mientras que el segundo está basado en el uso de rutinas de la API de C, cuyas implementaciones se asumen correctas.

El Algoritmo 3 muestra el mecanismo de generación de estructuras parcialmente simbólicas basado en repOK. Esencialmente, para una clase C, comienza realizando ejecución simbólica generalizada sobre el inv_c dado. Cada vez que la ejecución simbólica alcanza un estado final, puede ser por dos razones: la estructura actual s satisface inv_c o no. Si s satisface inv_c, entonces se agrega al conjunto $\mathcal{S}_f$ de estructuras factibles, ya que claramente es factible. En este punto, probablemente, s tiene la mayoría de campos con valores concretos (al menos aquellos que son requeridos para determinar la validez de inv_c), por lo que estructuras parcialmente simbólicas *factibles* son

automáticamente creadas a partir de s, tomando aleatoriamente valores v en s y reemplazándolos por valores simbólicos. Notar que este proceso de abstracción garantiza la creación de estructuras verdaderamente factibles: s en sí mismo evidencia la factibilidad de cada estructura "menos concreta" creada a partir de ella. Similarmente, cuando el estado final de la ejecución simbólica finaliza con una estructura s que no satisface inv_c, la estructura es agregada al conjunto $\mathcal{S}_i$ de estructuras *inviables*. En esta situación, generalmente, la estructura s tendrá algunos de sus campos con valores simbólicos, ya que a menudo la rutina repOK no necesita una estructura completamente concreta para determinar que es inválida. Entonces, cuando hay campos simbólicos presentes en s, estructuras parcialmente simbólicas adicionales son creadas, reemplazando algunos campos simbólicos en s por valores concretos del tipo correspondiente. Básicamente, este enfoque es sólido porque siempre mantenemos fija la parte de s que hace que el repOK determine su invalidez; no importa como concreticemos los valores simbólicos restantes en s, la estructura siempre continuará siendo inviable.

El Algoritmo 4 implementa nuestra segunda alternativa para la generación de estructuras parcialmente simbólicas. Este mecanismo basado en la API funciona bajo la suposición de que se proveen rutinas constructoras (constructores de la clase, métodos de inserción y eliminación) asumidas correctas. Dado que la implementación de estos métodos se asume correcta, pueden ser utilizados para construir estructuras válidas completamente concretas, empleando cualquier mecanismo de generación de inputs para tests, como por ejemplo, generación random como la implementada en herramientas como Randoop [68]. De hecho, este es el mecanismo que utilizamos en nuestra evaluación para construir automáticamente estructuras válidas completamente concretas. Una vez que construimos estas estructuras válidas completamente concretas, estructuras válidas parcialmente simbólicas son creadas mediante el mismo proceso de abstracción del mecanismo previo: reemplazar campos con valores concretos por valores simbólicos. Para construir estructuras parcialmente simbólicas *inválidas*, primero necesitamos una manera de construir estructuras concretas e inválidas. Esto se realiza a partir de las válidas de la siguiente manera: dada una estructura válida s, un objeto o alcanzable desde s y un campo f en o, el valor de $o.f$ es mutado aleatoriamente a `null` o a un valor previamente observado del mismo tipo, y verificando si la estructura s' obtenida es actualmente inválida utilizando el invariante provisto inv_c. Finalmente, a partir de cada estructura inválida, un conjunto de estructuras parcialmente simbólicas *inviables* es generado, manteniendo la parte concre-

Algorithm 4: API-based training set generation

Input: $\{b_0, b_1, \ldots, b_n\}$ assumed-correct building routines of class C
Output: the sets S_f and S_i.

```
1  Function GEN-TRAINING-SET({b_0, b_1, ..., b_n}):
2      T_s ← GEN-TEST-SUITE({b_0, b_1, ..., b_n});
3      C_f ← EXECUTE-TESTS(T_s);
4      C_i ← ∅;
5      for c in C_f do
6          m ← MUTATE(c);
7          if ¬ SATISFIES(repOK, m) then
8              C_i ← C_i ∪ {m};
9      S_f ← ∅;
10     for s in C_f do
11         S_f ← S_f ∪ {s} ∪ ABSTRACT(s);
12     S_i ← ∅;
13     for s in C_i do
14         S_i ← S_i ∪ {s} ∪ ABSTRACT-INF(s);
15     return ⟨S_f, S_i⟩
```

ta que causa que la estructura original sea inválida (los "campos accedidos", en la terminología de [6], visitados al ejecutar inv_c), pero con alguno de los campos restantes apuntando ahora a valores simbólicos.

El conjunto de entrenamiento para la NN que será entrenada está compuesto por los conjuntos $\mathcal{S}_f$ y $\mathcal{S}_i$ de estructuras parcialmente simbólicas factibles e inviables, respectivamente, construidos con cualquiera de los procedimientos descritos anteriormente. Como describimos en nuestros experimentos, el mecanismo basado en repOK es inherentemente más completo en la generación del conjunto de entrenamiento, por lo que priorizamos su uso cuando es posible. Por el contrario, el mecanismo basado en la API es significativamente más eficiente, pero menos completo y por lo tanto menos preciso; cuando la ejecución simbólica del repOK se vuelve muy costosa, pasamos a generar estructuras parcialmente simbólicas utilizando el mecanismo basado en la ejecución de rutinas constructoras.

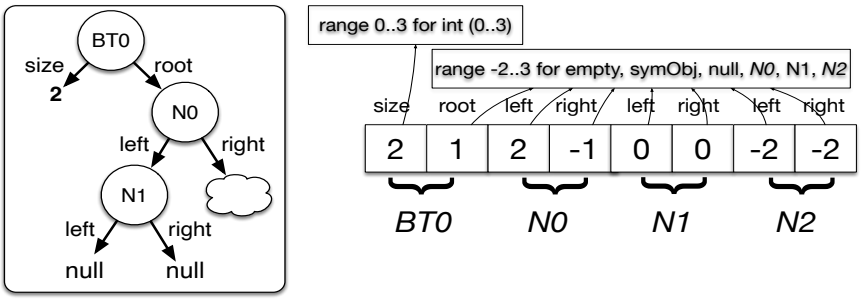


Figura 4.6: Un `BinTree` parcialmente simbólico y su representación vectorial.

4.2.2. Estructuras Simbólicas como Vectores

La mayoría de las implementaciones de NNs requieren como inputs vectores de valores, los cuáles, por lo general, están restringidos a tipos *numéricos*. Para codificar estructuras simbólicas como vectores, adoptamos el formato de vectores candidatos utilizado en Korat [6]. Dado que esta representación solo considera valores concretos, nosotros además proveemos soporte para valores simbólicos. Dado un scope k, que define rangos para tipos numéricos y un máximo número de instancias para referencias, cualquier instancia de una clase C dentro del scope k es representada en Korat por un vector conteniendo una celda por cada objeto $o' : C'$ dentro del scope y campo f de C'. El dominio de cada celda es un rango de números enteros, donde cada valor identifica unívocamente un objeto/valor dentro del scope. En nuestro caso, simplemente agregamos el valor especial -1 para representar un valor simbólico del tipo correspondiente (determinado por el índice del vector) y un valor especial -2 para representar campos no explorados, es decir, campos que todavía no fueron visitados, ya sea por un tamaño pequeño de la estructura o debido a la presencia de un valor simbólico. A modo de ejemplo, consideremos de nuevo el ejemplo de `BinaryTree`, y asumamos que nuestro análisis toma una scope de exactamente 1 árbol, 3 nodos, y `size` en el rango 0..3. Para este scope, los vectores tendrán 8 celdas, como se muestra en la Figura 4.6 (ignoramos la celda para el objeto receptor, que en este caso siempre inicializado con 1, representando el único objeto concreto de tipo `BinTree`. Además, la figura también muestra una asignación particular para el vector, y la estructura parcialmente simbólica que representa.

Vale la pena remarcar que otras técnicas alternativas para capturar heaps

como vectores podrían utilizarse, en particular *graph neural networks* [79]. Estas redes son muy generales, y permiten que uno pueda codificar grafos generales (dirigidos y no dirigidos), con nodos que podrían tener asociadas numerosas características y propiedades, a través de una codificación sofisticada basada en difusión de información. Nosotros favorecemos la representación ad hoc más simple usada por Korat porque los grafos que necesitamos codificar son muy simples, y el mecanismo de Korat maneja eficientemente la rotura de simetrías en heaps (es decir, garantiza una representación canónica para cada heap).

4.2.3. Arquitectura y Entrenamiento de la Red

Nuestra técnica está basada en el uso de NNs *feed-forward*, cuya entrada depende del tamaño de los vectores codificando estructuras parcialmente simbólicas. Asumiendo que el tamaño de los vectores es n, la capa de entrada de la red contendrá n neuronas, cada una recibiendo una posición del vector. La capa de salida siempre tendrá 1 neurona, dado que nuestro problema de clasificación solo involucra 2 clases, la clase de estructuras *factibles* y la clase de estructuras *inviables*. Sólo utilizamos una capa intermedia. El resto de los parámetros de la red como el número de neuronas de la capa intermedia y la función de activación de las neuronas, se determinan utilizando un proceso conocido como *random search* [5]. Más precisamente, utilizamos esta técnica ejecutando 10 combinaciones aleatorias de valores para los hiperparámetros, donde, de todas las estructuras parcialmente simbólicas generadas como describimos en la Subsección 4.2.1, el 75 % se utiliza para entrenamiento, y el 25 % restante para evaluar la performance de la NN. Luego, seleccionamos la combinación que exhibió el mejor rendimiento. El nivel de precisión que logramos en nuestros experimentos no exigió un ajuste adicional de los parámetros de la red (ver la siguiente sección). Tengamos en cuenta que esta división de 75 %/25 % es solo utilizada para la selección de los hiperparámetros; para la validación de la performance, realizamos un proceso de *k-fold cross validation*, como describimos más adelante.

En todos los experimentos que detallamos a continuación, construimos y entrenamos NNs feed-forward utilizando la librería Python `Keras`[1], y la librería Java `Deeplearning4j` [23] como un *wrapper* para los modelos obtenidos con Keras, de modo que estén disponibles en Java. Todos los experimen-

[1] https://keras.io

tos realizados, así como la implementación de nuestra técnica, se encuentra públicamente accesible para su descarga en el siguiente sitio:

https://sites.google.com/view/learning-symbolic-invariants

4.3. Evaluación

En esta sección presentamos la evaluación experimental de nuestra técnica. La parte principal de nuestra evaluación consiste en comparar lazy initialization, como está implementado en Symbolic PathFinder [70], con una extensión de esta herramienta, donde incorporamos nuestra técnica basada en NNs para análisis de factibilidad. Las dos técnicas comparadas funcionan de la siguiente manera. Dado un programa objetivo a ser analizado mediante ejecución simbólica, con un repOK (concreto) para la estructura de datos correspondiente, construimos la versión híbrida del repOK, de modo que pueda ser invocado en estructuras parcialmente simbólicas. El mecanismo estándar de Symbolic PathFinder utiliza lazy initialization (LI) con el repOK híbrido para podar estructuras inviables, en la que medida que el repOK es capaz de identificar. El mecanismo basado en NNs (LI+NN) es utilizado como un *complemento* a lazy initialization: luego de invocar la versión híbrida del repOK en una estructura parcialmente simbólica, si LI determina que es factible (es decir, LI deja "pasar" la estructura), invocamos a la NN para determinar si debe ser podada. Los casos de estudio involucrados en nuestro análisis son los métodos fundamentales de varias estructuras de datos encontradas en la literatura de ejecución simbólica generalizada, casos en los que las versiones "híbridas" del repOK ya estaban disponibles, incluyendo binary trees, red-black trees, AVL trees, y binomial heaps. Antes de entrar en la comparación entre las dos técnicas analizadas, primero discutiremos el desempeño de las NNs para detectar estructuras parcialmente simbólicas inviables.

4.3.1. Performance de las NNs

Para la evaluación de performance, inicialmente creamos los conjuntos de instancias simbólicas factibles/inviables utilizando el procedimiento de abstracción/concretización descrito en la Subsección 4.2.1. La arquitectura de las NNs es determinada utilizando el proceso de random search, como se menciona en la Subsección 4.2.3. Notemos que el scope (límite en el número de nodos y rangos para tipos numéricos) determina la estructura del vector; y

por lo tanto, la definición de las NNs y el análisis es realizado por cada scope. Finalmente, la evaluación del rendimiento de la red en un conjunto de datos dado consiste en medir métricas estándar, precision, recall, y f1-score [86]. La métrica precision representa la proporción de muestras positivas que fue correctamente clasificada por la NN, con respecto al número total de muestras clasificadas como positivas (esto incluye los falsos positivos, es decir, estructuras simbólicas factibles incorrectamente clasificadas como inviables). La métrica recall representa la proporción de muestras positivas y correctamente clasificadas, con respecto al número total de muestras positivas (esto incluye los falsos negativos, es decir, estructuras simbólicas inviables incorrectamente clasificadas como factibles). Finalmente, el f1-score promedia precision y recall. En todos los casos, mientras más alto sea el valor, mejor.

Para obtener estas métricas, utilizamos una técnica conocida como k-fold (con k=5) cross validation [39]. Este proceso divide el conjunto de datos en k grupos (folds), todos de aproximadamente el mismo tamaño. Después, realiza k iteraciones, donde en cada iteración uno de los folds es seleccionado como el conjunto de validación y el resto como el conjunto de entrenamiento. Luego, promedia las métricas de evaluación obtenidas en cada iteración. Dado que los conjuntos de entrenamiento no contienen estructuras repetidas, los conjuntos de entrenamiento y validación son siempre disjuntos. Notemos también que los conjuntos producidos por nuestra técnica son inherentemente no balanceados, es decir, para un scope dado, existe un número significativamente mayor de instancias simbólicas inválidas que de instancias simbólicas factibles. Para lidiar con este problema, utilizamos stratified cross validation [50], un caso particular de cross validation que garantiza que los folds preservan la distribución de clases del conjunto de datos original.

El Cuadro 4.1 resume los resultados de este primer análisis. Para cada caso de estudio y scope, reportamos el promedio de cada métrica de evaluación obtenido a partir de realizar 5-fold stratified cross validation. Precision y recall son en general muy altos, más de 91 % en todos los casos. Esto significa que nuestra técnica es, para los casos de estudio evaluados, muy efectiva en identificar correctamente estructuras parcialmente simbólicas factibles/inviables. Estos resultados están alineados con las observaciones en [90], de que los modelos de aprendizaje son en general muy buenos detectando propiedades estructurales. Sin embargo, nuestra técnica puede descartar (podar) un número de instancias factibles, debido a los falsos negativos. Es decir, el uso de nuestro técnica puede hacer que durante ejecución simbólica parte del espacio de búsqueda no sea explorado. Esto es aceptable, siempre y cuando la

Cuadro 4.1: Métricas de performance de las NNs en la clasificación de estructuras simbólicas.

Clase	**k**	**Métricas**		
		Precision(%)	**Recall(%)**	**f1-score**
BinTree	4	100	94.67	0.97
	5	100	98.60	0.99
	6	100	99.72	1.00
	7	100	99.98	1.00
	8	99.96	99.97	1.00
	9	100	99.96	1.00
TreeSet	4	97.57	96.85	0.97
	5	97.85	99.02	0.98
	6	99.56	99.63	1.00
	7	99.88	99.95	1.00
	8	99.97	99.97	1.00
	9	99.98	99.98	1.00
AvlTree	4	100.00	91.43	0.95
	5	100.00	95.65	0.98
	6	99.80	99.03	0.99
	7	99.22	98.39	0.99
	8	99.96	99.78	1.00
	9	99.91	99.87	1.00
BinomialHeap	4	97.57	99.58	0.99
	5	97.53	99.86	0.99
	6	98.64	99.76	0.99
	7	98.61	99.84	0.99
	8	98.81	99.92	0.99
	9	99.02	99.95	0.99

parte incorrectamente podada no sea significativa y si la ejecución simbólica se está utilizando para un análisis no exhaustivo (por ejemplo, testing), dado que tal análisis es inherentemente incompleto; pero puede ser inaceptable para tareas como verificación. Nuestro enfoque también puede dejar pasar estructuras no factibles en algunos casos (debido a los falsos positivos). Este es un problema que también afecta a lazy initialization, y que implica que estas técnicas exploren caminos inviables, es decir, podrían producir inputs de tests incorrectos o reportar bugs espurios.

Cuadro 4.2: Tiempos de ejecución de LI y LI+NN.

M	Tec.	Scope													
		S3	**S4**	**S5**	**S6**	**S7**	**S8**	**S9**	**S10**	**S11**	**S12**	**S13**	**S14**	**S15**	**S16**
BinTree															
bfs	LI	00:00	00:00	00:00	00:01	00:02	00:04	00:13	00:48	03:13	OOM				
	LI+NN	00:01	00:01	00:02	00:02	00:04	00:07	00:19	00:50	02:11	OOM				
dfs	LI	00:00	00:00	00:00	00:01	00:01	00:04	00:13	00:52	03:23	OOM				
	LI+NN	00:01	00:01	00:01	00:02	00:04	00:07	00:18	00:44	01:42	OOM				
TreeSet															
add	LI	00:01	00:02	00:09	00:44	03:57	23:47	TO							
	LI+NN	00:01	00:03	00:05	00:14	00:41	01:30	TO							
bfs	LI	00:00	00:00	00:00	00:01	00:04	00:13	00:45	02:40	10:28	38:50	TO			
	LI+NN	00:01	00:01	00:02	00:03	00:06	00:06	00:07	02:47*	11:09*	35:18*	43:13*	TO		
dfs	LI	00:00	00:00	00:00	00:01	00:01	00:04	00:12	00:46	03:06	TO				
	LI+NN	00:01	00:01	00:01	00:02	00:03	00:03	00:04	00:56*	04:16*	12:30*	15:14*	TO		
AvlTree															
insert	LI	00:03	00:05	00:31	02:54	15:21	TO								
	LI+NN	00:02	00:03	00:06	00:06	01:18	08:37	48:01	TO						
bfs	LI	00:00	00:00	00:01	00:01	00:04	00:13	00:47	02:47	10:57	44:22	TO			
	LI+NN	00:01	00:01	00:01	00:01	00:02	00:03	00:04	00:05	00:07	00:10	08:44*	04:26*	04:48*	32:06*
dfs	LI	00:00	00:00	00:00	00:01	00:03	00:08	00:28	01:45	06:18	23:32	TO			
	LI+NN	00:01	00:01	00:01	00:01	00:02	00:02	00:05	00:06	00:06	00:14	03:35*	06:57*	03:55*	28:35*
contains	LI	00:00	00:00	00:01	00:04	00:10	00:24	00:58	02:16	05:21	12:11	29:44	TO		
	LI+NN	00:01	00:01	00:01	00:01	00:07	00:07	00:10	00:13	00:12	01:20	11:16*	15:04*	02:01*	15:43*
BinomialHeap															
bfs	LI	00:00	00:00	00:00	00:01	00:02	00:05	00:16	00:55	03:47	12:02	52:09	TO		
	LI+NN	00:01	00:01	00:01	00:02	00:03	00:04	00:07	00:13	01:18	07:10	04:50	TO		
dfs	LI	00:00	00:00	00:01	00:01	00:02	00:04	00:10	00:31	01:44	06:30	21:46	TO		
	LI+NN	00:01	00:01	00:02	00:02	00:03	00:06	00:14	00:44	02:07	07:10	19:57	TO		
extrMin	LI	00:00	00:01	00:01	00:03	00:06	00:09	00:15	00:27	00:50	01:25	00:29	00:43	01:34	TO
	LI+NN	00:01	00:02	00:02	00:04	00:07	00:10	00:17	00:17	00:53	01:28	00:30	00:44	07:19	TO

4.3.2. LI vs. LI+NN

Analicemos ahora la parte principal de nuestra evaluación, es decir, la comparación entre LI y LI+NN. Como explicamos anteriormente, tomamos los métodos principales de las estructuras de datos analizadas, y realizamos ejecución simbólica para explorar *todos* los caminos acotados supuestamente factibles, en la medida que las técnicas son capaces de detectar, y colectamos

las instancias correspondientes. Este análisis es realizado para scopes cada vez más grandes. Los tiempos de ejecución para ambas técnicas se muestran en el Cuadro 4.2. Cada fila contiene los tiempos de ejecución (mm:ss) para cada técnica para un scope y método particular, con TO denotando timeout (de 1 hora) y OOM indicando que el proceso no se pudo completar debido al agotamiento de los 4GB de memoria del heap para la JVM. En relación a estos tiempos de ejecución, primero notemos que en el caso de binary trees, LI+NN es menos eficiente; y esto es razonable: como mencionamos anteriormente, binary tree es un caso en el que no hay margen de mejora con respecto a LI, ya que el repOK híbrido de este caso es capaz de identificar con precisión perfecta las estructuras parcialmente simbólicas factibles/inviables (por lo que nuestra técnica supone una sobrecarga para este caso). En el resto de los casos, a medida que el scope se incrementa, nuestra técnica supera a LI, con un margen notable en algunos casos, AVL en particular.

Pero evaluar el impacto de LI+NN solo en términos de tiempos de ejecución sería engañoso: nuestra técnica podría estar podando una parte significativa del espacio de búsqueda, exhibiendo más eficiencia pero al costo de un análisis menos completo. Aunque el Cuadro 4.2 sugiere que este no debería ser el caso, también medimos qué tan bien funcionan LI y LI+NN, cuando utilizamos estas técnicas para generación de tests basada en ejecución simbólica. Los resultados se muestran en el Cuadro 4.3. Para cada método y scope (S), reportamos el número exacto de estructuras factibles (#fact.), tal y como se reportan en [77] (el número exacto de estructuras factibles es conocido ya que son computados en [77] a partir de un invariante declarativo utilizando SAT solving, previniendo falsos positivos y falsos negativos). Luego, para cada técnica, scope y método, reportamos el número correspondiente de estructuras generadas (size); el porcentaje de estructuras factibles generadas (fact. %); el número de estructuras inválidas (#spurious); el tiempo que toma ejecutar el método correspondiente sobre el conjunto de estructuras producidas (exec_t); y el porcentaje de ramas cubiertas en el método correspondiente (bc %). Notar que LI solo puede podar estructuras efectivamente inviables. Por lo tanto, siempre produce el 100 % de estructuras factibles, pero el número de estructuras espurias puede ser alto para la mayoría de casos complejos. Por otro lado, la NN puede podar incorrectamente algunos casos, como esta tabla confirma (aunque el porcentaje de estructuras que se pierden es relativamente pequeño). Finalmente, notemos que incluso aunque LI+NN pierde estructuras factibles, desde el punto de vista de calidad de las test suites, medida en términos de branch coverage, las suites obtenidas con

Cuadro 4.3: Calidad de las test suites generadas con LI y LI+NN.

Método	k	#fact.	LI					LI+NN				
			size	feas. %	#spurious	$exec_t$	bc %	size	feas. %	#spurious	$exec_t$	bc %
BinTree												
bfs	5	64	64	100	0	00:01	100	64	100	0	00:01	100
	6	196	196	100	0	00:05	100	196	100	0	00:05	100
	7	625	625	100	0	00:17	100	625	100	0	00:17	100
dfs	5	64	64	100	0	00:01	100	64	100	0	00:01	100
	6	196	196	100	0	00:05	100	196	100	0	00:05	100
	7	625	625	100	0	00:17	100	625	100	0	00:17	100
TreeSet												
bfs	5	14	64	100	50	00:01	100	37	100	23	00:01	100
	6	26	196	100	170	00:05	100	185	100	159	00:05	100
	7	55	625	100	570	00:17	100	560	100	505	00:15	100
add	5	25	115	100	90	00:03	100	62	92.8	0	00:01	100
	6	75	265	100	210	00:07	100	224	92.3	0	00:06	100
	7	203	577	100	374	00:16	100	554	94.5	0	00:15	100
remove	5	91	417	100	326	00:11	79.3	192	89	111	00:05	79.3
	6	493	1542	100	1049	00:43	93.1	1060	87.2	630	00:29	93.1
	7	2229	5367	100	3138	02:30	93.1	3966	86.7	2033	01:51	93.1
AvlTree												
dfs	5	14	64	100	50	00:01	100	18	100	0	00:00	100
	6	18	196	100	178	00:05	100	21	100	0	00:00	100
	7	35	625	100	590	00:17	100	73	100	0	00:02	100
insert	5	29	165	100	136	00:04	71.4	38	93.1	11	00:01	71.4
	6	29	485	100	456	00:13	71.4	40	100	11	00:01	71.4
	7	121	1383	100	1262	00:38	100	204	72.7	116	00:05	100
remove	5	35	115	100	80	00:03	72.2	47	100	12	00:01	72.2
	6	51	427	100	376	00:11	72.2	63	100	12	00:01	72.2
	7	294	TO	-	-	-	-	333	71.7	122	00:09	100
BinomialHeap												
bfs	5	6	23	100	17	00:00	100	9	100	3	00:00	100
	6	7	41	100	34	00:01	100	14	100	7	00:00	100
	7	8	72	100	64	00:02	100	9	87.5	2	00:00	100
dfs	5	6	23	100	17	00:00	100	16	100	10	00:00	100
	6	7	41	100	34	00:01	100	32	100	25	00:00	100
	7	8	72	100	64	00:02	100	29	100	21	00:00	100
insert	5	13	25	100	12	00:00	100	23	100	10	00:00	100
	6	42	51	100	9	00:01	100	51	100	9	00:01	100
	7	79	107	100	26	00:02	100	82	76.1	3	00:02	100

LI+NN son comparables con aquellas obtenidas por LI. Además, en el caso de AVlTree.remove con scope 7, LI cae en timeout al generar la test suite,

Cuadro 4.4: Comparación entre LI+NN_{rb} y LI+NN_{ab}.

Método	**k**	**LI+NN_{rb}**			**LI+NN_{ab}**		
		time(s)	feas.(%)	#spurious	time(s)	feas.(%)	#spurious
				AvlTree			
bfs	4	00:01	100	0	00:01	100	8
	5	00:01	100	0	00:01	92.8	9
	6	00:01	100	0	00:02	83.3	11
dfs	4	00:01	100	2	00:01	87.5	9
	5	00:01	100	4	00:01	92.8	4
	6	00:01	100	4	00:01	55.5	10
insert	4	00:03	90.4	8	00:05	85.7	25
	5	00:06	93.1	11	00:07	100	19
	6	00:06	100	11	00:14	86.2	33
				TreeSet			
bfs	4	00:01	100	14	00:01	62.5	1
	5	00:02	100	23	00:01	42.8	4
	6	00:03	100	159	00:03	50	8
dfs	4	00:01	100	11	00:01	62.5	1
	5	00:01	100	23	00:01	85.7	5
	6	00:02	100	139	00:02	42.3	6

mientras que LI+NN logra un 100 % de cobertura, incluso perdiendo algunas estructuras factibles.

Nuestra técnica LI+NN tiene una limitación: incluso cuando la poda basada en la NN nos permitiría escalar la ejecución simbólica a scopes más grandes, estamos limitados por el scope máximo en el que podemos ejecutar simbólicamente el repOK (ya que necesitamos hacerlo para obtener el conjunto de entrenamiento para la NN). En este punto es donde se vuelven necesarios mecanismos alternativos para generar el conjunto de entrenamiento. Nuestro segundo mecanismo para generar el conjunto de entrenamiento está basado en la ejecución de rutinas constructoras y luego en la mutación de estructuras válidas, y puede escalar mejor que la ejecución simbólica de repOK. Sin embargo, esto tiene un costo en la precisión, ya que los conjuntos de entrenamiento obtenidos son menos completos. En el Cuadro 4.2 hemos identificado con * los casos para los que la NN utilizada fue entrenada con el mecanismo basado en las rutinas constructoras. Por ejemplo, en el método `bfs` de `TreeSet`, entrenamos con conjuntos obtenidos con el primer

mecanismo hasta scope 9, y desde ese punto en adelante, utilizamos conjuntos obtenidos con las rutinas constructoras y mutación de estructuras. Un ejemplo sobre como la precisión decrece cuando consideramos conjuntos de entrenamiento construidos a partir de las rutinas constructoras se muestra en el Cuadro 4.4 (NN_{rb} se refiere a NN basadas en repOK, y NN_{ab} a las NN basadas en la API). En el sitio acompañando la técnica se pueden encontrar más resultados.

4.4. Conclusión

A medida que la ejecución simbólica gana relevancia como una técnica de análisis de programas, y encuentra una adopción cada vez mayor tanto en entornos académicos como industriales, se hace evidente la necesidad de enfoques efectivos que amplíen la aplicabilidad de la técnica. En particular, la posibilidad de aplicar ejecución simbólica en abstracciones de datos alojadas en el heap es altamente relevante, y al mismo tiempo técnicamente muy desafiante. En este trabajo, introducimos un enfoque para mejorar lazy initialization, una de las principales técnicas para ejecutar simbólicamente código manipulando datos alojados en el heap. La técnica se basa en aprender un clasificador para datos parcialmente simbólicos, que puede ser utilizado para identificar heaps parcialmente simbólicos inviables y podar el espacio de búsqueda durante la ejecución simbólica. A diferencia de técnicas relacionadas, este enfoque no demanda más esfuerzos por parte del usuario: no se requieren especificaciones adicionales además de la especificación operacional de los datos siendo manipulados. Esta especificación estándar es aprovechada para producir datos de entrenamiento, que alimentan una NN, la cuál luego se utiliza para identificar heaps parcialmente simbólicos inválidos y oportunidades para podar los caminos de ejecución simbólica. De manera alternativa, un conjunto de rutinas asumidas correctas puede ser utilizadas para generar los datos de entrenamiento. Sin embargo, la técnica introducida, cualquiera sea el mecanismo de generación de datos de entrenamiento, es *unsound*, en el sentido de que puede conducir a la poda de caminos factibles (esto contrasta con la mayoría de las mejoras a ejecución simbólica conocidas, que solo podan casos inviables).

Hemos comparado nuestra técnica con lazy initialization, en un número de estructuras de datos de variada complejidad. Nuestros resultados muestran que nuestro enfoque basado en aprendizaje puede distinguir de manera

precisa heaps parcialmente simbólicos factibles e inviables, ayuda en la mejora de la detección de caminos simbólicos inviables, y reduce los tiempos de ejecución simbólica generalizada.

Capítulo 5

Un Enfoque Evolutivo para Traducción de Especificaciones

Este capítulo introduce otra de las técnicas basadas en aprendizaje que proponemos como parte de esta tesis, pero esta vez utilizando algoritmos evolutivos con el objetivo de traducir especificaciones escritas en un estilo *operacional* a especificaciones equivalentes escritas en un estilo *declarativo*. Este trabajo se da en el contexto de análisis de programas, donde es común que herramientas requieran especificaciones formales del software bajo análisis para realizar una tarea determinada. Mas aún, muchas de estas herramientas y técnicas requieren que tales especificaciones estén provistas en un estilo particular, o sigan ciertos patrones, con el objetivo de lograr un rendimiento aceptable. Por lo tanto, tener una especificación formal no siempre es suficiente para utilizar una técnica específica, ya que tal especificación puede no estar provista en el formalismo requerido.

En este trabajo, presentado previamente en [61], lidiamos con esta situación en el caso cada vez más común de tener una especificación *operacional*, mientras que por razones de análisis se quiere una especificación *declarativa*. Más precisamente nos enfocamos en especificaciones que capturan invariantes de clase ((Subsección 2.2.1)). Nuestra técnica propone un enfoque evolutivo para traducir invariantes de clase escritos en un lenguaje de programación secuencial, en una especificación declarativa, escrita en lógica relacional. El resto del capítulo está organizado de la siguiente manera. La Sección 5.1 introduce la técnica a través de un ejemplo ilustrativo. La Sección 5.2 presenta los detalles de nuestro algoritmo evolutivo para la traducción de especificaciones, incluyendo cada uno de los componentes del algoritmo. La Sección 5.3

```
public class SinglyLinkedList {
    private Node header;
    private int size;
    ...
}
```

```
public class Node {
    private int element;
    private Node next;
    ...
}
```

Figura 5.1: Clases Java definiendo listas simplemente enlazadas.

describe la evaluación experimental desarrollada así como los resultados obtenidos. Finalmente, la Sección 5.4 remarca las principales conclusiones de este trabajo.

5.1. Ejemplo Ilustrativo

Para motivar nuestra técnica, consideremos un escenario de análisis involucrando una estructura de datos simple, *listas simplemente enlazadas*, capturada con las clases `SinglyLinkedList` y `Node` de la Figura 5.1. Asumamos, por ejemplo, que necesitamos verificar que una rutina que manipula tal estructura de datos, como una rutina de inserción, preserva el invariante de representación, es decir, insertar un elemento en una lista *válida*, devuelve también una lista *válida*. Para proceder con esta verificación, necesitamos una especificación de qué significa que una lista simplemente enlazada sea válida. Una especificación particular, con una estilo presentado en [47], consiste en capturar el *invariante de representación* de la estructura (es decir, la condición de validez pretendida para listas simplemente enlazadas) a través de una rutina booleana, que chequea si la condición se cumple o no. Un ejemplo de este método, llamado `repOK`, indicando que las listas tienen que ser acíclicas y el que el número de nodos debe coincidir con el valor del campo `size`, se muestra en la Figura 5.2.

Un enfoque sustancialmente diferente al estilo *operacional* de utilizar código para escribir especificaciones, está basado en el uso de algún formalismo lógico adecuado, para la misma tarea. Este enfoque alternativo ha sido ampliamente utilizado, a partir de los trabajos seminales de Hoare [33] y Floyd [14], donde la lógica de primer orden es utilizada para expresar aserciones relacionadas a estados de programas, hasta lenguajes más modernos como JML [8] y Alloy [36], los cuáles debido a necesidades de poder expresivo, han extendido la lógica de primer orden con predicados de clausura y

```
public boolean repOK() {
    if (header == null) return size == 0;
    Set<Node> visited = new java.util.HashSet<Node>();
    visited.add(header);
    Node current = header;
    while (true) {
        Node next = current.getNext();
        if (next == null) break;
        if (!visited.add(next)) return false;
        current = next;
    }
    if (visited.size() != size) return false;
    return true;
}
```

Figura 5.2: Versión operacional del invariante de representación de la clase `SinglyLinkedList`.

```
one sig Null {}

sig SinglyLinkedList {}

pred repOK[this: SinglyLinkedList, next: SinglyLinkedList -> one
    SinglyLinkedList+Null, elem: SinglyLinkedList -> one Int] {
    (all n: this.header.*next | n !in n.^next) and
    (#(this.header.*next - Null) = this.size)
}
```

Figura 5.3: Versión declarativa del invariante de representación de la clase `SinglyLinkedList`, en la lógica relacional de Alloy.

alcanzabilidad. En particular, notemos que la lógica de primer de orden no es lo suficientemente expresiva para capturar la aciclicidad de las listas, en nuestro ejemplo. Un predicado declarativo, expresado en la lógica relacional de Alloy, y capturando exactamente la misma propiedad que el método `repOK` en la Figura 5.2, se muestra en la Figura 5.3.

Para entender mejor esta especificación, proveemos una breve descripción de la notación Alloy. Las signaturas (sig) declaran conjuntos de átomos (es decir, dominios de datos), con diferentes signaturas describiendo conjuntos disjuntos. El modificador `one` para `Null` indica que este dominio es un sin-

gleton; es la forma estándar de declarar constantes en Alloy. Los predicados son fórmulas "parametrizadas", es decir, fórmulas con variables libres. Las variables necesitan ser tipadas, pero los tipos no están limitados a dominios (signaturas), también pueden ser *relaciones*. Por ejemplo, el predicado `repOK` predica sobre una lista `this`, una función `next` que mapea nodos a otro nodo o a null, y una función `elem` que mapea nodos a valores enteros. La intuición es que `next` y `size` son las relaciones que capturan como las listas se relacionan con sus nodos iniciales, los nodos a su nodo "siguiente". El cuerpo del predicado está compuesto por una fórmula cuantificada (`all` es el cuantificador universal), y puede ser entendido directamente, considerando que los operadores `*` y `^` denotan clausuras reflexivo-transitiva y transitiva, respectivamente, `#` denota cardinalidad, y los conectivos proposicionales están denotados por `and` (conjunción), `or` (disyunción), `!` (negación) e `implies` (implicación); el operador `!in` denota no pertenencia. El cuerpo de la fórmula especifica que ningún nodo es alcanzable desde si mismo a través de `next`, y que el tamaño de la lista coincide con el número de nodos que no son null desde el comienzo de la lista. Para más detalles en relación a Alloy, referimos al lector a [36].

Para ilustrar la necesidad de traducciones efectivas entre distintos estilos de especificación, supongamos que solo contamos con el invariante *operacional*, especificado a través del método `repOK` en Java. Mientras que esta especificación es adecuada, por ejemplo, para generar inputs utilizando una herramienta como Korat [6], si queremos realizar verificación acotada utilizando una herramienta como TACO [20, 19], entonces esta especificación se vuelve inapropiada, ya que TACO espera una especificación *lógica*. Sin embargo, es posible traducir una especificación operacional (equivalente en contextos acotados), por ejemplo, utilizando traducciones embebidas en herramientas como TACO [20, 19] y CBMC [45]. La especificación lógica resultante de la traducción del `repOK` ilustrado en la Figura 5.2 se muestra en la Figura 5.4. Esta especificación, aunque es correcta con respecto a la semántica de la original (de nuevo, para un scope acotado particular), no es adecuada para verificación. Por ejemplo, verificar que el método `insert` preserva el invariante de representación de listas de tamaño como máximo 12 toma 3839 segundos, mientras que si utilizamos el invariante ilustrado en la Figura 5.3 toma 1648 segundos. Como mostraremos más adelante, esta diferencia en eficiencia se vuelve más notoria en estructuras de datos más complejas (ver Sección 5.3).

Este problema es la principal motivación de nuestro enfoque. Como ex-

```
pred repOK[thiz_0: SinglyLinkedList, header_0: SinglyLinkedList -> one
    (Node + Null), size_0: SinglyLinkedList -> one Int, next_0:
    SinglyLinkedList -> one (Node + Null), result_0, result_1: boolean] {
nodesToVisit_1 = thiz_0.size_0 and
current_1 = thiz_0.header_0 and ((lt[thiz_0.size_0, 0] and
result_1 = false and current_1 = current_4 and
nodesToVisit_1 = nodesToVisit_4 ) or (not lt[thiz_0.size_0,0]
and  ((current_1 = current_4 and
nodesToVisit_1 = nodesToVisit_4 ) or
(gt[nodesToVisit_1, 0] and current_1 != Null and
nodesToVisit_2 = sub[nodesToVisit_1, 1] and
current_2 = current_1.next_0 and ((current_2 = current_4 and
nodesToVisit_2 = nodesToVisit_4 ) or (gt[nodesToVisit_2, 0]
and current_2 != Null and nodesToVisit_3 = sub[nodesToVisit_2,1] and
current_3 = current_2.next_0 and ((current_3 = current_4 and
nodesToVisit_3 = nodesToVisit_4 ) or (gt[nodesToVisit_3, 0]
and current_3 != Null and nodesToVisit_4 = sub[nodesToVisit_3, 1] and
current_4 = current_3.next_0)))))) and not (gt[nodesToVisit_4, 0] and
current_4 != Null) and ((eq[nodesToVisit_4, 0] and
current_4 = Null and result_1 = true) or
(not (eq[nodesToVisit_4, 0] and
current_4 = Null) and result_1 = false))))
}
```

Figura 5.4: Versión declarativa del invariante de representación de la clase `SinglyLinkedList`, obtenida mediante traducción con preservación de semántica a partir del invariante operacional `repOK` de la Figura 5.2.

plicamos en la próxima sección, en este trabajo desarrollamos un algoritmo evolutivo para traducir especificaciones operacionales a especificaciones declarativas, con el objetivo de obtener especificaciones más adecuadas, desde el punto de vista de análisis de programas. Más precisamente, nuestro objetivo es obtener, a partir de las especificaciones operacionales como la ilustrada en la Figura 5.2, especificaciones declarativas cercanas a la ilustrada en la Figura 5.3 (en oposición a la ilustrada en la Figura 5.4), lo que nos permitiría una mayor eficiencia en ciertas tareas de análisis automático.

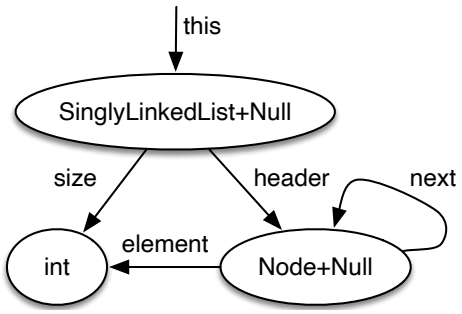


Figura 5.5: Grafo de tipos para la clase `SinglyLinkedList`.

5.2. Un Algoritmo Evolutivo para Aprender Especificaciones Declarativas

En esta sección presentamos los detalles de nuestra técnica para la traducción de especificaciones operacionales a especificaciones declarativas. Como mencionamos previamente, nuestro objetivo es computar una especificación declarativa ϕ en lógica relacional, a partir de una especificación operacional ϕ_{op}, escrita en un lenguaje de programación secuencial. Para lograr esta traducción, implementamos un algoritmo genético (Subsección 2.3.2) que busca una especificación declarativa adecuada mediante un proceso evolutivo, y cuyos componentes principales describimos a continuación.

5.2.1. Especificaciones como Cromosomas

Dado que los elementos de nuestra población son especificaciones declarativas, necesitamos un mecanismo que nos permita codificar estas especificaciones como *cromosomas*. Para capturar estas especificaciones candidatas, comenzamos a partir de la definición de la estructura, es decir, la descripción y relación de sus tipos, y construimos un *grafo de tipos*. El grafo de tipos para una estructura es construido automáticamente a partir de sus campos y sus tipos; donde los nodos representan tipos, mientras que los arcos capturan campos. A modo de ejemplo, consideremos el grafo de tipos para listas simplemente enlazadas, definidas en la Figura 5.1, que se muestra en la Figura 5.5. Estos grafos de tipos son utilizados para formar expresiones, que luego formarán parte de las especificaciones candidatas. Las expresiones son

construidas a partir de caminos en el grafo. Para que estas expresiones sean finitas, los campos recursivos son visitados como mucho una vez, e iteraciones adicionales son representadas a través de operadores de clausura. Por ejemplo, a partir del grafo de tipos de la Figura 5.5, las siguientes expresiones son computadas:

```
this
this.size
this.header
this.header.next
this.header.element
this.header.next.element
this.header.*next
this.header.*next.element
```

Más aún, en grafos de tipo con múltiples arcos conectando los mismos nodos origen y destino, su "unión" también es considerada para construir expresiones. Por ejemplo, para binary trees, existirán expresiones de la forma `this.root.left`, `this.root.right`, así como `this.root.(left+right)`. Estas expresiones son además complementadas con constantes, como `Null`, `0`, `none` (conjunto vacío), para construir otras expresiones (expresiones enteras también son generadas mediante la aplicación del operador de cardinalidad a expresiones que no son singletons). Adicionalmente, las cardinalidades de las expresiones son tenidas en cuenta (notar que las primeras 6 expresiones de arriba denotan singletons, mientras que las dos restantes denotan conjuntos de cualquier cardinalidad). Los genes, las unidades básicas (independientes) que caracterizan los cromosomas pueden ser:

- una constante booleana `true`,
- una fórmula atómica construida a partir de las expresiones originada en el grafo de tipos (incluyendo las constantes consideradas), respetando la gramática de la lógica relacional y teniendo en cuenta los tipos y cardinalidades (por ejemplo, `this.header != Null`, `this.header.*next = none`, etc.),
- una fórmula cuantificada, involucrando una variable `x` (acotada), y dos expresiones, una para el scope de `x`, y la otra para "predicar" en relación a `x` (por ejemplo, `all n : this.header.*next.element | n != 0`, siendo las dos expresiones `this.header.*next.element` y `0`); la

primera de estas expresiones está restringida a una expresión de tipo "conjunto", no un singleton.

Notemos que, de acuerdo a la gramática de Alloy, el segundo item de arriba incluye, para cada fórmula atómica α, su negación $\neg\alpha$. Esto es debido al hecho de que los operadores "booleanos" en Alloy incluyen sus contrapartes negadas (por ejemplo, `=` y `!=`, `in` y `!in`) [36].

A partir de estas construcciones, se construyen los cromosomas representando especificaciones candidatas. Un cromosoma c es simplemente un vector conteniendo genes (como los descritos previamente), y la especificación $spec_c$ representada por c es la *conjunción* de sus genes:

$$c = \left[\begin{array}{cccccc} g_0 & g_1 & g_2 & \dots & g_{n-1} \end{array} \right] \Rightarrow spec_c = g_0 \wedge g_1 \wedge g_2 \wedge ... \wedge g_{n-1}$$

En oposición a lo que es común en los algoritmos genéticos, nuestros cromosomas tienen longitud variable, y las posiciones de los genes no son consideradas. Es decir, si un gen pertenece a un cromosoma, es parte de la conjunción correspondiente, o en cualquier otra posición; esto es por supuesto debido a las propiedades de asociatividad y conmutatividad de la conjunción.

5.2.2. Población Inicial

La creación de la población $\mathcal{P}$ inicial consiste de dos pasos. En el primer paso, utilizamos la especificación operacional ϕ_{op} provista para generar un conjunto de instancias que satisface la especificación (instancias *positivas*), y un conjunto de instancias que no satisface la especificación (instancias *negativas*). Las mismas pueden ser generadas utilizando cualquier mecanismo de generación de inputs para tests (que requiera una especificación operacional). Estos son solo muestras de unas pocas entradas válidas e inválidas, que serán utilizadas en el segundo paso. Pueden generarse utilizando, por ejemplo, Korat [6], y colectando algunas de las primeras instancias generadas, e inválidas exploradas en el proceso. En nuestro caso, como describimos en la Sección 5.3, traducimos el `repOK` operacional en un predicado en lógica relacional para un scope muy pequeño, utilizando la traducción con preservación de semántica para programas acotados, y utilizamos SAT solving para producir 2 instancias válidas y 2 inválidas, es decir, 2 instancias que satisfacen el predicado obtenido, y 2 instancias que no lo satisfacen. Como segundo paso, para crear

la población $\mathcal{P}$ inicial de cromosomas, tomamos el conjunto de expresiones computadas a partir del grafo de tipos (descrito en la sección previa) y las instancias colectadas en el primer paso. Para cada expresión `expr` e instancia positiva (resp. negativa) o, el proceso funciona de la siguiente manera:

- Si `expr` denota un singleton, y evaluarla en la instancia positiva (resp. negativa) o resulta en un valor `v`, agregamos a $\mathcal{P}$ un cromosoma de tamaño 1 conteniendo el gen `expr = v` (resp. `expr != v`). Por ejemplo, si la expresión es `this.header`, y el header en la instancia positiva (resp. negativa) o es `N0`, generamos el cromosoma de único gen a partir de la expresión `this.header = N0` (resp. `this.header != N0`).

- Si la expresión `expr` denota un conjunto de cualquier cardinalidad, o un tipo no básico, entonces se agrega a $\mathcal{P}$ un cromosoma de tamaño 1 con la expresión `all n: expr | n != null`. Más aún, para cada campo `f` tal que `expr.f` es una expresión legal, se genera en $\mathcal{P}$ un cromosoma de tamaño 1 con la expresión `all n: expr | n != n.f`. Por ejemplo, a partir de la expresión `this.header.*next`, generamos cromosomas con las expresiones `all n: this.header.*next | n != null`, y `all n: this.header.*next | n != n.next`.

- Si la expresión `expr` denota un valor numérico, entonces para cada expresión `expr'` denotando un conjunto, agreamos a $\mathcal{P}$ un cromosoma de tamaño 1 conteniendo la fórmula `expr = #(expr')`. Por ejemplo, a partir de nuestro ejemplo motivador creamos un cromosoma conteniendo la fórmula `this.size = #(this.header.*next)`.

Notemos que solo el primero de los items involucra las instancias iniciales (positivas y negativas). Los otros dos items solo dependen de expresiones generadas a partir del grafo de tipos. Además, todos nuestros cromosomas iniciales son de tamaño 1. Estos servirán como las unidades básicas a partir de las cuáles se generarán especificaciones candidatas más complejas, como muestran nuestros operadores genéticos descritos debajo.

5.2.3. Operadores Genéticos

Como mencionamos previamente, los operadores genéticos son usados para explorar el espacio de búsqueda mediante la generación de nuevos individuos a partir de una población. En este trabajo utilizamos los tres operadores clásicos de los GA, *mutación*, *crossover* y *selección*.

Crossover

Nuestro algoritmo utiliza crossover de un único punto para construir nuevos cromosomas, eligiendo aleatoriamente un punto para "dividir" dos cromosomas dados, y combinando la parte inicial (resp., final) de uno de ellos con la parte final (resp., inicial) del otro. Si ambos cromosomas tienen tamaño 1, entonces su crossover es la unión de sus genes.

Mutación

La mutación permite construir nuevos individuos cambiando aleatoriamente características de individuos existentes. Nuestro algoritmo incorpora el siguiente conjunto de mutaciones:

- **Eliminación de un gen**. Es la mutación más simple, y puede ser aplicada a cualquier gen. Cambia el gen a `true`, y es equivalente a remover el gen del cromosoma:

$$c_i = \left[\ g_0 \quad \texttt{g1} \quad g_2\ \right]$$
$$c'_i = \left[\ g_0 \quad \texttt{true} \quad g_2\ \right]$$

- **Reemplazo de un operador**. Esta mutación reemplaza un operador por otro. La igualdad relacional y la inclusión relacional son reemplazadas por su contraparte negada, y viceversa. Operadores de comparación numérica como `=`, `!=`, `<`, `>`, `<=` o `>=`, son reemplazados aleatoriamente por otro operador del mismo conjunto. Las fórmulas cuantificadas también son mutadas cambiando el cuantificador, por ejemplo, `all` es reemplazado por `some`, y viceversa:

$$c_i = \left[\ g_0 \quad \texttt{this.size = 2} \quad g_2\ \right]$$
$$c'_i = \left[\ g_0 \quad \texttt{this.size >= 2} \quad g_2\ \right]$$

- **Extensión**. Utiliza el grafo de tipos y el operador de unión para extender una expresión navigacional con un nuevo campo:

$$c_i = \left[\ g_0 \quad \texttt{all n : ...| n != n.left} \quad g_2\ \right]$$
$$c'_i = \left[\ g_0 \quad \texttt{all n : ...| n != n.left.right} \quad g_2\ \right]$$

- **Inserción de un valor**. Reemplaza una sub-expresión de un gen por una constante del tipo correspondiente:

$$c_i = \left[\ g_0 \quad \texttt{all n: ...| n != n.next} \quad g_2\ \right]$$
$$c'_i = \left[\ g_0 \quad \texttt{all n: ...| n != NO} \quad g_2\ \right]$$

- **Suma/resta de un entero**. Aplica a genes que involucran enteros. Simplemente suma o resta una constante k a una expresión entera:

$$c_i = \left[\ g_0 \quad \texttt{this.size = \#(this.header.*next)} \quad g_2\ \right]$$
$$c'_i = \left[\ g_0 \quad \texttt{this.size = \#(this.header.*next) - k} \quad g_2\ \right]$$

- **Inserción/eliminación de clausura**. Inserta o elimina operadores de clausura de las expresiones, teniendo en cuenta los tipos de expresiones y sus aridades:

$$c_i = \left[\ g_0 \quad \texttt{this.header.next} \quad g_2\ \right]$$
$$c'_i = \left[\ g_0 \quad \texttt{this.header.*next} \quad g_2\ \right]$$

Como es usual en algoritmos genéticos, todas estas mutaciones son aplicadas a genes que son elegidos aleatoriamente, a partir de cromosomas de la población actual.

Selección

La operación de selección determina qué individuos se van a mantener en la próxima generación. Nuestra operación de selección se enfoca en dos aspectos. Primero, mantiene una cantidad predefinida de los individuos más aptos. Para ello, los individuos son ordenados en orden decreciente de acuerdo a sus valores de fitness, y luego los individuos del tope son seleccionados. Segundo, la selección también mantiene todos los individuos de tamaño 1 que representan propiedades válidas, es decir, aquellos cuyo valor de fitness es mayor que 0 (como explicamos debajo, estos son los individuos que aceptan todas las instancias que satisfacen ϕ_{op}). Esta última política de selección permite que el algoritmo mantenga todas las propiedades válidas descubiertas, que podrían ser parte de los individuos que mejor aproximan la especificación buscada.

5.2.4. Función de Fitness

Para computar el valor de fitness $f(c)$ de un cromosoma c dado, utilizamos tanto la especificación operacional ϕ_{op} provista como la solución candidata ϕ_c representada por c. Para ello, aprovechamos la traducción con preservación de semántica que, a partir de un scope (límite en el número de iteraciones o llamadas recursivas anidadas realizadas por ϕ_{op}), puede traducir ϕ_{op} en una especificación declarativa ϕ'_{op} equivalente en lógica relacional. Para esta traducción utilizamos un scope muy pequeño, ya que como mencionamos (y como mostramos en nuestra evaluación), las especificaciones obtenidas mediante este procedimiento muy rápidamente se vuelven poco prácticas para tareas de análisis. Primero, chequeamos por la satisfacibilidad de la fórmula:

$$\phi'_{op} \wedge \neg\phi_c$$

para el mismo scope utilizado en la traducción de ϕ_{op}. Recordemos que ϕ'_{op} resulta de la traducción de ϕ_{op} a lógica relacional, para un scope muy pequeño (en nuestros experimentos utilizamos 3). La fórmula ϕ_c representa la especificación correspondiente al candidato c (conjunción de sus genes). Si la fórmula de arriba es satisfacible, significa que existen casos, dentro del scope considerado, que *deberían* ser aceptados por ϕ_c, pero no lo son. En ese caso, $f(c) = 0$. En cambio, si la fórmula de arriba no es satisfacible, consideramos la siguiente fórmula:

$$\neg\phi'_{op} \wedge \phi_c$$

y enumeramos las instancias que, para el scope considerado, satisfacen la fórmula. Aquí, cualquier procedimiento de enumeración se ajusta al propósito (aunque diferentes técnicas pueden variar significativamente en rendimiento). En nuestro caso, utilizamos una técnica basada en SAT que realiza enumeración field-exhaustive [74]. Intuitivamente, esta enumeración omite las estructuras que cubren los mismos valores para campos que las estructuras previamente generadas, y producen más variabilidad con menos estructuras (cf. [74]). Luego, definimos $f(c)$ de la siguiente manera:

$$f(c) = (\mathsf{MAX} - \mathsf{neg}(c)) + \left(\frac{1}{\mathsf{len}(c) + 1}\right)$$

donde MAX es una constante más grande que cualquier número posible de casos negativos (se puede calcular como todas las asignaciones posibles a campos, dentro del scope dado); $\mathsf{neg}(c)$ es el número de casos que satisfacen

ϕ_c y no satisfacen ϕ'_{op} (el resultado de la enumeración mencionada); y $\mathsf{len}(c)$ es la longitud de c, es decir, el número de genes no triviales (genes que no son la constante `true`).

La razón de esta definición de la función de fitness tiene que ver con el hecho de que estamos intentando sobre-aproximar (en el sentido de que los candidatos útiles son más débiles que la especificación que estamos buscando) la especificación deseada. Esto también motiva la manera en la que capturamos las especificaciones candidatas. Por lo tanto, cuando un caso positivo no es aceptado por un candidato, simplemente lo consideramos no apto. El valor de fitness para otros candidatos tiene dos partes. Primero, mientras menos "contraejemplos", mejor; segundo, mientras más pequeña la especificación, mejor. Esta última parte puede ser pensada como una penalidad relacionada a la longitud de la fórmula, que hará que el algoritmo genético tienda a producir fórmulas más pequeñas. Por supuesto, este es un asunto secundario, y es por eso que contribuye una fracción al valor de fitness, en oposición al criterio de aceptación que dirige la búsqueda: el número de contraejemplos acercándose a cero.

5.2.5. Estructura general del Algoritmo Genético

Los elementos descritos previamente son las partes que forman nuestro algoritmo genético. Estas son puestas juntas siguiendo la estructura general de un algoritmo genético clásico (Algoritmo 1). En nuestro algoritmo en particular, la población inicial es generada produciendo cromosomas de tamaño 1, cubriendo combinaciones de las expresiones que forman parte de nuestros cromosomas. Durante todo el proceso evolutivo, la población está limitada a un tamaño de 100 individuos. Además, la selección de cromosomas para crossover está basada en una política "primero-el-mejor". Seleccionamos el 10 % más apto y, para crossover, tomamos aleatoriamente pares de ellos. Para mutación, los cromosomas con alto valor de fitness tienen una probabilidad de mutación de 0.3, y cromosomas de bajos valores de fitness tienen una probabilidad de mutación de 0.6. Más aún, para cromosomas de altos valores de fitness que son soluciones al problema (es decir, aquellos que no tienen contraejemplos) la única mutación posible es la eliminación de genes. Esto es debido al hecho de que, dado que ya tenemos una solución, solo queremos encontrar una expresión equivalente más concisa mediante la eliminación de partes redundantes del invariante. Finalmente, el algoritmo para luego de 20 evoluciones, o generaciones. Cuando una especificación adecuada es produ-

cida (es decir, una sin contraejemplos), es almacenada y su tiempo medido, pero el algoritmo no frena, con el objetivo de producir especificaciones más concisas.

La elección de los valores para los parámetros del algoritmo genético (tamaño de población, número de generaciones, probabilidades de mutación y crossover, etc) no es arbitraria. Estos valores fueron determinados a partir de ejecuciones de prueba-y-error de nuestro algoritmo genético, en un caso de estudio simple (listas simplemente enlazadas). Prueba-y-error es un mecanismo comúnmente usado, en el contexto de computación evolutiva, para definir los parámetros de la búsqueda evolutiva de manera adecuada. Es importante remarcar que, mientras que elegimos estos valores a partir de la experimentación, un solo caso de estudio fue utilizado en estos experimentos, y los mismos valores fueron utilizados para el resto de los casos de estudio. También realizamos un análisis a *posteriori* en relación a como estos parámetros afectan el rendimiento del algoritmo, en particular el tamaño de población, y la probabilidad de mutación. Estos análisis forman parte de la siguiente sección.

5.3. Evaluación Experimental

En esta sección describimos la evaluación experimental de nuestra técnica evolutiva para aprender especificaciones declarativas a partir de operacionales, analizando la eficiencia de nuestro algoritmo, la sensibilidad de algunos parámetros del GA y las mejoras producidas en algunas tareas de análisis automático como verificación y ejecución simbólica.

5.3.1. Setup experimental

Todos los experimentos fueron ejecutados en una estación de trabajo Ubuntu 16.04 LTS x86_64, con un Intel Core i7 2600, 3.40 Ghz, y 16 Gb de RAM; mientras que el algoritmo genético fue implementado utilizando la librería JGAP[1]. Nuestra evaluación considera un número de implementaciones de estructuras de datos con invariantes de variada complejidad. Estas implementaciones corresponden a las siguientes estructuras:

- singly linked lists;

[1] http://jgap.sourceforge.net

- sorted singly linked lists;
- circular linked lists;
- doubly linked lists;
- binary trees;
- binary search trees;
- heaps;
- (binary) directed acyclic graphs; and
- red-black trees.

Todas estas estructuras y sus invariantes operacionales correspondientes fueron tomadas del conjunto de ejemplos acompañando Korat [6], o son variantes simples de estas. La implementación de nuestro algoritmo genético, así como los casos de estudio considerados, pueden ser encontrados y reproducidos siguiendo los detalles en el siguiente sitio:

`https://sites.google.com/view/alloy-learning`

5.3.2. Eficiencia

La primer parte de nuestra evaluación analiza la eficiencia con la que nuestro algoritmo es capaz de aprender especificaciones declarativas a partir de operacionales. Para cada de estudio, ejecutamos el algoritmo 10 veces, con un límite de 20 generaciones (evoluciones de la población del algoritmo genético). Para generar los cromosomas de la población inicial, solo utilizamos 2 instancias positivas y 2 instancias negativas (y las expresiones producidas a partir del grafo de tipos), obtenidas a partir de traducir el `repOK` correspondiente en un predicado en lógica relacional (utilizando la traducción con preservación de semántica con un scope de 3), y consultando por la satisfacibilidad del predicado obtenido y su negación. Los resultados se muestran en el Cuadro 5.1. Reportamos las ejecuciones mínima, máxima y promedio, indicando el número de generaciones g que fueron necesarias, y el tiempo t (en segundos) requerido para generar el invariante correspondiente. Reportamos el costo de computar el primer invariante (tiempo y generaciones requeridas para obtener un invariante adecuado), y el costo de computar el "mejor"

Cuadro 5.1: Costo de generar invariantes declarativos a partir de operacionales, utilizando nuestro algoritmo evolutivo.

Estructura de Datos	Primer Invariante						Mejor Invariante					
	Min		Max		Avg		Min		Max		Avg	
	g	*t*	*g*	*t*	*g*	*t*	*g*	*t*	*g*	*t*	*g*	*t*
singly linked lists	1	*1*	1	*3*	1	*2*	1	*1*	3	*6*	2	*3*
singly linked sorted lists	1	*1*	3	*6*	1	*4*	2	*2*	7	*18*	4	*11*
singly circular lists	1	*2*	1	*3*	1	*2*	1	*2*	3	*4*	1	*3*
doubly linked lists	2	*5*	4	*19*	3	*11*	2	*7*	5	*39*	5	*23*
binary trees	1	*10*	1	*15*	1	*12*	3	*61*	9	*211*	7	*156*
binary search trees	1	*4*	2	*8*	1	*6*	4	*16*	7	*42*	4	*32*
heaps	1	*6*	3	*22*	2	*11*	2	*19*	8	*110*	6	*70*
binary DAGs	1	*3*	1	*5*	1	*5*	1	*3*	4	*25*	2	*16*
red-black trees	2	*10*	3	*17*	2	*14*	9	*71*	18	*406*	11	*234*

invariante (el algoritmo continua su ejecución luego de que un invariante ha sido encontrado, para intentar optimizarlo, es decir, hacerlo más conciso).

5.3.3. Sensibilidad de los Parámetros del GA

La segunda parte de nuestra evaluación se concentra en evaluar el impacto de diferentes parámetros en el rendimiento del algoritmo. Comencemos primero con la generación de la población inicial. Recordemos que, como describimos en la sección previa, la población inicial de nuestro algoritmo esta compuesta solo de cromosomas de tamaño 1. Para medir el impacto de esta decisión, analizamos dos versiones diferentes del algoritmo, una que comienza con cromosomas de tamaño variado generados aleatoriamente (utilizando el mismo método para generar genes descrito en la sección previa), y otra versión donde la población inicial está compuesta de solo de cromosomas de tamaño 1. Los Cuadros 5.2 y 5.3 reportan los costos promedios de computar el primer y el "mejor" invariante cuando utilizamos una población inicial de cromosomas de tamaño variado, y cuando utilizamos cromosomas de tamaño 1 como parte de la población inicial, respectivamente. En ambos casos, el mecanismo de selección para el análisis está basado en una política de "primero el mejor", sin considerar el tamaño de los cromosomas.

Continuando con el análisis del impacto de utilizar diferentes parámetros en el rendimiento de nuestro algoritmo genético, ejecutamos el algoritmo con diferentes configuraciones de parámetros y medimos la *probabilidad de*

Cuadro 5.2: Costo de generar invariantes declarativos a partir de operacionales, utilizando la política de selección "primero el mejor" y una población inicial con cromosomas de tamaño variado.

Estructura de Datos	**Primer Invariante Avg**		**Mejor Invariante Avg**		**Veces generado**
	g	t	g	t	
singly linked lists	5	*10*	6	*12*	1/10
singly linked sorted lists	10	*18*	11	*21*	9/10
singly circular lists	1	*3*	2	*9*	10/10
doubly linked lists	3	*17*	6	*29*	10/10
binary trees	13	*128*	16	*311*	8/10
binary search trees	-	-	-	-	0/10
heaps	15	*139*	17	*186*	4/10
binary DAGs	8	*56*	9	*58*	5/10
red-black trees	-	-	-	-	0/10

Cuadro 5.3: Costo de generar invariantes declarativos a partir de operacionales, utilizando la política de selección "primero el mejor" y una población inicial con cromosomas de tamaño 1.

Estructura de Datos	**Primer Invariante Avg**		**Mejor Invariante Avg**		**Veces generado**
	g	t	g	t	
s. linked lists	1	*2*	3	*6*	10/10
s. linked sort. lists	2	*6*	5	*19*	10/10
s. circular lists	1	*2*	2	*7*	10/10
doubly linked lists	3	*10*	5	*24*	10/10
binary trees	2	*20*	8	*280*	10/10
binary search trees	1	*10*	5	*60*	10/10
heaps	2	*15*	7	*80*	10/10
binary DAGs	1	*6*	3	*19*	10/10
red-black trees	2	*29*	12	*301*	9/10

optimalidad $Lopt(k)$ correspondiente. La $Lopt(k)$ está definida como la probabilidad estimada de que el algoritmo alcance una solución óptima en k generaciones. Se calcula a partir de realizar n ejecuciones del algoritmo de k generaciones, como m/n, donde m es el número de ejecuciones que produjeron una solución óptima [84]. Esta medida nos permite analizar la efectividad del algoritmo al elegir diferentes valores para los parámetros. Todas las ejecuciones de esta medida de rendimiento fueron realizadas en tres casos de estudio de diferentes complejidades: singly sorted list, binary heap y red black tree. La Figura 5.6 muestra como varía la $Lopt(50)$, para diferentes tamaños de población, de 20 a 100. Como es esperado, a medida que crece el tamaño de población, la efectividad del algoritmo aumenta, mostrando una efectivi-

Figura 5.6: Lopt(50) variando el tamaño de población.

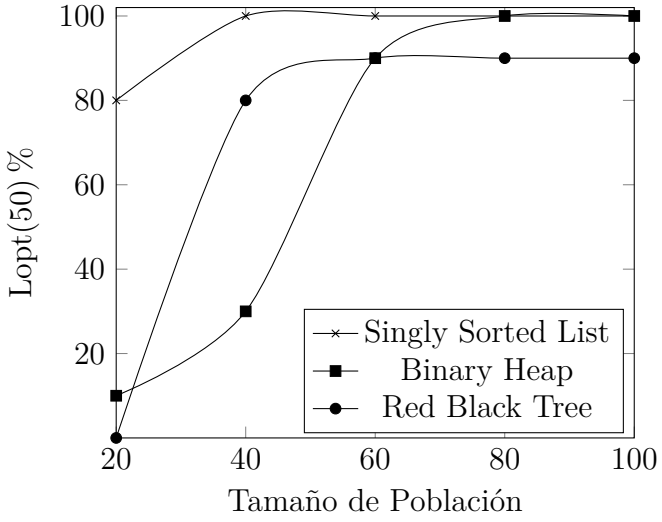
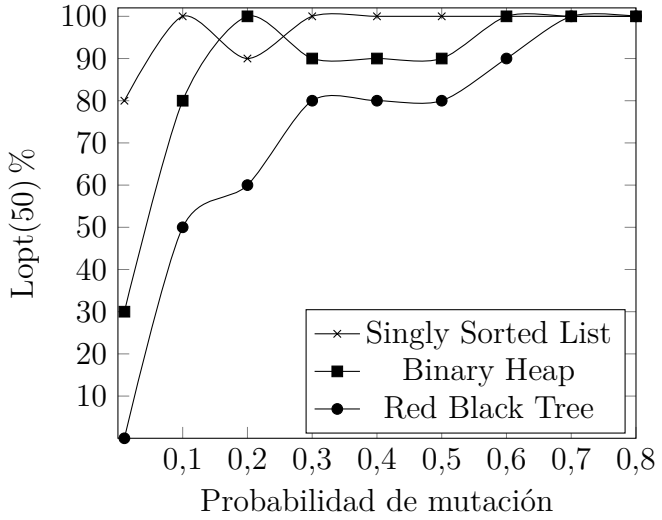


Figura 5.7: Lopt(50) variando la probabilidad de mutación y con un tamaño de población de 100.



dad sustancial para un tamaño de población de alrededor de 60 (en nuestro experimentos utilizamos 100).

También medimos la *Lopt*(50) al variar la probabilidad de mutación, para un tamaño de población fijo de 100. La Figura 5.7 muestra el impacto de

utilizar diferentes probabilidades de mutación con un tamaño de población de 100. Como se puede ver, en todos los casos la efectividad del algoritmo se incrementa con probabilidades de mutación más altas. Mientras que típicamente altas probabilidades de mutación tienen un impacto negativo en la efectividad de los algoritmos genéticos, nuestro algoritmo funciona bastante bien con altas probabilidades. En nuestra opinión, esto tiene que ver que con lo siguiente. Por un lado, nuestra población comienza con cromosomas de tamaño 1; por lo que altas probabilidades de mutación nos permiten explorar muchas mutaciones de cromosomas de tamaño 1, que son mantenidas a lo largo de las generaciones si son "válidas". Segundo, nuestra exploración de estados es altamente guiada por mutación, crossover simplemente "construye" conjunciones, pero es la mutación el mecanismo que genera los "conyuntos". Generar los conyuntos es responsabilidad de la población inicial y la mutación; nuestra política de "mantener los conyuntos válidos" es lo que permite que la operación crossover construya conjunciones válidas.

5.3.4. Impacto en Verificación y Ejecución Simbólica

La tercera parte de nuestro análisis evalúa las especificaciones aprendidas en contextos de verificación y ejecución simbólica. Primero comparamos nuestro enfoque con una traducción con preservación de semántica de especificaciones operacionales en declarativas, en un escenario de verificación específico. Verificamos, para scopes cada vez mayores (es decir, tamaño máximo de las estructuras correspondientes), que las rutinas de inserción de las estructuras de datos consideradas para el análisis, preserven el invariante de representación correspondiente. Para esta tarea utilizamos DynAlloy [18], con la especificación operacional original traducida a lógica relacional como se describe en [18, 20], y nuestras especificaciones declarativas aprendidas. Los tiempos de ejecución se muestran en mm:ss, en el Cuadro 5.4 (utilizamos un timeout de 1 hora, y marcamos con TO los análisis que superaron el timeout). Notemos que utilizamos diferentes scopes para distintos tipos de estructuras. En particular, estructuras lineales admiten scopes más grandes para análisis, comparadas con estructuras con forma de árbol.

Como contexto adicional, consideramos también la ejecución simbólica generalizada. Como describimos en la Subsección 2.4.2, un mecanismo importante para ejecutar simbólicamente programas manipulando datos alojados en el heap es lazy initialization, el cuál se puede beneficiar de la presencia de precondiciones o invariantes para podar los caminos. Más recientemen-

Cuadro 5.4: Comparación de los invariantes operacionales vs. nuestros invariantes declarativos computados, verificando la preservación del invariante en escenarios acotados.

Estructura de Datos	ϕ_{de}	ϕ_{op}	ϕ_{de}	ϕ_{op}	ϕ_{de}	ϕ_{op}	ϕ_{de}	ϕ_{op}
Scopes	**5**		**12**		**15**		**20**	
singly linked lists	< 00:01	< 00:01	00:01	00:03	00:07	00:10	01:46	01:38
singly linked sorted lists	< 00:01	< 00:01	00:30	01:54	03:25	10:16	21:51	TO
doubly linked lists	< 00:01	< 00:01	00:01	00:03	00:03	00:08	00:22	01:23
singly circular lists	< 00:01	< 00:01	00:02	00:04	00:10	00:22	01:37	02:18
Scopes	**5**		**7**		**8**		**9**	
binary trees	< 00:01	00:01	00:01	01:05	00:10	28:06	01:25	TO
binary search trees	00:01	00:09	01:32	01:13	49:11	TO	TO	TO
heaps	00:01	00:03	00:48	02:45	01:54	49:52	06:54	TO
binary DAGs	< 00:01	00:03	00:01	00:54	00:06	07:14	00:43	50:15
red-black trees	< 00:01	00:01	00:01	01:40	00:13	36:22	01:16	TO

te, en [21, 77], este mecanismo ha sido mejorado para el uso de *cotas* pre-computadas. Como sabemos, durante lazy initialization, todos los valores concretos posibles son considerados en el proceso de concretización del heap parcialmente simbólico, llevando a diferentes caminos de ejecución simbólica. Lo que estas cotas pre-computadas proveen es una *reducción* en el número de casos a considerar, basado en cómo los datos están restringidos. Por ejemplo, si una lista se asume acíclica, y estamos concretizando el campo "next" de un nodo dado en la lista, lazy initialization va a intentar todas las concretizaciones posibles, incluyendo aquellas que apuntan a nodos previos; las cotas reducen estos casos a solo dos: puede ser `null`, o apuntar a un nuevo nodo (con todos sus atributos simbólicos), ya que estos son los únicos dos casos que no violan la propiedad de aciclicidad. Esto mejora el desempeño de lazy initialization, pero con un costo adicional: los cotas son computadas a partir de especificaciones declarativas, y por lo tanto el desarrollador necesita proveer dos versiones de la misma especificación: la operacional típicamente usada en el contexto de ejecución simbólica, y la declarativa utilizada para computar estos límites. En este experimento nosotros comparamos lazy initialization estándar, con lazy initialization ayudada con cotas (con dos técnicas diferentes definidas en [77]), donde la especificación declarativa es automáticamente generada a partir de la operacional, utilizando nuestra técnica evolutiva. Los Cuadros 5.5 y 5.6 comparan los tiempos de ejecución simbólica de lazy initia-

Cuadro 5.5: Tiempos de ejecución simbólica para binary trees.

Método	Técnica	Scope								
		4	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
bfs	LI	**00:01**	**00:01**	**00:02**	00:03	00:07	00:18	01:02	04:23	15:06
	BLI	**00:01**	**00:01**	**00:02**	00:03	00:05	00:14	00:48	03:08	11:10
	RBLI	**00:01**	**00:01**	**00:02**	**00:02**	**00:04**	**00:08**	**00:21**	**01:09**	**03:38**
dfs	LI	**00:01**	**00:01**	00:02	00:03	00:05	00:16	00:56	03:43	14:38
	BLI	**00:01**	**00:01**	00:02	**00:02**	00:05	00:12	00:43	02:31	11:15
	RBLI	**00:01**	**00:01**	**00:01**	**00:02**	**00:04**	**00:10**	**00:35**	**02:12**	**08:55**
repOK	LI	**00:01**	**00:02**	**00:03**	**00:06**	00:17	00:54	03:09	11:47	41:35
	BLI	**00:01**	**00:02**	**00:03**	**00:06**	00:16	00:51	02:59	10:12	37:04
	RBLI	**00:01**	**00:02**	**00:03**	**00:06**	**00:15**	**00:44**	**02:31**	**08:11**	**29:31**

Cuadro 5.6: Tiempos de ejecución simbólica para red-black trees.

Método	Técnica	Scope								
		4	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
bfs	LI	**00:01**	**00:02**	**00:04**	00:09	00:29	01:44	06:32	23:51	1:27:29
	BLI	**00:01**	**00:02**	**00:04**	00:09	00:28	00:40	05:39	23:09	1:21:16
	RBLI	**00:01**	**00:02**	**00:04**	**00:08**	**00:26**	**01:26**	**05:05**	**19:20**	**1:10:02**
dfs	LI	**00:01**	**00:01**	**00:02**	00:03	00:07	00:21	01:20	05:20	21:44
	BLI	**00:01**	**00:01**	**00:02**	00:03	**00:05**	00:16	00:57	04:14	15:53
	RBLI	**00:01**	**00:01**	**00:02**	**00:02**	**00:05**	**00:13**	**00:52**	**03:09**	**12:42**
repOK	LI	**00:03**	00:13	01:14	07:44	50:54	TO			
	BLI	**00:03**	**00:12**	01:13	07:35	51:13	TO			
	RBLI	**00:03**	**00:12**	**01:09**	**07:24**	**48:36**	TO			

lization estándar (LI), con dos técnicas que utilizan cotas computadas con nuestras especificaciones aprendidas, bounded lazy initialization (BLI) y refined bounded lazy initialization (RBLI, que reduce casos a partir de las cotas, a medida que el heap parcialmente simbólico se va concretizando), para varias rutinas de dos estructuras de datos, binary trees y red-black trees. De nuevo, los tiempos se muestran en mm:ss, y los experimentos fueron ejecutados con un timeout de 1 hora, y los casos en los que se superó se marcan con TO. Estos resultados muestran que, mediante el uso de las especificaciones aprendidas para computar cotas, la ejecución simbólica sobre estas estructuras de datos es mejorada, por un margen significativo en algunos casos.

Cuadro 5.7: Invariantes generados vs invariantes inferidos con Daikon.

Nuestro enfoque	Daikon
singly linked lists	
`size = #(header.*next - Null)` `all n: header.*next | n !in n.^next`	`size >= 0` `header != Null`
singly sorted linked lists	

`size = #(header.*next - Null)`
`all n: header.*next | n !in n.^next`
`all n: header.*next-Null | n.val <= n.next.val`
	`size >= 0`
	`header != Null`
	`header.val = 0`

singly circular lists	
`size = #(header.*next)` `all n: header.*next | (n in n.^next)`	`size >= 0` `header = Null`
doubly linked lists	
`size = #(header.*(next+prev) - Null)` `all n: header.*(next+prev) | n = n.prev.next`	`size >= 0` `header.prev = Null`
binary trees	

`size = #(root.*(left+right) - Null)`
`all n : root.*(left+right) | (n.left.*(left+right)) &`
`(n.right.*(left+right)) in Null`
`all n : root.*(left+right) | n !in n.^(left+right)`
	`size >= 0`
	`#(root.^(left+right)) >= 0`
	`all n:Node |`
	`#(n.^(left+right))>=0`
	`#(n.left.^(left+right))>=0`
	`#(n.right.^(left+right))>=0`

binary search trees	

`size = #(root.*(left+right) - Null)`
`all n : root.*(left+right) | n !in n.^(left+right)`
`all n: root.*(left+right) |`
`(all x: n.left.*(left+right)-Null | x.val < n.val)`
`(all x: n.right.*(left+right)-Null | x.val >= n.val`
	`size >= 0`
	`all n:Node |`
	`n.left.val < n.val`
	`n.right.val > n.val`

heaps	

`size = #(root.*(left+right) - Null)`
`all n: root.*(left+right) | n !in n.^(left+right)`
`all n : root.*(left+right) | n.left.*(left+right) &`
`n.right.*(left+right) in Null`
`all n : root.*(left+right) | n.val >= n.left.val`
`all n : root.*(left+right) | n.val >= n.right.val`
	`size >= 0`
	`#(root^(left+right)) >= 0`
	`all n:Node |`
	`#(n.^(left+right)) >= 0`
	`#(n.left.^(left+right))>=0`
	`#(n.right.^(left+right))>=0`

binary DAGs	
`all n: root.*(left+right)-Null | n !in (n.^next)` `size = #(root.*(left+right)-Null)`	
red-black trees	

`root.color != Red`
`size = #(root.*(left+right)-Null)`
`all n : root.*(left+right) | n !in n.^(left+right))`
`all n : root.*(left+right) | n.left.*(left+right) &`
`n.right.*(left + right) in Null`
`all n : root.*(left+right)-Null | n.color=Red =>`
`((n.left.color!=Red) and (n.right.color!=Red))`
	`root.color = Black`
	`size >= 0`

5.3.5. Precisión

Finalmente, también analizamos la precisión de los invariantes obtenidos, en comparación con técnicas relacionadas. Comparamos nuestros invariantes aprendidos con otros obtenidos automáticamente utilizando Daikon [13] y Deryaft [53]. Daikon computa invariantes probables a partir de información en *runtime*, y por lo tanto requiere tests que ejerciten el programa bajo análisis. En este experimentos, alimentamos a Daikon con tests producidos aleatoriamente, computados utilizando Randoop [68]. Deryaft es una herramienta que genera propiedades de estructuras de datos complejas a partir de instancias; toma algunas estructuras concretas y genera un predicado que representa el invariante. Dado que Daikon computa invariantes para todos los casos involucrados, cuando un invariante se refiere a una clase auxiliar, por ejemplo `Nodo`, reportamos el invariante inferido como si fuera una propiedad de todos los nodos de la estructura. Los invariantes inferidos por Daikon son expresiones en JML. Nosotros las mostramos como expresiones en lógica relacional para facilitar la comprensión. La comparación con los invariantes obtenidos con Daikon se muestran en el Cuadro 5.7. En el caso de Deryaft, la salida es un predicado Java que representa el invariante, es decir, una rutina booleana que toma una instancia como entrada y devuelve `true` si y solo si satisface el invariante. Mientras que esta salida es *operacional*, el invariante (probable) producido está compuesto de un conjunto de propiedades de "catálogo" (por ejemplo, aciclicidad, orden, etc.), los cuáles tienen su contraparte declarativa conocida. El Cuadro 5.8 compara las propiedades generadas para cada caso de estudio utilizando nuestra técnica, con las propiedades generadas por Deryaft para las mismas estructuras de datos.

5.3.6. Discusión

Evaluemos ahora los resultados experimentales. Primero, consideremos los tiempos de ejecución de nuestro algoritmo genético. Para la mayoría de las estructuras y para la mayoría de ejecuciones, podemos computar los invariantes en unos pocos segundos. La estructura más compleja considerada, red-black trees, toma en algunos casos unos pocos minutos (alrededor de 2.5 minutos en el peor caso) para computar un invariante. En general, nuestro algoritmo corre eficientemente.

En relación a la eficiencia de nuestros invariantes en verificación acotada, nuestros invariantes declarativos muestran un beneficio significativo en análisis comparados con los operacionales, siendo la única excepción nuestro caso de estudio más simple, singly linked lists. En este caso de estudio, y para

Cuadro 5.8: Propiedades generadas por nuestro enfoque vs propiedades generadas con Deryaft.

Propiedades	Nuestro enfoque	Deryaft
singly linked lists		
Aciclicidad	✓	✓
Consistencia del size	✓	✗
singly linked sorted lists		
Aciclicidad	✓	✓
Consistencia del size	✓	✗
Orden	✓	✓
singly circular lists		
Ciclicidad	✓	✗
Consistencia del size	✓	✗
doubly linked lists		
Relación next-previous	✓	✗
Consistencia del size	✓	✗
binary trees		
Aciclicidad	✓	✓
Sub-árboles disjuntos	✓	✓
Consistencia del size	✓	✗
binary search trees		
Aciclicidad	✓	✓
Orden	✓	✓(incompleto)
Consistencia del size	✓	✗
heaps		
Aciclicidad	✓	✓
Sub-árboles disjuntos	✓	✓
Orden	✓	✓
Consistencia del size	✓	✗
binary DAGs		
Aciclicidad	✓	✓(error)
Consistencia del size	✓	✗
red-black trees		
Aciclicidad	✓	✓
Color de la raíz	✓	✗
Sub-árboles disjuntos	✓	✓
Regla de color	✓	✗
Altura nodos negros	✗	✗
Consistencia del size	✓	✗

el mayor scope considerado, el invariante obtenido mediante traducción con preservación de semántica a partir del operacional funciona mejor que el obtenido mediante nuestro algoritmo, si consideramos el tiempo de verificación (aunque por un pequeño margen). En todos los otros casos, el tiempo de verificación con el invariante obtenido mediante nuestro enfoque supera el tiempo

de verificación en comparación con los invariantes traducidos directamente. Notemos que el aprendizaje vale la pena en una gran medida, comparando el tiempo necesario para aprender y la velocidad al reemplazar el invariante "traducido" por el "aprendido". En el contexto de ejecución simbólica, la mejora provista por las cotas computadas a partir de los invariantes generados por nuestro enfoque sobre lazy initialization, también es significante en algunos casos. Por ejemplo, en el método `bfs` en el caso `BinTree`, RBLI (que utiliza cotas computadas con el invariante aprendido), es más de 4 veces más rápido que LI para scope 12. En otros casos el margen es menor, pero vale la pena mencionar que el computo de cotas es amortizado a través de todos los métodos para la misma clase, por lo que vale la pena aprender el invariante declarativo y luego computar las cotas.

Por supuesto, ninguna de las dos primeras partes de nuestro análisis es significativa si nuestros invariantes no son precisos. Nuestra tercera parte de los análisis confirma que los invariantes generados son de hecho precisos, comparados con los esperados. De hecho, en todos los casos excepto en red-black trees, aprendemos un invariante que es *equivalente* al `repOK`. Para chequear equivalencia, además de inspeccionar manualmente los invariantes obtenidos, enumeramos de manera acotada y exhaustiva las instancias que satisfacen el `repOK` utilizando Korat, para varios scopes elegidos, y comparamos el número de instancias obtenidas con el número de instancias acotadas que satisfacen nuestra especificación aprendida. En el caso de red-black trees, generamos la mayor parte del invariante esperado, excepto la propiedad de "altura de nodos negros". Esta propiedad indica que *"el número de nodos negros en todos los caminos desde la raíz hasta una hoja es el mismo"*. Esta restricción no es expresable con las expresiones que nuestro algoritmo genético considera, por lo que constituye una limitación de nuestra técnica.

En relación a los mecanismos alternativos para generar invariantes que consideramos, si comparamos con Daikon, nuestro enfoque construye especificaciones más precisas. De hecho, como el Cuadro 5.7 muestra, Daikon es capaz de computar invariantes más débiles, algunas veces incorrectos, resultando en propiedades que valen para los tests utilizados para la inferencia, pero no son verdaderos en el caso general. Por ejemplo, para doubly linked lists y binary search trees, Daikon computa algunos invariantes probables que no son ciertos en el caso general; como una muestra, la propiedad `header.prev == last.next`, que Daikon computa, solo vale para listas doblemente enlazadas no vacías (la implementación es acíclica, sin nodo centinela). Con respecto a Deryaft, aunque la herramienta muestra cierta precisión en algu-

nos casos de estudio, no computa un invariante adecuado en varios casos. El problema de precisión, en nuestra opinión, tiene que ver con el hecho de que Deryaft intenta construir un invariante solo a partir de ejemplos positivos, en oposición a nuestro caso. En nuestros experimentos, alimentamos Deryaft con el mismo conjunto de ejemplos positivos que utilizamos para computar la fitness de nuestro algoritmo, los cuáles fueron generados a partir de la especificación operacional ϕ_{op} (el input de nuestro algoritmo).

5.4. Conclusión y Trabajo Futuro

La creciente disponibilidad de tecnologías automáticas basadas en métodos formales evidencia la falta de especificaciones formales acompañando los sistemas de software, mientras que al mismo tiempo contribuyen a remarcar su necesidad. De hecho, muchas herramientas para análisis de programas, incluyendo *runtime checkers*, y herramientas de análisis estático para verificación, localización de fallas, generación de tests y *bug finding*, requieren especificaciones formales. En este trabajo, discutimos sobre el hecho de que, incluso en casos donde uno tiene una especificación formal disponible, muchas veces esta especificación no es adecuada para el tipo de análisis, herramienta o técnica, de interés. Estudiamos esta situación en el caso particular en el que una especificación operacional, representada a través de código, está disponible, pero uno necesita que tal especificación sea provista mediante un formalismo lógico. En este contexto, propusimos un algoritmo evolutivo que produce esa especificación declarativa a partir de una operacional, y mostramos que, para un conjunto de casos de estudio compuestos de estructuras de datos de complejidad variada, el algoritmo es capaz de producir invariantes de representación declarativos adecuados, a partir de sus contrapartes operacionales. Mas aún, mostramos que estos invariantes aprendidos son más adecuados para análisis, en particular para verificación acotada, que realizar una traducción con preservación de semántica a partir de la operacional y utilizar esos invariantes para el mismo análisis. También mostramos que los invariantes aprendidos pueden contribuir a mejorar otros análisis automáticos, en particular lazy initialization, y que nuestro algoritmo produce, para los casos de estudio analizados, especificaciones que son significativamente más precisas que aquellas generadas por técnicas de inferencia relacionadas.

Este trabajo abre varias líneas de trabajo futuro. Como explicamos en esta tesis, nos concentramos en propiedades de estructuras enlazadas, y todo

el diseño de nuestro algoritmo y las expresiones que el mismo soporta hace que en ocasiones no sea posible aprender algunas propiedades relevantes (la propiedad relacionada a altura de los nodos negros en red-black trees es un ejemplo de esta situación). Una línea obvia de investigación es trabajar en una generalización de nuestro enfoque, para permitir aprender un conjunto más rico de especificaciones. Nuestros casos de estudio están limitados a invariantes de representación de estructuras de datos, por lo que analizar el enfoque en otro tipos de programas, es también parte de nuestros planes. En particular, intentar aprender especificaciones a partir de programas más grandes traerá problemas de escalabilidad, con los que deberemos lidiar. Finalmente, nuestro enfoque operacional-a-declarativo permite interconectar herramientas y técnicas de análisis, algunas de las cuáles hemos mencionado. Planeamos aprovechar nuestro algoritmo evolutivo para implementar usos cruzados de estas herramientas.

Capítulo 6

Inferencia de Poscondiciones con Computación Evolutiva

Este capítulo presenta nuestra segunda técnica basada en algoritmos evolutivos para la inferencia de especificaciones, introducida previamente en [64]. Más precisamente, este trabajo presenta un mecanismo que, dado un método Java, automáticamente produce una especificación del comportamiento actual del método, en la forma de una poscondición (Subsección 2.2.2). La técnica está basada en la generación de pares de estados pre/post *válidos* e *inválidos* (es decir, pares de estados que representan, y que no representan, el comportamiento actual del método, respectivamente), los cuáles guían a un algoritmo genético para producir una aserción al estilo JML que caracteriza los pares válidos, dejando afuera a los inválidos. La idea es, mediante el uso de nuestra técnica, obtener aserciones precisas que puedan ser utilizadas para diversas tareas de análisis de programas, particularmente en contextos de regresión. El resto del capítulo está organizado de la siguiente manera. La Sección 6.1 introduce nuestra técnica a través de un ejemplo motivador. La Sección 6.2 presenta EvoSpex, la herramienta que implementa nuestro mecanismo para inferencia de poscondiciones. La Sección 6.3 presenta la evaluación experimental de nuestra técnica sobre un conjunto de proyectos Java, con implementaciones complejas de clases basadas en referencias. Y finalmente, la Sección 6.4 discute las principales conclusiones de este trabajo.

6.1. Un Ejemplo Motivador

A modo de ejemplo, consideremos la clase Java `AvlTreeList` que implementa listas, ilustrada parcialmente en la Figura 6.1[1]. Esta clase implementa operaciones de listas con árboles balanceados, con operaciones de inserción y eliminación de elementos en tiempo $O(\log n)$, a diferencia de las implementaciones clásicas de listas que utilizan arreglos o listas enlazadas. Analicemos el método `add`, el cuál inserta un elemento en la lista. Notemos como la precondición del método es capturada en el mismo código fuente, verificando la validez del índice para la inserción y que el árbol no ha alcanzado su tamaño máximo. Por otro lado, la poscondición del método no está presente en esta implementación. Tener la poscondición permite múltiples aplicaciones, en particular como aserciones para testear mejoras futuras de este método, y como una descripción declarativa de lo que este método hace (cómo opera sobre la estructura de datos), entre otras. Sin embargo, escribir la especificación no es trivial, y por lo tanto, encontrar una expresión adecuada para la poscondición es un problema importante.

Una herramienta conocida para asistir al desarrollador en esta situación es Daikon [13]. Daikon realiza detección de invariantes probables en *runtime*; ejecuta el programa en un conjunto de casos de tests, y observa que propiedades valen durante estas ejecuciones en puntos de programa particulares, como después de invocar los métodos. Luego, sugiere como invariantes probables aquellas propiedades que no fueron falsificadas por ninguna ejecución, o equivalentemente, que fueron verdaderas para todas las ejecuciones observadas. La calidad de los invariantes obtenidos depende fuertemente de las ejecuciones de los programas considerados por Daikon (es decir, el conjunto de tests que el usuario provee), y el conjunto de expresiones candidatas a ser consideradas. En particular, para el método `add` de la Figura 6.1, Daikon produce la poscondición que se muestra en la Figura 6.2 cuando lo alimentamos con una test suite que construye *todos* los árboles válidos de hasta tamaño 4. La poscondición que se muestra es la producida por Daikon, pero manualmente filtrada quitando las expresiones inválidas (provocando falsos positivos) que no pudieron ser falsificadas por la suite. Aún así, como se puede ver, la poscondición generada en este caso es relativamente débil: esperaríamos tener alguna información adicional sobre como los atributos de los nodos se manipulan en esta implementación de listas con árboles. La razón por la que

[1]Esta implementación fue tomada de `https://www.nayuki.io/page/avl-tree-list`.

```
import java.util.AbstractList;

public final class AvlTreeList<E> extends AbstractList<E> {

    private Node<E> root;

    public void add(int index, E val) {
        if (index < 0 || index > size())
            throw new IndexOutOfBoundsException();
        if (size() == Integer.MAX_VALUE)
            throw new IllegalStateException("Max size reached");
        root = root.insertAt(index, val);
    }

    private static final class Node<E> {

        private E value;
        private int height;
        private int size;
        private Node<E> left;
        private Node<E> right;

        public Node<E> insertAt(int index, E obj) {
            assert 0 <= index && index <= size;
            if (this == EMPTY_LEAF)
                return new Node<>(obj);
            int leftSize = left.size;
            if (index <= leftSize)
                left = left.insertAt(index, obj);
            else
                right = right.insertAt(index-leftSize-1, obj);
            recalculate();
            return balance();
        }

    }
}
```

Figura 6.1: Método de inserción de la clase `AvlTreeList`.

Daikon produce esta poscondición simple en este caso tiene que ver con el conjunto de expresiones candidatas que considera, las cuáles son producidas a partir de la definición del programa, y que están restringidas a propiedades

```
// height
this.root.height >= old_this.root.height &&
this.root.height >= old_this.root.left.height &&
this.root.height >= old_this.root.right.height &&
// size
this.root.size > old_this.root.height &&
this.root.size > old_this.root.left.height &&
this.root.size > old_this.root.right.height &&
// left height
this.root.left.height <= old_this.root.height &&
this.root.left.height >= old_this.root.left.height &&
this.root.left.height >= old_this.root.right.height &&
// right height
this.root.right.height <= old_this.root.height &&
this.root.right.height >= old_this.root.left.height &&
this.root.right.height >= old_this.root.right.height
```

Figura 6.2: Poscondición generada por Daikon para el método `add` de la clase `AvlTreeList`.

de programas relativamente simples (por ejemplo, restricciones estructurales, chequeos de pertenencia, etc., no son consideradas) [13].

Nuestro objetivo es proveer poscondiciones más fuertes en casos como este. En esencia, nuestro enfoque es similar al de Daikon: el método bajo análisis es ejecutado para diferentes entradas, y a partir de la información extraída de estas ejecuciones, proponemos una poscondición para el método. Sin embargo, hay múltiples diferencias. En primer lugar, nuestro enfoque está basado en generar ejecuciones para el método bajo análisis de manera *exhaustiva acotada*, a diferencia de Daikon, que quiere que los tests sean provistos (en el ejemplo de arriba, la suite que utilizamos fue la misma que nuestra técnica produce). Nuestra técnica para generar la suite exhaustiva acotada está basada en ejercitar la API de las entradas del programa bajo análisis, a diferencia de técnicas relacionadas que requieren una especificación [6, 42]. En segundo lugar, al intentar determinar la poscondición de un método consideramos tanto estados de programas *válidos* como *inválidos* (aunque, como explicamos más adelante, el mecanismo para generar estados inválidos podría generar válidos incorrectamente), en lugar de solo ejecuciones *válidas*, como es el caso de Daikon. Tercero, nuestro enfoque está basado en evolucionar especificaciones, en lugar de considerar propiedades candidatas no falsificadas.

```
// root
this.root != null &&
this.root.left != null &&
// height
all n : this.root.*(left+right) : (
    n.left != null => n.height > n.left.height &&
    n.right != null => n.height > n.right.height
) &&
// size
old_this.root.size < this.root.size &&
this.root.size == #(this.root.*(left+right - null)) - 1 &&
all n : this.root.*(left+right) : (
    n.left != null => n.size > n.left.size &&
    n.right != null => n.size > n.right.size
) &&
// arguments
index != this.root.size &&
val in this.root.*(left+right).value &&
// structural
all n : this.root.*(left+right) : n !in n.^(left + right)
```

Figura 6.3: Poscondición generada por EvoSpex para el método `add` de la clase `AvlTreeList`.

Los detalles de nuestra técnica se presentan en la siguiente sección. A modo ilustrativo, para el método `add` de la clase `AvlTreeList`, la poscondición que obtenemos mediante nuestra técnica es la que se muestra en la Figura 6.3. Notemos como la actualización del `size` (refiriéndonos a la relación entre el estado pre y pos) y la pertenencia del elemento insertado son capturadas, así como algunas propiedades estructurales de la representación.

6.2. EvoSpex

En esta sección presentamos los detalles de EvoSpex, nuestra técnica para inferir poscondiciones de métodos. El enfoque general se muestra en la Figura 6.4. La técnica se compone de dos fases principales: *(i)* la generación de estados y *(ii)* la inferencia de las poscondiciones. Durante la generación de estados, producimos los pares de estados pre/pos del programa que luego serán utilizados en la fase de inferencia para guiar la búsqueda del algoritmo

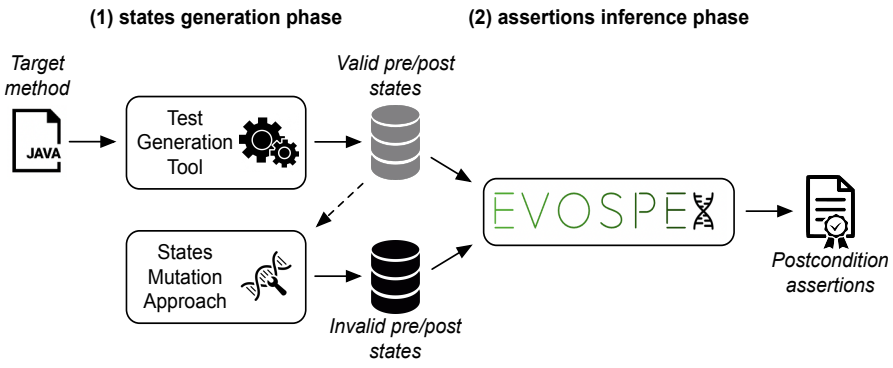


Figura 6.4: Una vista general de nuestro enfoque.

genético hacia aserciones de poscondición adecuadas. Dos tipos de pares de estados son generados: los pares *válidos*, que capturan el comportamiento actual del método que las aserciones candidatas deberían satisfacer; y los pares *inválidos*, que capturan comportamientos incorrectos (pares de estados pre/pos que no corresponden al comportamiento actual del método), que las aserciones candidatas no deberían satisfacer. Los pares pre/pos válidos son generados mediante la ejecución del método objetivo, utilizando una técnica de generación de tests; claramente, estos pares corresponden al comportamiento actual del métodos, ya que fueron generados a partir de su ejecución. Por otro lado, los pares pre/pos inválidos son generados mediante *mutaciones* de los pares válidos, generando pares por fuera del conjunto de pares válidos; a diferencia del caso de los pares válidos, no está garantizado que nuestros pares inválidos representen efectivamente comportamientos incorrectos del método, es decir, que representan comportamiento que *no* son exhibidos por el método. Esto podría afectar la precisión de las aserciones obtenidas, ya que el algoritmo estaría guiado para evitar algunos comportamientos supuestamente inválidos que en realidad son válidos. En estas situaciones, las aserciones obtenidas serían más fuertes que lo necesario, llevando a un mayor número de falsos positivos, al momento de evaluar la calidad de las aserciones. Durante el diseño de nuestra técnica, abordamos este problema de la siguiente manera. Primero, la efectividad al momento de generar pares verdaderamente inválidos depende de la exhaustividad del conjunto de pares válidos: mientras más exhaustivo sea el conjunto de pares válidos, mayores son las chances de que mutar por fuera de este conjunto lleve a compor-

tamiento verdaderamente incorrecto. En segundo lugar, como no podemos garantizar la solidez del mecanismo para la generación de pares verdaderamente inválidos, uno podría arriesgarse a favorecer aserciones incorrectas en base a asumir estados como inválidos de manera incorrecta. El primero, motiva el uso de un mecanismo de generación de tests exhaustivo acotado para la generación de pares válidos. El segundo, impulsa un tratamiento asimétrico de los pares de estado válidos e inválidos en la función de fitness, que da más relevancia a la información provista por los pares válidos. En el resto de la sección describimos en más detalle como manejamos estos problemas, así como otros detalles de nuestro algoritmo genético.

6.2.1. Generación de Estados

La fase de inferencia de nuestro algoritmo depende de un conjunto de ejecuciones válidas/inválidas del método, el cuál guía la búsqueda de las aserciones de la poscondición. Esta es una parte importante de nuestra técnica. El proceso general comienza mediante la generación de ejecuciones del método objetivo m, recolectando los estados pre/pos $\langle s, s' \rangle$ de cada ejecución, los cuáles formarán el conjunto de pares de ejecuciones válidas $\mathcal{V}$. Para generar los pares de ejecuciones inválidas $\mathcal{I}$, los pares válidos son *mutados*: dado un par válido $\langle s, s' \rangle$, mutamos s' a s'', y chequeamos que el nuevo par $\langle s, s'' \rangle$ no pertenece a V, para considerarlo parte de $\mathcal{I}$. Por supuesto, el par pre/pos mutado podría en realidad corresponder a una ejecución válida de m que no hemos cubierto en $\mathcal{V}$. La efectividad de este último mecanismo depende de cuán completo es $\mathcal{V}$ (aunque todavía podemos generar pares de ejecución válidos "no vistos" a través de mutación), lo que motiva un enfoque exhaustivo acotado para la generación de los pares válidos.

El mecanismo para generar pares de ejecuciones válidas funciona de la siguiente manera. Sean $C, C_1, \ldots, C_n$ clases, y m, el método objetivo, un método en C con parámetros de tipo $C_1, \ldots, C_n$. Los estados iniciales para la ejecución de m serán tuplas $\langle o_C, o_{C_1}, \ldots, o_{C_n} \rangle$ de objetos de tipo $C, C_1, \ldots, C_n$, respectivamente. Para formar estas tuplas, estos objetos son construidos de manera exhaustiva acotada del siguiente modo. Sea C_i una clase, y los métodos $b_1, \ldots, b_l$ con conjunto de *constructores* para C_i, es decir, un conjunto de métodos manualmente identificados que pueden ser utilizados para construir objetos de tipo C_i. Dada una cota k (máxima longitud de secuencias de métodos), construimos un conjunto de objetos de clase C_i utilizando una variante de Randoop [68]. Randoop genera de manera aleatoria secuencias de

métodos de la API de C_i, de longitud cada vez mayor, mediante un proceso iterativo en el que trazas previamente producidas son seleccionadas aleatoriamente, junto con un método para generar una nueva traza que invoca a este método. Nuestra variante incorpora dos modificaciones principales a este proceso:

- La selección aleatoria de un método para extender una traza t previamente producida (caso de test), implementada en [68], es reemplazada por un mecanismo que elige sistemáticamente *todos* los métodos en $b_1, \ldots, b_l$, llevando a l extensiones diferentes de t. Esto es aplicado hasta que se alcanza la cota k.

- Un mecanismo de *matcheo* de estados es implementado, para reducir el número de combinaciones de métodos: cuando una traza nueva lleva a un objeto que coincide con uno previamente colectado, la traza (y el objeto) son descartados. El enfoque de matcheo de estados utiliza la representación canónica de objetos presentada en [75].

Además de la cota k para la longitud de trazas, el mecanismo de matcheo de estados también requiere un número máximo de objetos por tipo, y un rango para tipos primitivos (por ejemplo, $0, \ldots, k-1$ para enteros). Este es un scope basado en el valor k, tal y como se define en los procesos de *finitization* de Korat [6] (un problema estándar en generación exhaustiva acotada). Utilizando el mecanismo mencionado, creamos las tuplas de estados iniciales, para ejecutar m. Luego, ejecutamos m en cada una de estas tuplas, y colectamos los estados pos correspondientes, construyendo de este modo el conjunto $\mathcal{V}$ de pares de estados pre/pos válidos para m.

Para producir el conjunto $\mathcal{I}$ de pares de estados pre/pos inválidos, una operación de mutación es aplicada. Esta operación, dado un par válido $\langle s, s' \rangle$ perteneciente a $\mathcal{V}$, y crea un par $\langle s, s'' \rangle$, donde s'' es resultado de mutar s' mediante la selección de un campo aleatorio en el objeto receptor o valor de retorno (que forman parte de s'), y reemplazando el valor por otro del tipo correspondiente generado de manera aleatoria dentro del scope actual. Antes de incluir el nuevo par en el conjunto $\mathcal{I}$, chequeamos que no forme parte del conjunto de pares válidos $\mathcal{V}$.

6.2.2. Poscondiciones como Cromosomas

Nuestra representación de aserciones candidatas está basada en la codificación utilizada en nuestro trabajo previo involucrando algoritmos genéticos, presentado en el Capítulo 5 y de manera más resumida en [61], donde los cromosomas representan conjunciones de aserciones (cada gen en un cromosoma representa una aserción). Es decir, dado un cromosoma c, la poscondición candidata φ_c representada por c está definida del siguiente modo:

$$c = \langle g_1, g_2, \ldots, g_n \rangle \Rightarrow \varphi_c = g_1 \wedge g_2 \wedge ... \wedge g_n$$

A diferencia de lo que es común en algoritmos genéticos, en esta representación los cromosomas tienen longitud variable (hasta una longitud máxima), y las posiciones de los genes no son tenidas en cuenta por los operadores genéticos, debido a la asociatividad y conmutatividad de la conjunción. Los genes necesitan codificar aserciones complejas; debajo describimos como estos genes son construidos, mutados y combinados.

6.2.3. Población Inicial

Describamos ahora como construimos la población inicial, para comenzar el GA. Con el objetivo de crear individuos representando poscondiciones "significativas", es decir, aserciones capturando propiedades de objetos que son alcanzables al final de las ejecuciones del método, tenemos en cuenta la información de los tipos, del mismo modo que en [61]. Esencialmente, consideramos el *grafo de tipos* construido automáticamente a partir de la clase bajo análisis: los nodos representan tipos, y por cada campo f de tipo T en una clase C, se produce un arco en el grafo yendo desde el nodo que representa a C al nodo que representa a T. Por ejemplo, si consideramos la clase `AvlTreeList` en la Figura 6.1, el grafo de tipos correspondiente sería el que se muestra en la Figura 6.5. Es sencillo observar que expresiones tipadas pueden construirse al visitar el grafo, utilizando los campos del objeto a partir del cuál el método fue ejecutado. Algunos ejemplos son `this.root`, `this.root.left`, `this.root.size`, `this.root.val`, etc. Más aún, a partir de loops en el grafo, pueden crearse expresiones denotando conjuntos, como `this.root.*left` (el conjunto de nodos alcanzable desde `this.root` mediante aplicaciones de solamente el campo `left`), `this.root.*right` y `this.root.*(left+right)` (como explicamos previamente, utilizamos `*` para la clausura reflexivo-transitiva, como en [36]). Utilizando expresiones de-

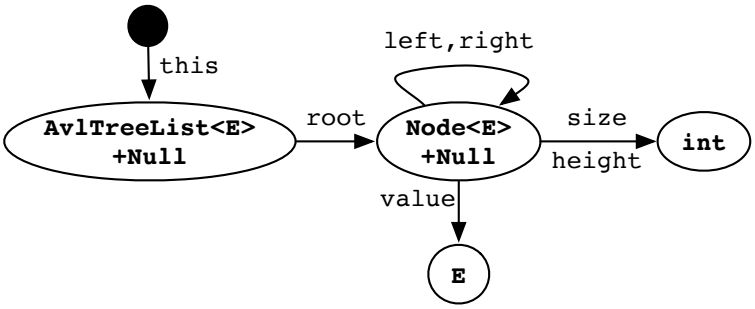


Figura 6.5: Grafo de tipos para la clase `AvlTreeList`.

notando un solo valor, y evaluándolas en un subconjunto de ejecuciones de métodos válidas (resp. inválidas) determinado aleatoriamente, construimos cromosomas de tamaño 1 del siguiente modo: si el resultado de evaluar una expresión `expr` en una tupla t válida (resp. inválida) devuelve un valor v, entonces creamos el individuo $\langle$`expr == v`$\rangle$ (resp. $\langle$`expr != v`$\rangle$).

En adición a estos estos individuos básicos, también creamos cromosomas conteniendo comparaciones de expresiones aleatorias del mismo tipo (por ejemplo, `this.root == this.root.right`), cromosomas con fórmulas cuantificadas considerando expresiones que denotan conjuntos (por ejemplo, `all n : this.root.*(left+right) - null : n == n.right`), e individuos comparando expresiones enteras con otras capturando cardinalidad de conjuntos (por ejemplo, `this.root.height == #(this.root.*right)`). Finalmente, dado que el método bajo análisis podría tener un valor de retorno o un conjunto de argumentos, también incluimos, en el conjunto de candidatos iniciales, expresiones que los comparan con otras expresiones del mismo tipo (por ejemplo, si consideramos el argumento de tipo entero `index` del método `AvlTreeList.add`, podríamos construir la expresión `index < this.root.size`). Las expresiones utilizadas para comparar con la variable resultado o los argumentos, asi como los operadores, se eligen de manera aleatoria.

Notemos que *todos* nuestros cromosomas iniciales son de tamaño 1. La principal razón de esta decisión de diseño es permitir que el algoritmo genético produzca de manera progresiva poscondiciones candidatas cada vez más complejas por medio de los operadores genéticos, que definimos más adelante. Mientras que el uso de cromosomas de tamaño 1 no es estándar en algoritmos genéticos, en nuestro caso ayuda a que el algoritmo converja de manera más

rápida a individuos más aptos.

6.2.4. Función de Fitness

Nuestra función de fitness evalúa cuán buena es una poscondición candidata, mediante la distinción del conjunto $\mathcal{V}$ de ejecuciones válidas y el conjunto $\mathcal{I}$ de ejecuciones inválidas. Para ello, antes de computar el valor de fitness de un cromosoma c dado, primero obtenemos la poscondición φ_c representada por c, y computamos los conjuntos $\mathcal{P}$ y $\mathcal{N}$ de contraejemplos positivos y negativos, respectivamente. Estos conjuntos están definidos de la siguiente manera:

$$\mathcal{P} = \{v \in V | \neg\varphi_c(v)\} \quad \mathcal{N} = \{i \in I | \varphi_c(v)\}$$

Básicamente, los conjuntos $\mathcal{P}$ y $\mathcal{N}$ contienen aquellas ejecuciones para las que la poscondición φ_c no se comporta de manera correcta. Recordemos que, a diferencia del caso del conjunto $\mathcal{V}$ que de manera confiable representa información de ejecuciones actuales de m, el conjunto $\mathcal{I}$ podría contener ejecuciones mutadas que son consideradas "inválidas", pero que en realidad corresponden a ejecuciones válidas de m. Esto motiva la definición de nuestra función de fitness que trata los conjuntos $\mathcal{P}$ y $\mathcal{N}$ de manera asimétrica. La función de fitness $f(c)$ se computa del siguiente modo:

$$f(c) = \begin{cases} \#\mathcal{P} > 0 \rightarrow (\mathsf{MAX} - \#\mathcal{P} - \#\mathcal{I}) + \left(\frac{1}{l_c + comp_c}\right) + \frac{mca_c}{l_c} \\ \#\mathcal{P} = 0 \rightarrow (\mathsf{MAX} - \#\mathcal{N}) + \left(\frac{1}{l_c + comp_c}\right) + \frac{mca_c}{l_c} \end{cases}$$

Esta definición por casos tiene como objetivo considerar los contraejemplos negativos solo cuando no hay contraejemplos positivos. De hecho, para dos candidatos arbitrarios c_1 y c_2, si c_1 no tiene contraejemplos positivos y c_2 tiene algunos contraejemplos positivos, entonces está garantizado que f producirá un peor valor de fitness para c_2, no importa cuántos contraejemplos negativos estos candidatos tengan. El motivo en este caso es darle más importancia a la información confiable provista por los contraejemplos positivos.

La definición de la función de fitness tiene 3 partes. El primer término refleja el aspecto más importante: minimizar el número de contraejemplos. El hecho de que cuando la poscondición candidata φ_c tiene contraejemplos positivos, es decir, es falsificada por alguna ejecución correcta del método,

todos los elementos del conjunto $\mathcal{I}$ de ejecuciones inválidas son considerados como contraejemplos, es lo que garantiza la observación de más arriba en relación a la priorización de candidatos sin contraejemplos positivos. Más precisamente, el primer término de la función sustrae $\#\mathcal{I}$ cuando $\#\mathcal{P} > 0$, para asegurar que el valor de fitness de tal individuo es menor que el valor de fitness de cualquier otro individuo que solo tiene contraejemplos negativos (es decir, $\#\mathcal{P} = 0$). El segundo término de la función de fitness actúa como una penalidad en relación a dos aspectos: la longitud l_c del candidato y la "complejidad" $comp_c$ del candidato. La longitud del candidato es simplemente el número de conyuntos en la aserción, y es considerada para guiar al algoritmo hacia la producción de poscondiciones más concisas. La complejidad del candidato es la suma de la complejidad de cada conyunto. Intuitivamente, la complejidad de una igualdad entre dos campos enteros es menor que la complejidad de una igualdad entre un campo entero y la cardinalidad de un conjunto, y ambas son menores que la complejidad de una fórmula cuantificada, y así sucesivamente. El último término de la función actúa como una recompensa favoreciendo aquellos candidatos con un mayor número de "method component assertions" mca_c, es decir, con un alto número de conyuntos de la poscondición candidata que representan propiedades en relación a componentes del método como parámetros, el resultado, o una relación entre estados iniciales y finales de un objeto. Como describimos, la penalidad relacionada a la longitud y la complejidad del candidato, así como la recompensa priorizando aserciones de componentes del método solo contribuyen una fracción al valor de fitness, ya que queremos que el algoritmo se enfoque en individuos cuyo número de contraejemplos se aproxima a 0.

6.2.5. Operadores Genéticos

Durante la evolución, los operadores genéticos permiten que el algoritmo explore el espacio de búsqueda de soluciones candidatas, realizando operaciones que producen individuos con nuevas características así como combinaciones de individuos existentes. En particular, nuestro algoritmo implementa dos operadores genéticos conocidos, *mutación* y *crossover*. Algunos de estos operadores fueron inspirados por algunos similares introducidos en [61], mientras que otros son nuevos. Además, implementamos un operador de *selección* que mantiene en la población aquellos candidatos que son más aptos para ser parte de la poscondición real.

Mutación

Cada gen de un cromosoma es elegido para mutación con una probabilidad de 0,3, y la operación puede realizar una variedad de modificaciones dependiendo del tipo de expresión del gen elegido. Desde un punto de vista general, el conjunto de mutaciones consideradas es el siguiente:

Eliminación de un gen. Puede ser aplicada a cualquier gen, y simplemente elimina el gen del cromosoma.

Negación. Niega la expresión del gen y es aplicada a cualquier gen excepto a aserciones cuantificadas.

Suma/resta numérica. Es aplicada a genes que comparan dos expresiones numéricas, y suma/resta una expresión numérica aleatoriamente elegida al lado derecho de la comparación.

Reemplazo de una expresión. Se aplica a cualquier gen, y reemplaza alguna parte del gen con una expresión del mismo tipo aleatoriamente seleccionada.

Extensión de una expresión. Puede ser aplicada a cualquier gen que involucre una expresión navigacional, y extiende esta expresión con un nuevo campo, por ejemplo reemplazando `this.root` por `this.root.left`.

Reemplazo de un operador: reemplazada un operador por otro alternativo. Los operadores pueden variar dependiendo de la expresión actual del gen. Por ejemplo, para igualdades relacionales, los posibles operadores son $\{==, != \}$; para comparaciones numéricas los operadores son $\{==, !=, <, >, <=, >=\}$; y para expresiones cuantificadas, los cuantificadores son $\{all, some\}$.

Crossover

Para producir combinaciones de individuos, utilizamos un ratio de crossover de 0,35. Dados dos cromosomas c_1 y c_2 elegidos aleatoriamente, nuestro operador de *crossover* simplemente produce un nuevo individuo conteniendo la unión de los genes de c_1 y c_2, representando la poscondición candidata $\varphi_{c1} \wedge \varphi_{c2}$. Un detalle importante de nuestro operador de *crossover* es que antes de elegir los individuos a combinar, filtramos la población, manteniendo individuos que solo tienen contraejemplos negativos, es decir, que representan fórmulas que son consistentes con todas las ejecuciones válidas. La principal razón de esta política es que queremos que el algoritmo una cromosomas que ya son consistentes con las ejecuciones válidas del método.

Selección

Finalmente, para mantener en la población los *mejores* candidatos de cada generación, nuestro operador de *selección* está definido de la siguiente manera: dado un número n a ser utilizado como tamaño de población constante, nuestro operador primero ordena todas las poscondiciones candidatas por su valor de fitness en orden decreciente, y luego los candidatos que se pasan a la próxima generación son los primeros $n/2$ más $n/2$ individuos unarios elegidos aleatoriamente de los cromosomas de tamaño 1. En algunos casos, el gen será una fórmula que aún tiene contraejemplos positivos, mientras que en otros el candidato será unario y válido, solo tendrá contraejemplos negativos. Esta última política en nuestro operador de selección nos permite mantener en la población propiedades válidas descubiertas que el algoritmo puede usar en futuras operaciones de crossover.

6.3. Evaluación

Para evaluar nuestra técnica de inferencia de poscondiciones, realizamos experimentos enfocados en las siguientes preguntas de investigación:

RQ1 *Las poscondiciones producidas por EvoSpex, ¿tienen alguna deficiencia comparadas con las producidas por técnicas relacionadas?*

RQ2 *Las aserciones producidas por el algoritmo, ¿son cercanas a contratos escritos manualmente?*

6.3.1. Calidad de las Poscondiciones

Para evaluar la RQ1, necesitamos considerar programas (en nuestro caso, programas Java) a partir de los cuáles inferir poscondiciones de métodos. Como mencionamos anteriormente, y como es claro a partir del espacio de búsqueda de aserciones y de los operadores definidos, nos enfocamos en clases y métodos con implementaciones basadas en referencias, en particular clases donde la representación interna correspondiente tiene invariantes (implícitos) fuertes. Como fuente para nuestros casos de estudio, consideramos SF110[2] (originalmente utilizado en [16]), una colección de 110 proyectos Java (100 proyectos aleatorios, y los 10 más populares de acuerdo a SourceForge), que

[2]https://www.evosuite.org/experimental-data/sf110/

cubre una gran variedad de software, representativo del desarrollo open source. Nuestro proceso para evaluar las deficiencias de las aserciones de las poscondiciones utiliza la herramienta OASIs [37], esencialmente, para evaluar la calidad de una poscondición en términos de su número de falsos positivos y falsos negativos. El proceso de computar estos números requiere un proceso manual (como se describe en [37], para computar los falsos negativos se necesita primero eliminar los falsos positivos, lo que implica tener que refinar manualmente las poscondiciones producidas cada vez que OASIs reporta la presencia de un falso positivo). Por lo tanto, no podemos considerar los 110 proyectos en nuestra evaluación. Elegimos aleatoriamente 16 proyectos, salteando los casos que tienen alguna dependencia en el entorno (nuestra técnica involucra generación automática de tests, y dependencias del entorno afectan seriamente a estas herramientas). Los 16 proyectos pueden encontrarse en el Cuadro 6.2. Para cada caso de estudio, elegimos varios métodos con diferentes comportamientos para análisis, definimos manualmente un conjunto definido de constructores (en caso de ser necesario), y generamos las ejecuciones de métodos válidas e inválidas correspondientes con un scope pequeño (3 en todos los casos). Luego, ejecutamos nuestra herramienta de la siguiente manera: para cada método m elegido para análisis, ejecutamos EvoSpex para producir una poscondición φ_m para m hasta que el algoritmo llegaba a las 30 generaciones o completaba un timeout de 10 minutos. Repetimos esta ejecución 10 veces, y elegimos la poscondición que se repitió el mayor número de veces, a partir de las 10 producidas. Adicionalmente, para comparar nuestra herramienta con técnicas relacionadas, ejecutamos Daikon para inferir poscondiciones para cada método m. Es importante remarcar que las test suites que proveímos a Daikon para producir poscondiciones para los métodos bajo análisis, fueron exactamente las mismas suites que fueron utilizadas para generar las ejecuciones válidas del método en nuestra técnica (nuestras suites exhaustivas acotadas). Tanto nuestra herramienta como Daikon pueden producir aserciones que llevan a falsos positivos, así como también aserciones redundantes.

Los resultados de este experimento se muestran en los Cuadros 6.1 y 6.2. El Cuadro 6.1 presenta las poscondiciones generadas por las herramientas, luego de remover los falsos positivos y las aserciones redundantes, con el objetivo de dar una mirada clara de la complejidad de las aserciones que cada técnica es capaz de generar. A partir de estas aserciones, medimos la calidad de las poscondiciones computando automáticamente los falsos positivos y falsos negativos, utilizando la herramienta OASIs [37]. El Cuadro 6.2

Cuadro 6.1: Poscondiciones generadas por EvoSpex y Daikon.

EvoSpex	Daikon
jiprof - ClassAllocation.getAllocCount() : int	
`result == this._count`	`result == this._count`
	`result == old(this._count)`
jiprof - ClassAllocation.incAllocCount() : void	
`this._count == 1 + old(this._count)`	`this._count - old(this._count) - 1 == 0`
	`this._count >= 1`
jmca - JParser.SimpleNode.jjtSetParent(Node n) : void	
`n == this.parent`	`this.parent == old(n)`
	`this.children == old(this.children)`
	`this.id == old(this.id)`
	`this.parser == old(this.parser)`
	`this.identifiers == old(this.identifiers)`
bpmail - EmailFacadeState.setState(int ID, boolean dirtyFlag) : void	
`ID in this.states.keySet()`	`this.states == old(this.states)`
byuic - JavaScriptIdentifier.preventMunging() : void	
`mungedValue == old(mungedValue)`	`mungedValue == old(mungedValue)`
`refCount == old(refCount)`	`refCount == old(refCount)`
`all n : declaredScope.*parentScope |`	`declaredScope == old(declaredScope)`
`n !n n.^parentScope`	`!this.markedForMunging`
dom4j - LazyList.add(E elem) : boolean	
`old(this.size) == this.size - 1`	`header == old(header)`
`result == true`	`this.size >= 1`
`element in this.header.*next.element`	`result == true`
	`this.size - old(this.size) - 1 == 0`

muestra los resultados de esta evaluación de calidad. Específicamente, para cada caso de estudio, reportamos: *(i)* las líneas de código (LoC) del proyecto evaluado; *(ii)* el número de métodos analizados del proyecto correspondiente; *(iii)* el número de aserciones producidas como parte de las poscondiciones; *(iv)* la cantidad y porcentaje de falsos positivos presentes en las aserciones generadas; y *(v)* número de métodos para los cuáles fueron detectados falsos negativos. Notemos que, como se propone en [37], la detección de falsos negativos es realizada una vez que todos los falsos positivos han sido eliminados de la poscondición (de ahí la tarea manual que nos hizo considerar un subconjunto de SF110). Tanto para la detección de falsos positivos como la detección de falsos negativos, ejecutamos OASIs con un timeout de un minuto. Algunos problemas con OASIs nos impidieron reportar el número de falsos

Cuadro 6.2: Utilizando OASIs para medir la calidad de las poscondiciones inferidas por Daikon y EvoSpex.

Proyecto	**LOCs**	**Métodos**	**Técnica**	**Aserciones**	**FPs**		**FNs**
					Total	%	
imsmart	1407	3	Daikon	21	2	9.52	1
			EvoSpex	4	1	25	1
beanbin	4784	5	Daikon	35	5	14.29	0
			EvoSpex	7	0	0	0
byuic	7699	7	Daikon	165	21	12.73	4
			EvoSpex	36	4	11.11	2
geo-google	20974	7	Daikon	93	30	32.26	0
			EvoSpex	10	3	30	4
templateit	3315	7	Daikon	37	4	10.81	3
			EvoSpex	20	0	0	2
water-simulator	9931	9	Daikon	39	3	7.69	9
			EvoSpex	18	3	16.67	9
dsachat	5546	9	Daikon	138	15	10.87	3
			EvoSpex	18	2	11.11	2
jmca	16891	9	Daikon	205	26	12.68	0
			EvoSpex	25	1	4	3
jni-inchi	3100	10	Daikon	122	12	9.84	2
			EvoSpex	50	1	2	4
bpmail	2765	11	Daikon	46	6	13.04	8
			EvoSpex	17	0	0	7
dom4j	42198	18	Daikon	166	27	16.27	7
			EvoSpex	25	2	8	10
jdbacl	28618	19	Daikon	115	17	14.78	10
			EvoSpex	80	3	3.75	8
jiprof	26296	20	Daikon	352	81	23.01	20
			EvoSpex	35	4	11.43	19
summa	119963	21	Daikon	273	67	24.54	6
			EvoSpex	62	5	8.06	5
corina	78144	22	Daikon	155	13	8.39	17
			EvoSpex	55	1	1.82	17
a4j	6618	23	Daikon	257	59	22.96	9
			EvoSpex	60	5	8.33	5
TOTAL		200	Daikon	2219	388	17.49	99
			EvoSpex	522	35	**6.70**	**98**

negativos para cada método y caso de estudio; más precisamente, cuando la herramienta reportó la existencia de falsos negativos, en algunos casos fue

incapaz de producir contraejemplos testigos (casos de test), impidiéndonos medir el número de falsos negativos identificados por la herramienta en estos casos. Este problema fue discutido con los desarrolladores de la herramienta. Por lo tanto, decidimos informar el número de métodos para los que OASIs reportó la existencia de falsos negativos, en lugar de el número de falsos negativos encontrado, ya que esta información no se produjo de manera confiable para todos los casos. Por ejemplo, para el proyecto `imsmart`, de los 3 métodos analizados, OASIs encontró que una de las aserciones descubiertas por Daikon tenía falsos negativos, y que también una de las aserciones descubiertas por EvoSpex tenía falsos negativos.

6.3.2. Reproducción de Contratos

La evaluación de la RQ2 requiere clases con métodos equipados con contratos escritos manualmente. Más aún, como mencionamos previamente, las aserciones para *runtime checking* son aserciones típicamente débiles, que pueden ser chequeadas eficientemente, y que capturan de manera muy débil la semántica de las clases y métodos correspondientes [57, 72]. Para comparar con contratos más fuertes, tomamos:

- Un conjunto de casos de estudio con contratos escritos para verificación de programas orientados a objeto. Más precisamente, estos programas están escritos en Eiffel [58], y los contratos se usaron para verificación utilizando la herramienta AutoProof [89], un verificador para programas orientados a objeto escritos en el lenguaje de programación Eiffel, para programas Eiffel.

- Un conjunto de casos de estudio para los que representaciones de datos e implementaciones de métodos fueron automáticamente sintetizados a partir de una especificación de alto nivel. Más precisamente, las implementaciones sintetizadas son tomadas de [49], generadas por la herramienta Cozy[3], y son garantizadas correctas con respecto a la especificación de alto nivel, las cuáles sirven como contratos de métodos.

De [89], consideramos específicamente varios métodos y sus poscondiciones correspondientes, para los siguientes casos:

[3] `https://github.com/CozySynthesizer/cozy`

- `Composite`: un árbol con una restricción de consistencia entre el padre y los nodos hijos. Cada nodo mantiene una colección de hijos y una referencia a su padre; el cliente puede modificar cualquier nodo intermedio. El valor en cada nodo debería ser el máximo de todos los valores de sus hijos.

- `DoublyLinkedListNode`: nodo en una doubly linked list (circular) con un invariante estructural obligando a que sus enlaces izquierdo y derecho sean consistentes con sus vecinos.

- `Map<K,V>`: implementación del tipo abstracto de datos `Map`, basada en dos listas que contienen las claves y los valores, y con operaciones que realizan búsquedas lineales en las listas.

- `RingBuffer<G>`: cola acotada implementada sobre un arreglo circular.

Dado que nuestra herramienta es para Java, y que estas implementaciones están en Eiffel, realizamos una traducción manual de todas las clases a Java, para poder analizarlas con EvoSpex (esto también nos impidió considerar casos de estudio más sofisticados). Mientras que la traducción fue manual, nos esforzamos en hacerla de modo sistemático, preservando la estructura del código original, y teniendo en cuenta la semántica de las referencias (por ejemplo, tipos expandidos en Eiffel), indexación de arreglos en Eiffel vs. Java, etc., utilizando como guía J2Eif [88]. Eiffel también difiere de Java en otros aspectos importantes que no afectaron la traducción (por ejemplo, herencia, visibilidad de atributos, etc.). Aunque no verificamos formalmente nuestras traducciones, las mismas fueron revisadas de manera independiente por coautores de este trabajo.

De [49], consideramos varias especificaciones de alto nivel y sus implementaciones Java sintetizadas correspondientes:

- `Polyupdate`: bag de elementos que mantiene la suma de sus elementos positivos.

- `Structure`: clase que encapsula una función y almacena un valor.

- `ListComp02`: estructura compuesta de dos colecciones de elementos diferentes, y operaciones que combinan elementos de las colecciones.

- `MinFinder`: bag de elementos con una operación para obtener el mínimo.

Cuadro 6.3: Comparando contratos escritos manualmente (en clases Eiffel verificadas) con poscondiciones inferidas con EvoSpex.

Método	**Contratos Eiffel**	**EvoSpex**
Composite		
add_child(Composite c) : void	child added	✓
	c value unchanged	
	c children unchanged	
	ancestors unchanged	✓
DoublyLinkedListNode		
insert_right(DoublyLinkedListNode n) : void	n left set	✓
	n right set	
remove() : void	singleton	✓
	neighbors connected	
Map<K,V>		
count() : int	result is size	✓
extend(K k,V v) : int	key set	✓
	data set	✓
	other keys unchanged	
	other data unchanged	
	result is index	
remove(K k) : int	key removed	✓
	other keys unchanged	
	other data unchanged	
	result is index	
RingBuffer<G>		
count() : int	result is size	
extend(G a_value)	value added	✓
item() : G	result is first elem	
remove() : void	first removed	
wipe_out() : void	is empty	✓

- `MaxBag`: conjunto de elementos, con una operación para obtener el máximo.

Para inferir poscondiciones para los métodos en estas clases, primero generamos las ejecuciones válidas e inválidas de los métodos, como describimos anteriormente, para cada método objetivo utilizando un scope de 4. Luego, ejecutamos nuestro algoritmo utilizando la misma configuración que para la RQ1 (30 generaciones con un timeout de 10 minutos). De nuevo, repetimos esta ejecución 10 veces y elegimos la poscondición que se repetía el mayor número de veces. Notemos que nuestro enfoque no está utilizando los contratos reales que ya acompañaban a estos métodos. Estos son ignorados comple-

Cuadro 6.4: Infiriendo poscondiciones de colecciones sintetizadas.

Estado de alto nivel	Método	Spec de alto nivel	EvoSpex
	Polyupdate		
x : Bag<Int>	a(Integer y) : void	y added to x	✓
s : Int		y added to s if positive	
	sm() : Integer	result is s + sum of x	✓
	Structure		
x : Int	foo() : Integer	result is x + 1	✓
	setX(Integer y)	x = y	✓
	ListComp02		
Rs : Bag<R>	insert_r(R r) : void	r added to Rs	✓
Ss : Bag<S>	insert_s(S s) : void	s added to Ss	✓
	q() : Integer	result is the sum of Rs ⊗ Ss	
	MinFinder		
xs : Bag<T>	findmin() : T	result is min of xs	✓
	chval(T x, int nv) : void	inner value of T is x	
	MaxBag		
l : Set<Int>	get_max() : Integer	result is max of l	✓
	add(Integer x) : void	x added to l	✓
	remove(Integer x) : void	x removed from l	

tamente durante el proceso de inferencia, y solo consideramos el código de los métodos, tanto para la generación de ejecuciones de método válidas/inválidas como para la inferencia evolutiva. Computamos las poscondiciones para las implementaciones Java, y luego las comparamos con las especificaciones originales de alto nivel, a partir de las que las implementaciones Java fueron sintetizadas.

Los resultados de este experimento se muestran en los Cuadros 6.3 y 6.4. En el Cuadro 6.3, para cada uno de los métodos analizados, la columna Contratos Eiffel lista las propiedades que estaban presentes en la poscondición original (expresadas como texto, para una referencia más sencilla). En el Cuadro 6.4, la poscondición original está descrita en la columna Spec de alto nivel en términos del estado abstracto declarado en la especificación. En ambas tablas, la columna EvoSpex indica cuáles de las aserciones correspondientes en el contrato original fueron producidas por nuestro algoritmo. Finalmente, el Cuadro 6.5 resume estos resultados y también reporta el número de aserciones *inválidas* sintetizadas como parte de las especificaciones inferidas para cada caso de estudio de Eiffel y Cozy.

Todos los experimentos fueron ejecutados en una estación de trabajo con un procesador Intel Core i7 3.2Ghz, 16Gb de RAM, ejecutando GNU/Linux

Cuadro 6.5: Resumen de las aserciones producidas por EvoSpex en los casos de estudio de la RQ2.

Caso de Estudio	**Métodos**	**Contratos manuales**	**Aserciones Inferidas**	
			Total	Inválidas
		Eiffel		
Composite	1	4	7	0
DoublyLinkedListNode	2	4	5	0
Map<K,V>	3	10	10	0
RingBuffer<G>	5	5	30	4
		Cozy		
Polyupdate	2	3	3	0
Structure	2	2	2	0
ListComp02	3	3	6	0
MinFinder	2	2	3	0
MaxBag	3	3	33	5

(Ubuntu 16.04). Nuestra herramienta, todos los casos de estudio, y una descripción sobre como reproducir los experimentos presentados en esta sección pueden encontrarse en el siguiente sitio:

`https://sites.google.com/view/evospex`

6.3.3. Discusión

Discutamos ahora los resultados de nuestra evaluación. En relación a la RQ1, los resultados muestran que nuestro enfoque es capaz de generar poscondiciones conteniendo aserciones más complejas que las producidas por Daikon. Esto se debe principalmente al hecho de que nuestra técnica se enfoca en evolucionar aserciones apuntando a condiciones basadas en referencias de implementaciones también basadas en referencias, a diferencia de Daikon cuyas expresiones son comparativamente propiedades más simples, que no incluyen restricciones estructurales complejas, chequeos de pertenencia, etc. (con la excepción de arreglos e implementaciones de `java.util.List`, caso en el que Daikon genera propiedades estructurales interesantes). Además, como muestra el Cuadro 6.2, un número significativo de aserciones inferidas por nuestra técnica, son *verdaderos positivos*, es decir, aserciones que valen para todos los estados pos de los métodos correspondientes, para cualquier

scope. Por supuesto, este chequeo de verdaderos positivos es al final manual (analizamos cuidadosamente como opera cada uno de los métodos evaluados, e inspeccionamos las aserciones obtenidas luego de filtrar los conyuntos en base a la evaluación con OASIs); el análisis de deficiencias realizado por OASIs es inherentemente incompleto, no podemos garantizar la veracidad de las aserciones que quedan.

Como se muestra en el Cuadro 6.2, en la mayoría de los casos de estudio (13 de 16), el porcentaje de falsos positivos que genera EvoSpex, cuando consideramos la cantidad total de aserciones producidas, es menor que el producido por Daikon. Por lo tanto, en comparación con Daikon, y considerando los falsos positivos, nuestras aserciones son significativamente más precisas. De hecho, en un total de 200 métodos analizados, nuestra técnica tuvo un 6.7 % de falsos positivos, comparado con el 17.49 % de Daikon (un orden de magnitud de mejora). Más aún, la relación entre el número de aserciones producidas (en total, 522 de EvoSpex vs las 2219 producidas por Daikon) y la presencia de falsos negativos, muestra que nuestra técnica produce aserciones de fortaleza similar, con significativamente menos restricciones. Daikon parece hacer mucho uso de valores específicos observados en las ejecuciones, llevando a aserciones que, mientras que son verdaderas considerando la test suite provista, son inválidas cuando se consideran scopes más grandes. Nuestro algoritmo está guiado tanto por estados pre/pos válidos como inválidos, lo que le brinda una ventaja sobre Daikon, y explora el espacio de búsqueda de aserciones candidatas que son menos afectadas por valores específicos observados en las ejecuciones. En relación a los falsos negativos, tanto Daikon como nuestra técnica conducen a aserciones para las que OASIs es capaz de identificar falsos negativos (teniendo nuestra técnica una pequeña ventaja en ese aspecto). La conclusión es clara: las aserciones obtenidas por ambas herramientas son más débiles que la poscondición "más fuerte", no detectando entonces algunas mutaciones de los métodos analizados (observar como OASIs identifica los falsos negativos [37]). Desafortunadamente, como discutimos previamente, un problema con OASIs nos impidió realizar una comparación más detallada, basada en el *número* de falsos negativos identificados en cada caso. Sin embargo, mirando las poscondiciones obtenidas, como se muestra en el Cuadro 6.1, es evidente que nuestra técnica produce aserciones más fuertes.

Con respecto a la RQ2, las aserciones que nuestra técnica puede producir son cercanas a aquellas que podrían ser definidas por los desarrolladores al momento de especificar contratos ricos. Cómo muestra el Cuadro 6.3, nues-

tro algoritmo generó de manera exacta al menos una propiedad definida en las aserciones originales para los métodos de Eiffel en 8 de 11 casos. Similarmente, como indica el Cuadro 6.4, en 9 de los 12 métodos identificamos correctamente al menos una propiedad de las poscondiciones correspondientes. Esto confirma que nuestra técnica es capaz de generar aserciones que son verdaderamente positivas y son independientes del scope. En el caso de las propiedades restantes que el algoritmo no pudo inferir, son aserciones específicas en relación a los argumentos, propiedades complejas sobre conjuntos que están más allá de las aserciones que el algoritmo podría producir, como la propiedad "other keys unchanged" en la rutina `Map.extend`, o son restricciones aritméticas relativamente complejas como las que se presentan en los métodos de `ListComp02` y `RingBuffer` en los casos de Cozy (notemos que nuestras aserciones se concentran en propiedades de campos basados en referencia en lugar de aserciones aritméticas complejas). En relación a la precisión de las especificaciones generadas para los casos de estudio de Eiffel y Cozy, es también importante analizar si la herramienta produce aserciones inválidas. Como muestra el Cuadro 6.5, solo 2 de los 9 casos contienen aserciones inválidas en las poscondiciones inferidas correspondientes, siendo todas aserciones que son verdaderas en los escenarios acotados a partir del que fueron computadas, pero no lo son si el scope es extendido. Estos casos fueron los que involucraron un número mayor de campos. Aún así, el porcentaje de poscondiciones inválidas para estos casos fue de alrededor del 15 % o menos (4 de 30 en un caso, 5 de 33 en el otro). El Cuadro 6.5 también muestra que, para la mayoría de los casos de estudio, EvoSpex produce aserciones válidas adicionales, en comparación con la poscondición correspondiente. Generalmente, esto tiene que ver con la información válida que no está explícitamente mencionada en la poscondición original. Por ejemplo, para `Composite.addChild`, EvoSpex produjo una poscondición de 7 conyuntos, 2 de los cuáles están en el contrato manual; los 5 restantes son o bien triviales (por ejemplo, la lista de hijos no ha cambiado, el padre no ha cambiado), o capturan información válida que no estaba presente en el "ensure" original (por ejemplo, aciclicidad de la estructura padre). Para más detalles, referimos al lector al sitio acompañando a este trabajo, donde pueden encontrarse todas las aserciones producidas para caso de estudio.

6.4. Conclusión

El problema del oráculo se ha vuelto un problema muy importante en ingeniería de software, y en este contexto, la síntesis o inferencia de especificaciones es particularmente desafiante [4]. En este trabajo, proponemos un algoritmo evolutivo para la inferencia de especificaciones, en particular para inferir aserciones de métodos en forma de *poscondiciones*. Nuestra técnica presenta varias características novedosas, incluyendo un mecanismo para generar tests de manera exhaustiva acotada, a partir de una API, y la definición de un algoritmo genético cuyo espacio de búsqueda de aserciones candidatas incluye restricciones complejas involucrando los parámetros del método, valor de retorno, estados internos de los objetos, y la relación entre estados pre y pos de la ejecución del método. Nuestra evaluación experimental muestra que nuestra herramienta es capaz de producir aserciones más precisas (contratos más fuertes en el sentido de [73], con los beneficios asociados descritos), con un total de 6.70 % de falsos positivos, comparados con el 17.49 % de falsos positivos de técnicas relacionadas, para un conjunto de métodos elegidos aleatoriamente a partir de un conjunto de proyectos Java open source. Además, nuestra evaluación muestra que nuestra técnica es capaz de inferir una parte importante de aserciones ricas de programas, tomadas a partir de un conjunto de casos de estudio involucrando contratos para verificación y síntesis de programas.

Este trabajo también abre varias líneas de trabajo futuro. Por un lado, nuestro algoritmo genético usa un conjunto finito de operadores genéticos, en particular los utilizados para mutación; extender este conjunto de operadores y explorar algunos nuevos podría ser necesario para incrementar el tipo de propiedades que el algoritmo podía producir, especialmente al momento de lidiar con programas más sofisticados. Las funciones de fitness de los algoritmos genéticos juegan un papel crucial en la calidad de las soluciones; adaptar la función de fitness de nuestro algoritmo para priorizar aspectos generales de las poscondiciones de métodos podría mejorar considerablemente nuestros resultados. Nuestros experimentos estuvieron basados en el uso de una variante de generación aleatoria para la producción de suites exhaustivas acotadas. Utilizar enfoques de generación de test suites alternativos como generación completamente aleatoria podría permitirnos producir poscondiciones diferentes. La existencia de falsos negativos en nuestras aserciones también abre líneas de mejoras para nuestro mecanismo de inferencia.

Capítulo 7

Inferencia de Especificaciones basada en Fuzzing

Este capítulo presenta nuestra última técnica para la inferencia de especificaciones, que será publicada próximamente en [62]. Este enfoque, se diferencia de los anteriores particularmente en que se utiliza *fuzzing* como mecanismo de exploración de especificaciones candidatas de una clase dada. Más precisamente, SpecFuzzer, como llamamos a nuestra técnica, hace las siguientes contribuciones:

- Una nueva técnica para la inferencia de aserciones que combina fuzzing basado en gramáticas y detección dinámica de invariantes.
- Un mecanismo eficiente para la detección de aserciones similares y para ranking de aserciones basado en su fortaleza.
- Una evaluación completa de nuestra técnica, en comparación con las técnicas relacionadas GAssert y EvoSpex, donde métricas de performance son computadas en relación a aserciones de *ground truth* escritas manualmente.

El resto del capítulo se organiza del siguiente modo. La Sección 7.1 introduce dos ejemplos ilustrativos donde se muestra el desempeño de las técnicas existentes y como SpecFuzzer supera algunas limitaciones. La Sección 7.2 presenta todos los detalles técnicos de SpecFuzzer, desde la generación de aserciones candidatas basada en fuzzing hasta la inferencia y detección de invariantes mediante detección dinámica. La Sección 7.3 describe nuestra

```java
/* Returns the minimum of two integers */
public static int min(int x, int y) {
  if (x <= y) return x;
  else return y;
}
```

Figura 7.1: Método que obtiene el mínimo de dos valores enteros.

evaluación experimental que compara la performance de SpecFuzzer con las técnicas existentes para inferir aserciones de un *ground truth*. Finalmente, en la Sección 7.4 discutimos las conclusiones y líneas de trabajo futuras.

7.1. Ejemplos Ilustrativos

Esta sección introduce dos ejemplos simples con el propósito de (1) resaltar las limitaciones del estado del arte con respecto a técnicas de inferencia de especificaciones, y (2) de ilustrar los principales aspectos de SpecFuzzer.

Ejemplos

La Figura 7.1 muestra `min`, un método Java que computa el mínimo de dos enteros, mientras que la Figura 7.2 muestra `SortedList`, una clase Java que implementa una lista ordenada de números enteros. El método `min` es sencillo. La clase `SortedList` es ligeramente más elaborada. Tiene dos campos de instancias, `elem` y `next`, que representan el valor de un nodo de la lista y una referencia al siguiente nodo, respectivamente. También tiene un campo de clase (`SENTINEL`) que almacena un valor especial –el máximo entero posible en Java– como una marca para el final de la lista. El centinela debería ser ubicado al final de la lisa y no debería estar repetido. El constructor por defecto crea un nodo marcando el final de la lista. El método `insert` toma como parámetro un entero `data` y lo inserta de manera ordenada en la lista enlazada. Dado que no es posible que un elemento dado sea mayor al centinela, la búsqueda garantiza que el elemento será insertado antes que el nodo centinela.

```
public class SortedList {
  private int elem;
  private SortedList next;
  private static final int SENTINEL = Integer.MAX_VALUE;

  /* Constructors */
  public SortedList() { this(SENTINEL, null); }
  private SortedList(int elem, SortedList next) {
    this.elem = elem;
    this.next = next;
  }

  /* Method to insert an element in the list */
  void insert(int data) {
    if (data > elem) {
      next.insert(data);
    } else {
      next = new SortedList(elem, next);
      elem = data;
    }
  } }
```

Figura 7.2: Clase `SortedList` implementando una lista ordenada de enteros.

Propiedades Relevantes

El comportamiento esperado del método `min` es que compute el mínimo entre `x` e `y`. Una especificación de la poscondición es la siguiente:

```
(result == x || result == y) && (result <= x) && (result <= y)
```

La poscondición del método `SortedList.insert` involucra varias propiedades: la lista es acíclica y ordenada de manera creciente, el nodo centinela está en la lista (al final), y el elemento `data` es efectivamente insertado. Utilizando la notación JML [11], esta poscondición puede ser especificada del siguiente modo:

```
all SortedList l: reach(this, next).has(l) ==> !reach(l.next, next).has(l)
all SortedList l: reach(this, next).has(l) ==> l.elem <= l.next.elem
exists SortedList l: reach(this, next).has(l) && l.elem == SENTINEL
exists SortedList l: reach(this, next).has(l) && l.elem == data
```

Podríamos considerar estas aserciones como la especificación *ground truth* de los métodos correspondientes, y lo que idealmente esperaríamos que herramientas de inferencia de especificaciones produzcan para estos casos.

7.1.1. Técnicas para Inferencia de Especificaciones

Daikon

Daikon [13] es una técnica dinámica que infiere especificaciones mediante el monitoreo de ejecuciones de tests. Además de un programa $\mathcal{P}$, para inferir especificaciones Daikon require una test suite $\mathcal{T}$ para $\mathcal{P}$. Daikon utiliza $\mathcal{T}$ para ejercitar $\mathcal{P}$; monitorea estados de programa en varios puntos del programa $\mathcal{P}$; considera un conjunto de aserciones obtenidos mediante la instanciación de patrones de aserciones, y aquellas que no son invalidadas por ningún test en un punto del programa dado son reportadas al usuario como *invariants probables* en el punto en cuestión.

GAssert and EvoSpex

GAssert [85] y EvoSpex [64], uno de nuestros trabajos previos (descrito en el Capítulo 6), son técnicas de inferencia de especificaciones más recientes. Del mismo modo que Daikon, estas herramientas ejecutan una test suite $\mathcal{T}$ del programa $\mathcal{P}$ bajo análisis y observan ejecuciones para *inferir* especificaciones que son consistentes con las observaciones. Mientras que Daikon require que la test suite $\mathcal{T}$ sea provista, GAssert y EvoSpex utilizan sus propios mecanismos de generación de tests (herramientas de generación de tests de terceros en el caso de GAssert, un enfoque de generación de tests personalizado en el caso de EvoSpex). Aunque ambas técnicas están basadas en búsqueda evolutiva, tienen algunas diferencias claves. GAssert implementa un algoritmo co-evolutivo que explora el espacio de aserciones posibles (la co-evolución lidia con falsos positivos y falsos negativos a través de procesos evolutivos que cooperan) y utiliza la herramienta de evaluación de oráculos OASIs [37] para mejorar las aserciones de manera iterativa. EvoSpex implementa un algoritmo genético clásico para explorar el espacio de búsqueda, y usa una técnica de mutación de estados para generar estados de poscondición en los que la aserción siendo buscada debería fallar. Las operaciones evolutivas de GAssert se enfocan en producir principalmente aserciones aritméticas y lógicas, mientras que las de EvoSpex se enfocan en propiedades navigacio-

Cuadro 7.1: Desempeño de GAssert y EvoSpex en los ejemplos.

	GAssert	EvoSpex
	min(int x,int y) - postcondition	
1	`(x > r && y == r) || (r <= y && r == x)`	`r <= x`
2		`r <= y`
	SortedList.insert(int data) - postcondition	
1	`elem-(data-elem) <= old(elem)`	`exists SortedList l: reach(this, next).has(l) && l.elem == data`
2		`old(elem) <= next.elem`

nales de objetos. Para estas herramientas, cambiar los lenguajes de aserción implica la re-definición de los operadores evolutivos correspondientes y otros parámetros de los algoritmos evolutivos, lo que no es trivial.

Los Cuadros 7.1 y 7.2 muestran como Daikon, GAssert y EvoSpex se comportan en los ejemplos (por simplicidad, eliminamos `this` de las expresiones no cuantificadas en el ejemplo `insert`). GAssert se comporta de manera perfecta en el ejemplo `min`, pero de manera muy débil en `SortedList` (no logra capturar la mayoría del ground truth); EvoSpex infiere aserciones complejas para `SortedList.insert` (que el elemento es insertado) y se pierde las tres aserciones restantes; también infiere parte del ground truth para `min`. Daikon infiere lo mismo que EvoSpex en el caso de `min`, y en el caso de `SortedList.insert`, solo infiere orden específico entre los primeros pocos elementos de la lista, pero falla en generalizar esta relación para la estructura completa; además se pierde completamente las aserciones restantes en el ground truth.

SpecFuzzer

SpecFuzzer usa una combinación de análisis estático, fuzzing basado en gramática, y análisis de mutación para inferir especificaciones. La herramienta procede de la siguiente manera. Primero, utiliza un análisis estático liviano para producir una gramática para el lenguaje de especificación, que está adaptada para el software bajo análisis. Luego, utiliza un fuzzer basado en gramática para generar especificaciones candidatas a partir de esa gramática. En el siguiente paso, un detector dinámico determina cuáles de esas propiedades son consistentes con el comportamiento exhibido por una test suite provista. Finalmente, SpecFuzzer elimina especificaciones relevan-

Cuadro 7.2: Desempeño de Daikon y SpecFuzzer en los ejemplos.

	Daikon	SpecFuzzer
	min(int x,int y) - postcondition	
1	`r <= x`	`x >= r`
2	`r <= y`	`x < r ==> r <= 1`
3		`x >= y ==> y == r`
4		`x >= y || y != r`
5		`x <= y || y <= r`
6		`x <= y || y >= r`
7		`x >= y <==> y == r`
8		`x == y ==> y <= r`
9		`x <= y <==> x == r`
	SortedList.insert(int data) - postcondition	
1	`elem <= next.elem`	`exists SortedList l: reach(this, next).has(l) && l.elem == SENTINEL`
2	`elem <= next.next.elem`	`all SortedList l: reach(this, next).has(l) ==> l.elem <= l.next.elem`
3	`next.elem <= next.next.elem`	`exists SortedList l: reach(this, next).has(l) && l.elem == data`
4	`next != null`	`next != null`
5	`elem <= old(elem)`	`elem != old(elem) + 1`
6	`elem <= old(next.elem)`	`next.elem >= old(elem)`
7	`elem <= old(next.next.elem)`	`data >= next.elem || next.elem = old(elem)`
8	`elem <= data`	`elem == data xor data > old(elem)`
9	`next.elem >= old(elem)`	`elem > data ==> data = next.elem`
10	`next.elem <= old(next.elem)`	`exists SortedList l: reach(this.next, next).has(l) && l.elem != 1`
11	`next.elem <= old(next.next.elem)`	`exists SortedList l: reach(this, next).has(l) && l.elem > this.elem`
12	`next.next.elem >= old(elem)`	`exists SortedList l: reach(this, next).has(l) && l.elem <= l.next.elem`
13	`next.next.elem >= old(next.elem)`	`all SortedList l: reach(this.next, next).has(l) ==> l.elem >= this.next.elem`
14	`next.next.elem <= old(next.next.elem)`	`elem <= old(elem) ==> old(elem) < old(next.next.elem)`
15		`elem != next.next.elem + old(next.next.elem)`
16		`data >= next.next.elem || next.next.elem == old(next.elem)`

tes y equivalentes utilizando un mecanismo basado en análisis de mutación y clustering. Una característica destacada de SpecFuzzer es que los desarrolladores pueden ajustar el conjunto de especificaciones producido adaptando la gramática en lugar de hacer cambios en la herramienta.

El Cuadro 7.2 muestra las aserciones que SpecFuzzer infiere como poscon-

diciones para los métodos `min` y `SortedList.insert`. Notemos que SpecFuzzer utiliza fuzzing y reporta un mayor número de aserciones en comparación con las otras técnicas. Configuramos SpecFuzzer para producir 2000 aserciones candidatas por caso de estudio y encontramos que 51 y 437 fueron confirmadas como invariantes probables por un detector dinámico para `min` y `insert`, respectivamente. La estrategia de partición basada en mutación es lo que permite a SpecFuzzer reducir considerablemente el número de aserciones reportadas a 9 y 16, respectivamente.

En el caso de `min`, las 9 aserciones inferidas son válidas y su conjunción es equivalente al ground truth correspondiente. Para `SortedList.insert`, las primeras 3 aserciones cubren 3 de las 4 aserciones del ground truth (la única ausente es la de aciclicidad). Las otras aserciones inferidas o bien son válidas pero menos relevantes (4-13), o inválidas (14-16). Las inválidas son especificaciones que son verdaderas en las test suites provistas, pero que existe algún escenario no visto en el que son falsificadas. Notemos que esto también afecta a las otras técnicas, incluso aunque GAssert y EvoSpex incluyen mecanismos costosos para reducir las aserciones inválidas (en particular, la aserción inferida por GAssert para `SortedList.insert` es una propiedad inválida).

7.2. SpecFuzzer

Esta sección presenta SpecFuzzer, nuestra técnica para inferencia de especificaciones basada en una combinación de análisis estático, fuzzing basado en gramática, y análisis de mutación.

La Figura 7.3 muestra el flujo de trabajo de nuestra técnica. SpecFuzzer toma como input una clase Java[1] C, y produce aserciones que buscan caracterizar propiedades de diferentes puntos de programas en C, como precondiciones y poscondiciones. Siguiendo la terminología de Daikon, generalmente nos referiremos a aserciones que valen en puntos específicos del programa como *invariantes*. La técnica está organizada como un pipeline de cinco componentes: (1) un componente de *Generación de Tests y Mutantes* que produce

[1]Aunque nuestro enfoque es general e independiente del lenguaje, algunas partes de nuestro prototipo actual, como la extracción de la gramática y la evaluación de especificaciones candidatas obtenidas mediante fuzzing, están implementadas para Java. Soportar otros lenguajes que Daikon puede manejar, como C, requeriría la implementación de esas partes.

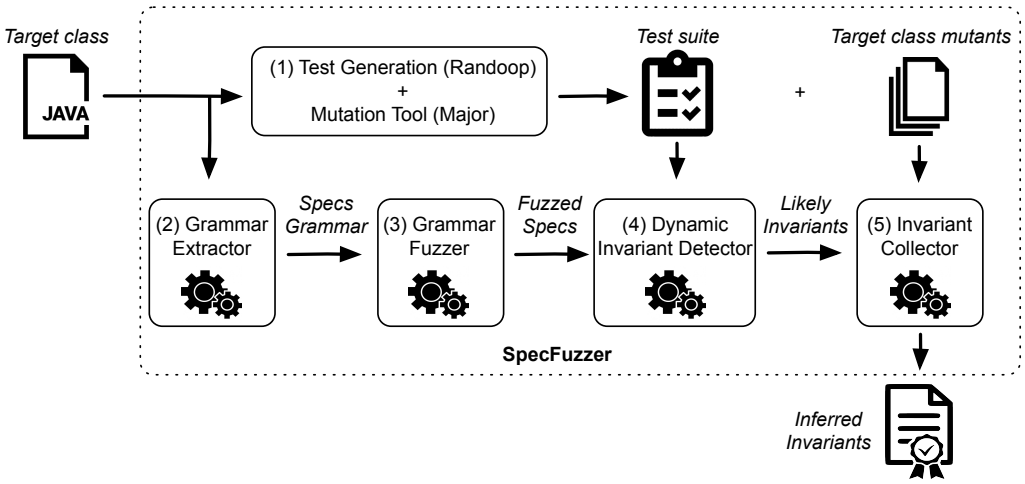


Figura 7.3: Flujo de trabajo de la técnica propuesta.

tests y mutantes para otros componentes del pipeline, (2) un *Extractor de Gramática* que analiza C para generar una gramática de especificación para esa clase, (3) un *Fuzzer basado en Gramática*, el cuál produce aserciones candidatas explorando las reglas de producción a partir de la gramática extraída, (4) un *Detector Dinámico de Invariantes* , responsable de inferir los invariantes probables a partir de las aserciones generadas con el fuzzer mediante observaciones sobre las ejecuciones de una test suite dada, y (5) un componente *Selector de Invariantes*, que particiona los invariantes probables producidos por el componente previo para descartar aserciones poco útiles (débiles), agrupa aserciones similares, y reporta un conjunto reducido de aserciones, priorizando las más fuertes. Las siguientes subsecciones presentan estos componentes en mayor detalle.

7.2.1. Generación de Tests y Mutantes

El primer paso de nuestro proceso para inferir especificaciones para una clase C consiste de *(i)* generar una test suite T ejercitando los métodos de la clase objetivo C, y *(ii)* producir un conjunto $M_1, \ldots, M_n$ de mutantes de C, representando fallas en la clase. Como muestra la Figura 7.3, estos artefactos son utilizados en distintas etapas de la técnica. Para la generación de los

⟨FuzzedSpec⟩ ::= ⟨QuantifiedExpr⟩ | ⟨BooleanExpr⟩
⟨QuantifiedExpr⟩ ::= ⟨Quantifier⟩ ⟨Typed_Var⟩ : ⟨BooleanExpr⟩
⟨Quantifier⟩ ::= `all` | `exists`
⟨BooleanExpr⟩ ::= ⟨NumCmpExpr⟩ | ⟨LogicCmpExpr⟩ | ⟨MembershipExpr⟩ | `!`⟨BooleanExpr⟩
⟨NumCmpExpr⟩ ::= ⟨NumExpr⟩ ⟨NumCmpOp⟩ ⟨NumExpr⟩
| ⟨NumExpr⟩ ⟨NumCmpOp⟩ ⟨NumExpr⟩ ⟨NumBinOp⟩ ⟨NumExpr⟩
⟨NumExpr⟩ ::= ⟨NumVar⟩ | ⟨NumConst⟩
⟨LogicCmpExpr⟩ ::= ⟨BooleanExpr⟩ ⟨LogicOp⟩ ⟨NumCmpExpr⟩
| `(`⟨BoolVar⟩⟨LogicOp⟩⟨BoolVar⟩`)` ⟨LogicOp⟩ ⟨NumCmpExpr⟩
| `(`⟨NumCmpExpr⟩`)` ⟨LogicOp⟩ `(`⟨NumCmpExpr⟩`)`
⟨MembershipExpr⟩ ::= ⟨type_SetExpr⟩`.has(`⟨type_Var⟩`)`
⟨NumCmpOp⟩ ::= `==` | `!=` | > | < | >= | <=
⟨NumBinOp⟩ ::= `+` | `-` | `*` | `/` | `%`
⟨LogicOp⟩ ::= `||` | `xor` | `==>` | <==>

Figura 7.4: Fragmento de la gramática base B.

tests utilizamos Randoop[2] y para la generación de mutantes Major[3]. Aunque en nuestra implementación actual utilizamos estas herramientas, el usuario podría reemplazarlas con otras herramientas o incluso proveer su propia test suite y versiones mutadas de la clase C.

7.2.2. Extractor de Gramática

El *Extractor de Gramática* toma como input una clase C y crea una gramática G_c, que expresa el lenguaje de aserciones candidatas para C. Estas aserciones denotan precondiciones, poscondiciones e invariantes de clase. El extractor instancia nuestra gramática base, a las que nos referimos como B, con información que es específica a C, por ejemplo, tipos de los atributos, expresiones navigacionales legalmente tipadas involucrando los atributos, etc.

La Figura 7.4 muestra un fragmento de la gramática base B, capturando las partes fijas del lenguaje de especificación, es decir, las partes que son comunes a cualquier clase input de interés. Actualmente, la gramática B soporta comparaciones numéricas, expresiones lógicas, expresiones de pertenencia, y expresiones cuantificadas. Las comparaciones numéricas y las expresiones

[2] `https://randoop.github.io/randoop/`
[3] `https://mutation-testing.org/`

lógicas son las construcciones más simples del lenguaje. Relacionan expresiones numéricas y expresiones booleanas utilizando operadores numéricos tradicionales y conectivos lógicos—$\langle NumCmpOp \rangle$ y $\langle LogicOp \rangle$, respectivamente. Las expresiones de pertenencia permiten expresar cuando un elemento tipado pertenece a un conjunto (colección) del tipo correspondiente. El fragmento de la gramática utiliza la notación `has` de JML [11], y muestra una regla de producción para variables tipadas. Aunque no se muestra explícitamente en el fragmento de la gramática, el operador `reach` es una forma de construir expresiones de conjuntos tipadas. Un ejemplo concreto de una expresión de pertenencia de una fórmula mostrada en la Sección 7.1 es la siguiente:

```
reach(this, next).has(l)
```

expresando que la lista `l` pertenece al conjunto de objetos alcanzables desde `this` mediante navegaciones (cero o más) a través de `next`. Finalmente, la gramática permite cuantificación existencial y universal. De nuevo, un ejemplo de expresión cuantificada de la Sección 7.1 es el siguiente:

```
exists SortedList l:
     reach(this, next).has(l) && l.elem == SENTINEL
```

cuya lectura intuitiva es que existe un objeto alcanzable desde `this` con el campo `elem` teniendo el valor `SENTINEL`.

Para obtener la gramática G_c, el extractor toma el fragmento B y agrega o elimina símbolos y reglas de producción, a partir de la estructura de C. El proceso básicamente depende de los campos directos e indirectos (campos declarados en C o una clase alcanzable desde C) de C. Intuitivamente, a partir de cada campo/navegación, un símbolo terminal del tipo correspondiente es definido (por ejemplo, `this.next` será un terminal de tipo `SortedList`).

Las expresiones de conjuntos merecen una descripción más detallada. Primero, si un campo `f` es de un tipo `java.util.Collection`, entonces `f` estará en un terminal de `SetExpr` (tipado). Por ejemplo, si `SortedList` fuera una implementación de `java.util.Collection`, entonces `this` y `this.next` serían terminales de tipo `SortedList_SetExpr`. Segundo, el operador `reach` también está involucrado en la creación de expresiones denotando conjuntos. Para la expresión `e` y el campo recursivo `f` (un campo es recursivo si está definido en una clase C y tiene como tipo C) de la clase `C`, una regla de producción permite que la expresión `reach(e, f)` tenga el tipo `C_SetExpr`. Entonces, la expresión `reach(this, next)` tiene tipo `SortedList_SetExpr`.

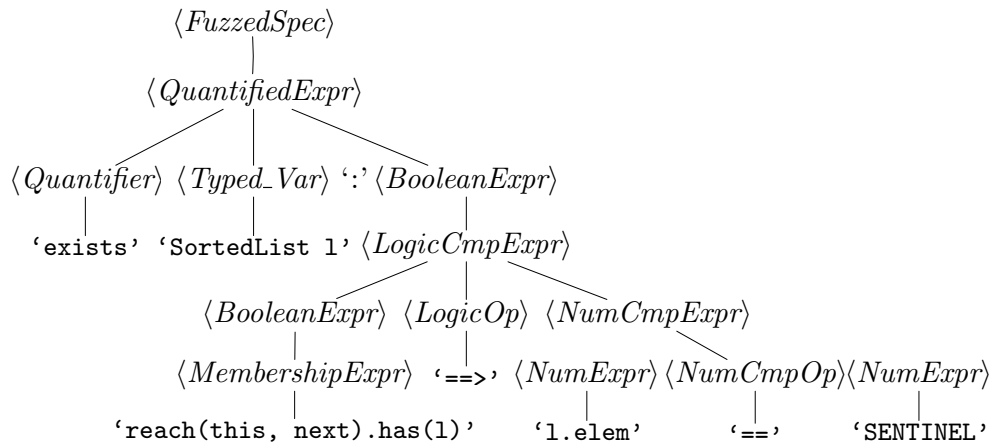


Figura 7.5: Árbol de derivación para la propiedad `exists SortedList l: reach(this, next).has(l) && l.elem == SENTINEL`.

7.2.3. Fuzzer basado en Gramática

El objetivo del *Fuzzer basado en Gramática* es producir aserciones candidatas. Para lograr este objetivo, utilizamos un fuzzer generativo estándar basado en gramática [35, 94]. Este componente produce derivaciones de G_c—es decir, cadenas en $\mathcal{L}(G_c)$—para obtener aserciones para C. Comienza con el símbolo inicial ⟨*FuzzedSpec*⟩ y va expandiendo símbolos no-terminales hasta que no quedan más no-terminales presentes. Cada símbolo no-terminal es expandido basado en una elección no determinista, para evitar expansiones llevando a caminos de derivación infinitos, y se define un límite de 5 en el número de no-terminales. Además, para evitar situaciones en las que el número de símbolos ya no puede ser reducido, el número total de expansiones también se limita a 100. La razón de esta elección es que las aserciones complejas pueden ser creadas mediante la combinación de otras más pequeñas, en lugar de más grandes. La Figura 7.5 muestra el árbol de derivación de una propiedad utilizada en nuestro ejemplo ilustrativo: `exists SortedList l: reach(this, next).has(l) && l.elem == SENTINEL`.

Mediante el uso de este mecanismo de derivación, nuestro *Fuzzer basado en Gramática* produce predicados candidatos de manera muy eficiente. En todos nuestros experimentos generamos hasta 2000 candidatos diferentes cada vez que ejecutamos SpecFuzzer. Además, como la gramática G_c ha sido específicamente extraída para una clase C, todas las especificaciones

generadas por el fuzzer tienen la garantía de expresar propiedades sobre C. Implementamos nuestro fuzzer en Java, reproduciendo un fuzzer basado en gramática general escrito en Python [94][4].

7.2.4. Detector Dinámico de Invariantes

El objetivo del *Detector Dinámico de Invariantes* es evaluar la validez de las aserciones candidatas producidas por el fuzzer. Como muestra la Figura 7.3, el detector dinámico de invariantes toma como input una test suite, producida por el generador de tests, y un conjunto de aserciones, producidas por el fuzzer. Este componente instrumenta el programa con las aserciones generadas por el fuzzer y ejecuta los tests para verificar que aserciones valen a través de todas las ejecuciones. Las aserciones resultantes son reportadas como *invariantes probables*.

El detector dinámico está construido encima de Daikon[5], una herramienta estado del arte para detección de invariantes probables. Utilizamos Daikon del siguiente modo. Lo configuramos para incluir las aserciones provistas, es decir, las expresiones producidas por el fuzzer, en el pool inicial de aserciones candidatas. Para ello, utilizamos un mecanismo provisto por Daikon para incorporar nuevos invariantes candidatos[6]. Además, incluimos, junto con los nuevos invariantes candidatos, un componente que le permite a la herramienta interpretar y evaluar tales candidatos en tiempo de ejecución.

7.2.5. Selector de Invariantes

El objetivo del *Selector de Invariantes* es particionar las aserciones que fueron determinadas válidas por el detector dinámico de invariantes, agrupando aserciones similares, y tomando un único representante de cada partición. Al mismo tiempo, este componente descarta aserciones que, aunque fueron confirmadas por el detector de invariantes, son consideradas débiles y menos relevantes. Este componente toma como input el conjunto de invariantes probables obtenidos a partir del paso previo, y el conjunto de mutantes para la clase input C (obtenidos mediante una herramienta de mutación). Este componente reporta un subconjunto de las aserciones probables recibidas como input, valorando los invariantes por el número de fallas en las

[4] `https://www.fuzzingbook.org/html/Grammars.html#A-Simple-Grammar-Fuzzer`
[5] `http://plse.cs.washington.edu/daikon/`
[6] `http://plse.cs.washington.edu/daikon/download/doc/developer.html`

correspondientes aserciones de código. El Selector de Invariantes reduce el número de aserciones reportadas. Para resumir, este componente descarta aserciones por dos razones:

1. la aserción es considerada *débil*, y no captura propiedades relevantes de la clase objetivo,

2. la aserción es (semánticamente) *similar* a otras aserciones.

Debajo describimos nuestro mecanismo para aproximar la detección de especificaciones débiles y equivalentes mediante análisis de mutación.

Detectando Especificaciones Débiles

Recordemos que el fuzzer produce miles de propiedades y el detector dinámico de invariantes (en nuestro caso, Daikon) solo puede descartar especificaciones invalidadas por los tests. Varias propiedades todavía pueden "sobrevivir" al proceso de filtrado descrito en la Sección 7.2.4. Incluso con mejores test suites algunas aserciones todavía "sobreviven" ese proceso. Por ejemplo, una *tautología*, como `x >= y || x <= y`, *no* sería invalidada por Daikon ya que es una proposición válida, pero es poco probable que sea útil. Siendo dirigido de manera sintáctica, el fuzzer puede producir aserciones válidas que no proveen información interesante. La suposición es que también genera aserciones interesantes. SpecFuzzer utiliza *análisis de mutación para descartar aserciones que no son interesantes*. La idea es la siguiente: si una aserción –una que no es falsificada por la test suite de C– no puede ser falsificada por ningún mutante de C, entonces es una propiedad que no solo vale en `C`, sino que también vale en todas las versiones *buggy* de C. Esta aserción es considerada *débil*, y descartada como *irrelevante*. Este enfoque usa análisis de mutación para generar especificaciones más efectivas (más fuertes), como ya ha sido usado en trabajo previo, notablemente en [17], como así también en trabajo reciente de mejora de especificaciones [85] e inferencia de especificaciones [64].

Clustering de Especificaciones Similares

Es posible que SpecFuzzer produzca aserciones sintácticamente diferentes que son semánticamente válidas (o similares con respecto a una métrica de distancia). SpecFuzzer trata de identificar y *eliminar* estas aserciones. A

modo de ejemplo, consideremos las siguientes aserciones que son producidas por SpecFuzzer en nuestro ejemplo `SortedList`.

```
all SortedList l: reach(this, next).has(l) ==>
                                    l.elem <= l.next.elem
all SortedList l: !(reach(this, next).has(l) &&
                                    l.elem > l.next.elem)
```

Ambas aserciones expresan la propiedad de orden de las listas. La equivalencia de estas especificaciones surge de las leyes de De Morgan [67], y propiedades algebraicas de enteros, y la equivalencia de conectores lógicos. Para identificar aserciones equivalentes y aserciones que son similares con respecto a su habilidad para capturar fallas introducidas, SpecFuzzer usa análisis de mutación nuevamente. Dos aserciones serán consideradas similares si matan el mismo conjunto de mutantes, es decir, son falsificadas en el mismo conjunto de fallas. Por ejemplo, las dos aserciones de arriba matan el mismo conjunto de 2 mutantes, junto con otras 74 especificaciones. Más precisamente, SpecFuzzer particiona el conjunto de aserciones probables de acuerdo a los mutantes que matan. Finalmente, de cada partición, SpecFuzzer elige una aserción representativa. Por ello, procedemos con la siguiente heurística: las aserciones en cada partición son valoradas por el número de veces que fallan cuando se ejecuta la test suite en los mutantes (mientras que todas matan el mismo conjunto de mutantes, algunas aserciones pueden fallar un mayor número de veces, es decir, para más tests en la test suite). El razonamiento es que las aserciones que fallan más veces representan propiedades más fuertes, y por lo tanto pueden incluir propiedades capturadas por las otras aserciones de la partición. La Figura 7.6 muestra el desglose de aserciones clasificadas como irrelevantes (Section 7.2.5), equivalentes (Section 7.2.5), y las finalmente reportadas. Considerando el ejemplo `SortedList.insert`, este mecanismo le permitió a SpecFuzzer reducir el número de especificaciones reportadas de 437 a 16.

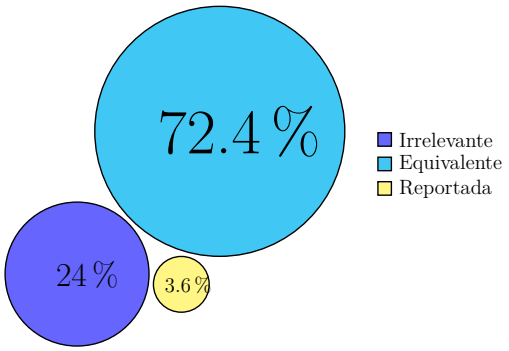


Figura 7.6: Desglose de los motivos por los que se descartan especificaciones en `SortedList.insert`. Solo el 3.6 % de las especificaciones que sobreviven la etapa de detección son reportadas.

Implementación

SpecFuzzer, junto con instrucciones para realizar los experimentos presentados en la siguiente sección, está públicamente disponible para su descarga en el siguiente sitio:

`https://sites.google.com/view/specfuzzer`

7.3. Evaluación Experimental

Para evaluar SpecFuzzer, realizamos una serie de experimentos enfocados en las siguientes preguntas de investigación:

RQ1 *¿Es el fuzzing basado en gramática efectivo para generar aserciones relevantes?*

RQ2 *¿Es el selector basado en análisis de mutación exitoso para eliminar aserciones redundantes/irrelevantes?*

RQ3 *¿Cómo se compara SpecFuzzer con técnicas alternativas?*

La RQ1 analiza la efectividad de utilizar fuzzing basado en gramática como técnica de generación de aserciones candidatas, con respecto a un ground truth. La RQ2 evalúa la capacidad del componente de selección de aserciones basado en análisis de mutaciones, para descartar aserciones débiles y priorizar las más relevantes. Finalmente, la RQ3 compara la efectividad de SpecFuzzer con las técnicas estado del arte GAssert [85] y EvoSpex [64].

7.3.1. Casos de Estudio

La evaluación de performance en los trabajos previos utiliza métricas indirectas para computar precision y recall [85, 64]. En este trabajo, computamos estas métricas directamente, lo que requiere tener una *ground truth* para comparación, para cada caso de estudio. Para ello, tomamos los 34 métodos Java que fueron parte de la evaluación de GAssert [85], y los 23 métodos en la evaluación de reproducibilidad de contratos de EvoSpex [64], obteniendo un conjunto de 57 métodos. Estudiamos cada método, y creamos manualmente las aserciones "ground truth" correspondientes capturando el comportamiento esperado de cada método. Estas aserciones fueron chequeadas por todos

los autores de este trabajo de manera cruzada y pueden ser encontradas en el sitio. Notemos que, debido a qué el trabajo previo se enfoca solo en la inferencia de poscondiciones, nuestra evaluación también se centró en estos puntos de programas, aunque SpecFuzzer puede inferir aserciones para varios puntos de programas. Obtuvimos un total de 80 aserciones de ground truth, para los 57 métodos. Luego, examinamos cuidadosamente cada aserción del ground truth, para determinar si podían ser expresadas en el lenguaje de aserciones de al menos una de las herramientas evaluadas. Dado que 15 de las 80 aserciones no estaban soportadas por ninguna de las herramientas, fueron descartadas[7]. Los métodos que, luego de esta eliminación, terminaron sin ninguna poscondición, también fueron eliminados. Esto resultó en 65 aserciones, para 43 métodos Java.

Cada aserción dentro de las 65 en el ground truth puede ser expresada en el lenguaje de aserciones de al menos una herramienta. El lenguaje de Gassert puede expresar 28 de las 65; el lenguaje de EvoSpex puede expresar 29 de las 65; y SpecFuzzer puede expresar 41 de las 65. Aunque nuestra gramática soporta las 65 aserciones, cuando implementamos soporte para la gramática en la instanciación de aserciones a partir de templates de Daikon, algunas expresiones son ignoradas por la infraestructura de Daikon (por ejemplo, expresiones cuya instanciación a partir de templates requieren objetos de diferentes clases). No hay ninguna razón fundamental por la que estos problemas no puedan ser resueltos, pero demandan modificaciones en Daikon.

7.3.2. Setup Experimental

SpecFuzzer require una test suite para la clase bajo análisis, y un conjunto de mutantes para esta clase. Las test suites fueron generadas utilizando Randoop [68], con un máximo de 500 secuencias de tests. Los mutantes fueron generados utilizando Major [41], con todos los operadores de mutación soportados habilitados. El fuzzer fue ejecutado para producir hasta 2000 aserciones candidatas diferentes (los duplicados sintácticamente fueron eliminados), para cada clase objetivo.

En relación a GAssert y EvoSpex, seguimos la misma metodología descrita en los artículos correspondientes [85, 64], usando exactamente la misma

[7]Las aserciones que descartamos incluían propiedades trigonométricas complejas, restricciones del producto cruz de vectores, aserciones involucrando strings y caracteres, y conversiones entre caracteres y codificaciones hexadecimales.

configuración de parámetros para los procesos evolutivos de cada técnica. Más aún, para tener en cuenta la aleatoriedad de cada técnica, para cada uno de los 43 métodos Java inferimos poscondiciones con cada herramienta un total de 10 veces. Todos los resultados reportados en esta sección corresponden al promedio de las ejecuciones.

Definimos un timeout de 90 minutos, para cada ejecución de cada herramienta. Todas las herramientas fueron ejecutadas en un procesador Intel Core i7 3.2 Ghz, con 16Gb de RAM, con GNU/Linux (Ubuntu 18.04). Una descripción detallada sobre como reproducir los experimentos está disponible en el sitio.

7.3.3. Efectividad de Fuzzing basado en Gramáticas

La efectividad del fuzzing basado en gramáticas en la generación de aserciones relevantes es medida con respecto a las aserciones del ground truth. El experimento para la RQ1 consistió en ejecutar SpecFuzzer en cada caso, y analizar el porcentaje de aserciones del ground truth que la herramienta fue capaz de inferir. Recordemos que el selector de invariantes usa una heurística (basada en mutaciones) para descartar aserciones. De este modo, podría descartar de manera incorrecta aserciones relevantes. Por esa razón, para responder la RQ1, ejecutamos SpecFuzzer con el selector de invariantes deshabilitado.

El output de SpecFuzzer fue inspeccionado manualmente. Más precisamente, analizamos manualmente las aserciones que reportó SpecFuzzer para verificar si estaban presentes o no en el ground truth. En algunos casos, fue difícil determinar si la aserción era equivalente a ciertas aserciones del ground truth. Cuando las aserciones obtenidas lo permitieron, utilizamos un SMT solver a través de Microsoft IntelliTest para chequear equivalencia de expresiones. En resumen, generamos programas C# para chequear cuando el conjunto de aserciones inferidas era equivalente a las aserciones en el ground truth.

El Cuadro 7.3 resume los resultados de los experimentos para RQ1, con respecto al ground truth general (65 aserciones) y el subconjunto del ground truth que es actualmente soportado por SpecFuzzer (como explicamos previamente, 41 de 65 aserciones son actualmente soportadas por nuestra implementación). Mostramos el número de aserciones reportadas, el número de aserciones del ground truth detectadas por la herramienta, y el porcentaje de aserciones en el ground truth que fueron detectadas. Si consideramos el

Cuadro 7.3: Infiriendo aserciones con fuzzing basado en gramáticas.

Ground-truth	#Reportadas	#Detectadas	%Detectadas
65	20,277	40	**61.5**
41	15,555	40	**97.5**

lenguaje de aserciones soportado por nuestra implementación, nuestra herramienta detecta correctamente el 97.5 % de las aserciones en el ground truth; si consideramos el lenguaje soportado por al menos una de las herramientas de inferencia, SpecFuzzer detecta correctamente el 61.5 % de las aserciones en el ground truth. Estos resultados confirman que el fuzzing basado en gramáticas es efectivo para generar aserciones relevantes (como mostramos más adelante, incluso cuando consideramos las 65 aserciones en el ground truth, la performance de fuzzing es comparable con el estado del arte).

7.3.4. Performance del Selector de Invariantes

El selector de invariantes de SpecFuzzer implementa heurísticas basada en mutaciones para reducir el número de aserciones reportadas, descartando aserciones "débiles" (aserciones que sobreviven a todos los mutantes), y eligiendo representantes entre las aserciones que matan los mismos mutantes. La RQ2 evalúa la performance de esta etapa. El experimento en este caso compara las aserciones obtenidas luego de la detección de invariantes, con las aserciones que son preservadas luego de ejecutar la selección de invariantes. La comparación mide la reducción de aserciones, y el porcentaje del ground truth que es cubierto antes y después de la selección de aserciones.

El Cuadro 7.4 muestra estos resultados para todos los casos de estudio, agrupados por nombre de clase. En cada caso reportamos las aserciones en el ground truth, las aserciones reportadas antes y después de la selección de invariantes, y el porcentaje del ground truth que es cubierto, de nuevo, antes y después de la selección de invariantes. Finalmente, indicamos el ratio de reducción obtenido mediante selección de invariantes (número de aserciones luego de la selección, con respecto al número de aserciones previo a la selección).

Los resultados de selección de invariantes muestran que las heurísticas basadas en mutación incorporadas en SpecFuzzer efectivamente reducen el

Cuadro 7.4: Performance del Selector de Invariantes reduciendo aserciones.

Caso de Estudio	#GT	#Reportadas		Detectadas(%)		Red. (%)
		Antes	Después	Antes	Después	
oasis.SimpleMethods	4	115	31	75	75	73
daikon.StackAr	8	2067	70	87.5	62.5	96.6
daikon.QueueAr	8	4699	152	50	50	96.7
math.ArithmeticUtils	1	4	2	100	100	50
math.FastMath	2	60	31	100	100	48.3
math.MathUtils	1	19	4	0	0	78.9
lang.BooleanUtils	5	49	12	100	100	75.5
guava.IntMath	1	314	46	0	0	85.3
tsuite.Angle	2	3	0	50	0	100
tsuite.MathUtil	3	22	13	33.3	33.3	40.9
tsuite.Envelope	1	1094	27	0	0	97.5
eiffel.Composite	4	8696	42	75	0	99.5
eiffel.DLLN	3	137	29	100	100	78.8
eiffel.Map	6	140	22	16.6	16.6	84.2
eiffel.RingBuffer	5	1947	269	20	20	86.1
cozy.Polyupdate	3	382	119	66.6	66.6	68.8
cozy.Structure	2	153	21	100	100	86.2
cozy.ListComp02	2	62	5	0	0	91.9
cozy.MinFinder	1	17	2	100	100	88.2
cozy.MaxBag	3	297	78	100	100	73.7
TOTAL	65	20277	975	61.5	52.3	95.1
	41	15555	618	97.5	82.9	96

número de aserciones reportadas, con una pérdida relativamente pequeña en detección de propiedades (con respecto al ground truth). Más precisamente, las aserciones reportadas son reducidas por un ∼95 %, y sólo 6 de las 40 aserciones correctamente producidas mediante fuzzing son descartadas (cubriendo un ∼52 % del ground truth de 65 aserciones, y un ∼83 % de aserciones del ground truth que la herramienta soporta).

Para entender las razones por las que perdemos 6 aserciones del ground truth durante la selección de invariantes, analizamos como estas aserciones fueron clasificadas por el detector. En todos los casos, las aserciones son consideradas *irrelevantes*, es decir, no son falsificadas por ningún mutante. Mientras que el problema podría ser una test suite débil, se vuelve claro que, si se observan las aserciones, no hay un operador de mutación capaz de falsificar estas aserciones (las aserciones se muestran en el Cuadro 7.5). El problema no es específico de Major (la herramienta de mutación que utilizamos); otras herramientas como PIT tampoco tienen mutantes capaces de falsificar estas aserciones. Proveamos dos ejemplos concretos. La aserción `abs(res) <= 1`

Cuadro 7.5: Aserciones válidas descartadas por el Selector de Invariantes.

Caso de Estudio	**Aserción**
StackAr.pop	`theArray[old(top)] == null`
StackAr.topAndPop	`theArray[old(top)] == null`
Angle.getTurn	`abs(res) <= 1`
Composite.addChild	`c.value == old(c.value))`
	`children == old(children)`
	`ancestors == old(ancestors)`

para `Angle.getTurn` corresponde a un método cuyo `resultado` es 0, -1 o 1; ningún mutante hace que esta método retorne un valor diferente a estos tres. En el método `Composite.addChild`, la aserción `c.value == old(c.value)` sería falsificada si un mutante cambiara el valor de `c`, un parámetro del método; un operador de mutación que logre este efecto debería incluir una nueva sentencia.

Estas observaciones sugieren que podríamos mejorar la efectividad de nuestras heurísticas extendiendo Major con soporte para operadores de mutación adicionales, específicos para nuestros propósitos.

7.3.5. Comparando GAssert, EvoSpex y SpecFuzzer

La RQ3 compara SpecFuzzer con las herramientas estado del arte GAssert y EvoSpex. La comparación está basada en métricas estándar de performance: precision, recall y f1-score. Estas métricas son computadas con respecto al ground truth que definimos para los casos de estudio. Las herramientas fueron ejecutadas para inferir aserciones como describimos previamente en esta sección, y los resultados se muestran en el Cuadro 7.6, agrupados por clases. Las columnas #M y #GT muestran el número de métodos en el caso de estudio y el número de aserciones en el ground truth, respectivamente. Para cada técnica mostramos el número de aserciones inferidas, la precision y el recall con respecto al ground truth, y el f1-score. Resumimos las medidas de performance para el ground truth general de 65 aserciones, así como también en el contexto de aserciones que son soportadas por cada herramienta particular (recordemos que GAssert soporta en su lenguaje 28 de 65, EvoSpex soporta 29 de 65, y SpecFuzzer 41 de 65). Es decir, las columnas Total-SpecFuzzer, Total-GAssert y Total-EvoSpex muestran la performance de las técnicas en la porción del ground truth que SpecFuzzer, GAssert y

Cuadro 7.6: Precision, Recall y F1-Score de GAssert (GA), EvoSpex (ES) y SpecFuzzer (SF) en los casos de estudio.

Caso de Estudio	#M	#GT	#Inferidas			Precision(%)			Recall(%)			F1-Score		
			GA	ES	SF	GA	ES	SF	GA	ES	SF	GA	ES	SF
oasis.SimpleMethods	4	4	7	4	31	100	75	100	50	25	75	0.66	0.37	0.85
daikon.StackAr	5	8	5	6	70	100	83.3	87.1	37.5	37.5	62.5	0.54	0.51	0.72
daikon.QueueAr	5	8	8	12	152	100	91.6	61.1	37.5	25	50	0.54	0.39	0.54
math.ArithmeticUtils	1	1	1	0	2	100	100	50	100	0	100	1	0	0.66
math.FastMath	1	2	3	1	31	100	0	61.2	100	0	100	1	0	0.75
math.MathUtils	1	1	1	1	4	100	0	100	0	0	0	0	0	0
lang.BooleanUtils	2	5	2	2	12	50	100	83.3	0	0	100	0	0	0.9
guava.IntMath	1	1	3	3	46	100	66.6	97.8	100	0	0	1	0	0
tsuite.Angle	1	2	3	1	0	100	100	100	0	0	0	0	0	0
tsuite.MathUtil	1	3	3	1	13	100	100	84.6	33.3	0	33.3	0.49	0	0.47
tsuite.Envelope	1	1	3	4	27	100	100	18.5	0	0	0	0	0	0
eiffel.Composite	1	4	0	7	42	100	100	50	0	50	0	0	0.66	0
eiffel.DLLN	2	3	0	4	29	100	100	89.6	0	66.6	100	0	0.79	0.94
eiffel.Map	3	6	4	10	22	50	100	81.8	16.6	66.6	16.6	0.24	0.79	0.27
eiffel.RingBuffer	5	5	9	31	269	88.8	87	64.6	20	40	20	0.32	0.54	0.3
cozy.Polyupdate	2	3	3	3	119	66.6	100	1.6	33.3	66.6	66.6	0.44	0.79	0.03
cozy.Structure	2	2	2	2	21	100	100	95.2	100	100	100	1	1	0.97
cozy.ListComp02	2	2	0	4	5	100	100	83.3	0	100	0	0	1	0
cozy.MinFinder	1	1	0	2	2	100	100	100	0	100	100	0	1	1
cozy.MaxBag	3	3	8	33	78	100	84.8	94.8	0	66.6	100	0	0.74	0.97
Total-65	43	65	65	131	975	92.3	88.5	65.8	27.6	38.4	52.3	0.42	0.53	0.57
Total-SpecFuzzer	30	41	43	76	618	95.3	88	62.4	41.4	43.9	82.9	0.57	0.58	0.71
Total-GAssert	23	28	39	54	615	94.8	85.1	63.4	64.2	46.4	71.4	0.76	0.60	0.67
Total-EvoSpex	24	29	34	82	624	91.1	92.8	59.9	44.8	86.2	72.4	0.60	0.89	0.65

EvoSpex soportan, respectivamente.

Aserciones Inferidas

Si nos enfocamos en el número de aserciones inferidas, GAssert y EvoSpex reportan menos aserciones que SpecFuzzer. Esta es una ventaja de las técnicas previas, ya que el output producido es más fácil de interpretar. La razón principal es que ambas técnicas utilizan procesos evolutivos, que tienen como objetivo minimizar el tamaño de las aserciones (este es uno de los objetivos en los procesos evolutivos). En este aspecto, SpecFuzzer es una técnica más simple. Aún así, el selector de invariantes permite que nuestra herramienta reporte un número razonable de aserciones (22 por método, en promedio).

Precision

La precision es el aspecto en que tanto GAssert como EvoSpex superan a SpecFuzzer. De nuevo, esto tiene que ver con el hecho de que tanto GAssert como EvoSpex incorporan mecanismos para reducir activamente el número de falsos positivos (entendidos como propiedades inválidas). En particular, GAssert mejora las aserciones iterativamente usando OASIs [37], ejecutando instancias de EvoSuite para buscar y detectar defectos en las aserciones candidatas. EvoSpex utiliza una técnica exhaustiva acotada de generación de tests con el objetivo de construir una test suite más completa, capaz de descartar más falsos positivos. Ambas técnicas tienen desventajas asociadas con estos procesos. GAssert paga un precio en eficiencia (es la más costosa de las tres); la generación exhaustiva acotada de EvoSpex tiene problemas de escalabilidad (la generación exhaustiva acotada supone dificultades para escalar a casos de estudio más grandes).

Para lidiar con precision, SpecFuzzer utiliza el mecanismo que emplea Daikon. También se puede lidiar con este problema mejorando la calidad de las test suites. En nuestros experimentos utilizamos Randoop, el cuál podría ser complementado con técnicas automáticas de generación de tests adicionales.

Recall

Recall es el aspecto donde SpecFuzzer supera a GAssert y EvoSpex. Esto es para el ground truth general, y para la mayoría de los subconjuntos de ground truth específicos de cada herramienta. EvoSpex tiene mejor recall que SpecFuzzer para su lenguaje específico; infiere 5 aserciones que SpecFuzzer no. De esas 5, 2 aserciones son descartadas por el selector de invariantes (describimos las razones más arriba). Las tres restantes son soportadas por nuestra gramática, pero actualmente Daikon no puede instanciar estas aserciones (también describimos este problema anteriormente); es decir, estas 3 aserciones no son parte de las 41 que nuestro prototipo soporta actualmente.

La mejora en recall de SpecFuzzer por sobre las otras técnicas hace nuestra herramienta más efectiva en general, como muestran los valores de f1-score. Notemos que SpecFuzzer tiene mejor f1-score comparado con técnicas previas, para el ground truth general, es decir, incluso teniendo en cuenta sus limitaciones de precision, y los problemas en relación al soporte de aserciones.

7.3.6. Amenazas a la Validez

Nuestra evaluación experimental fue realizada en un conjunto de datos construido con casos de estudio de trabajos previos. Tuvimos que estudiar estos casos de estudio manualmente para poder definir las aserciones del ground truth. Para mitigar el riesgo de errores, chequeamos estas aserciones utilizando Microsoft IntelliTest (previamente llamado Pex [87]).

Las amenazas a la validez interna podrían surgir de la aleatoriedad de cada técnica. Para tener en cuenta este problema, evaluamos SpecFuzzer, GAssert y EvoSpex haciendo múltiples ejecuciones en cada método analizado, y reportamos los promedios. Como trabajo extra, planeamos extender la evaluación experimental a proyectos Java de mayor escala, lo que probablemente implicará abandonar el cómputo de las métricas de performance sobre ground truth, debido al esfuerzo que involucraría estudiar proyectos grandes y escribir manualmente aserciones correctas.

7.4. Conclusión

Las especificaciones formales de clase tienen aplicaciones en varias áreas del desarrollo de software, incluyendo diseño de software, bug finding, y comprensión de programas. Técnicas para la inferencia automática de especificaciones de clase han sido propuestas, pero tienen limitaciones, por ejemplo, soportan un número limitado de tipos de aserciones y no son flexibles al cambio. Nosotros presentamos SpecFuzzer, una técnica para inferir especificaciones probables de clases que combina análisis estático, fuzzing basado en gramáticas, y análisis de mutaciones. Nuestra evaluación muestra que SpecFuzzer tiene una performance superior en comparación con las herramientas estado del arte GAssert y EvoSpex, especialmente considerando el recall. Además, el uso de fuzzing basado en gramática permite que SpecFuzzer sea fácilmente adaptado a diferentes lenguajes de aserción.

Este trabajo también abre varias líneas de mejora, dado que hemos identificado algunas limitaciones concretas de nuestro enfoque. El mecanismo basado en mutaciones para descartar aserciones débiles es afectado por la ausencia de operadores de mutación, que podrían permitir que nuestra herramienta detecte algunas propiedades específicas. Otros mecanismos más sofisticados para lidiar con aserciones no falsificadas por ningún mutante también podrían ser incorporados (por ejemplo, técnicas basadas en restric-

ciones). En general, la estructura modular de nuestra técnica nos permite mejorar componentes específicos, por ejemplo, generación de tests (para mejorar la precisión), fuzzing (para considerar técnicas de fuzzing más efectivas/eficientes), etc. Finalmente, limitaciones de implementación en la fase de detección dinámica constituye un cuello de botella para las capacidades de inferencia de aserciones de SpecFuzzer, que planeamos abordar en futuras extensiones de nuestra herramienta.

Los artefactos de evaluación de SpecFuzzer y una descripción detallada sobre como reproducir nuestros experimentos se encuentra públicamente disponible en el siguiente link: **`https://sites.google.com/view/specfuzzer`**.

Capítulo 8

Conclusión

Las especificaciones de software juegan un papel central en varias etapas del desarrollo del software, como ingeniería de requisitos, verificación y mantención de software, etc. En el contexto de análisis de programas, hay una creciente disponibilidad de técnicas poderosas, incluyendo generación de tests [68, 59, 1], bug finding [20, 55], localización de fallas [95, 92], y reparación de programas [91, 46, 71], para las que la disponibilidad de especificaciones se vuelve crucial. Si bien muchas de estas herramientas recurren a tests como especificaciones, en general, podrían beneficiarse de la disponibilidad de especificaciones más fuertes y generales, como las provistas por invariantes y poscondiciones. Estos tipos de especificaciones se están volviendo más comunes en el desarrollo de software, con metodologías que las incorporan [57, 48], y herramientas que pueden aprovecharlas significativamente cuando están disponibles, para realizar distintos tipos de análisis que ayudan a incrementar la calidad y confiabilidad del software.

El problema del oráculo [4] se ha vuelto un problema muy importante en la ingeniería de software, y dentro del contexto del mismo, la derivación o inferencia de especificaciones es particularmente desafiante. En esta tesis, describimos técnicas basadas en aprendizaje y búsqueda/optimización para sintetizar especificaciones a partir de otras especificaciones previamente existentes o del comportamiento actual del software. Más precisamente, las técnicas presentadas se concentran en tres estrategias diferentes: el uso de modelos de aprendizaje automático para capturar el comportamiento de especificaciones; el uso de computación evolutiva para inferir especificaciones; y el uso de fuzzing, también para inferir especificaciones. Las evaluaciones experimentales que realizamos muestran que nuestras técnicas son capaces de

producir especificaciones más precisas (contratos más fuertes en el sentido de [73], con los beneficios asociados allí descritos), en comparación a especificaciones producidas por técnicas relacionadas, especialmente para implementaciones basadas en referencias y de componentes con contratos implícitos fuertes.

8.1. Resumen de las Contribuciones

En resumen, esta tesis hace las siguientes contribuciones:

- Dos técnicas basadas en redes neuronales para capturar el comportamiento de especificaciones. La primera de ellas, usa redes neuronales para aproximar el comportamiento de invariantes de clase (Capítulo 3); la segunda técnica, se enfoca en capturar una especificación adaptada a partir de invariantes de clase, que permite mejorar procesos de ejecución simbólica generalizada (Capítulo 4).

- Dos enfoques basados en computación evolutiva para la inferencia de especificaciones. El primero propone un mecanismo de traducción de especificaciones operacionales, escritas en un lenguaje de programación imperativo, a especificaciones declarativas, escritas en lógica relacional (Capítulo 5); el segundo enfoque tiene como objetivo la inferencia de poscondiciones, haciendo énfasis en propiedades de implementaciones basadas en referencias, de métodos Java (Capítulo 6).

- Una técnica nueva que emplea una combinación de análisis estático, fuzzing basado en gramáticas, y análisis de mutaciones, para inferir especificaciones de clase (precondiciones, poscondiciones e invariantes) en Java (Capítulo 7).

8.2. Revisión y Trabajo Futuro

En esta sección, remarcamos las principales conclusiones a partir de los resultados experimentales de cada técnica presentada en esta tesis, y discutimos algunas líneas futuras de investigación.

8.2.1. Redes Neuronales aproximando Especificaciones

Los Capítulos 3 y 4 presentan dos técnicas basadas en el uso de redes neuronales para aproximar especificaciones. Por un lado, el Capítulo 3 presenta una técnica para inferir un clasificador de instancias de clase, que puede ser utilizado como un invariante de clase; por otro lado, el Capítulo 3 introduce una técnica para aprender un clasificador de heaps parcialmente simbólicos, que puede ser utilizado para identificar heaps parcialmente simbólicos inviables y podar el espacio de búsqueda durante ejecución simbólica generalizada.

Nuestras evaluaciones experimentales de ambos enfoques, basadas principalmente en capturar especificaciones de estructuras de datos de variada complejidad, muestran que las NNs pueden lograr una alta precisión en la tarea de clasificación correspondiente, en comparación con mecanismos relacionados. Esta alta precisión de los modelos de aprendizaje en tareas de clasificación involucrando estructuras de datos con propiedades complejas, también ha sido comprobada en trabajos relacionados (particularmente [90]). Por último, mostramos que utilizar estos clasificadores basados en NNs puede mejorar tareas de bug finding, permitiendo detectar bugs que técnicas relacionadas no pueden. Además, las NNs que clasifican heaps parcialmente simbólicos permiten mejorar considerablemente la ejecución simbólica generalizada basada en lazy initialization.

Estas dos técnicas abren varias líneas para trabajos futuros. Aunque ambas técnicas incorporan mecanismos efectivos de generación automática de los datos de entrenamiento, en algunos casos sufren problemas de escalabilidad. Alternativas o mejoras de estos procesos podrían tener un impacto en la aplicabilidad de estas técnicas. Adicionalmente, Las NNs que construimos utilizan parámetros estándares y su arquitectura es relativamente simple, por lo que explorar el uso de arquitecturas más complejas, incluso empleando técnicas de *feature engineering* [32], podría mejorar el rendimiento de nuestras NNs, especialmente al analizar software más complejo.

8.2.2. Inferencia basada en Computación Evolutiva

La computación evolutiva, más precisamente algoritmos genéticos, es la técnica subyacente a los dos enfoques presentados en los Capítulos 5 y 6. El Capítulo 5 presenta un enfoque basado en un algoritmo genético para traducir una especificación operacional, escrita en un lenguaje de programación imperativo, a una especificación declarativa equivalente, escrita en lógica re-

lacional. El Capítulo 6 propone EvoSpex, un algoritmo genético para la inferencia de poscondiciones de métodos Java, en la forma de aserciones al estilo JML [11]. Ambas técnicas son similares, en el sentido de que evolucionan una población de especificaciones candidatas con el objetivo de encontrar una que sea consistente con elementos de software dado, una especificación en el caso de la traducción, y ejecuciones de un método en el caso de EvoSpex.

Las evaluaciones que realizamos muestran que nuestras técnicas basadas en algoritmos genéticos permiten inferir especificaciones más precisas que técnicas relacionadas, y ayudar a mejorar tareas de análisis. La técnica de traducción fue evaluada en casos de estudio compuestos de estructuras de datos de complejidad variada, donde mostramos que el algoritmo es capaz de producir invariantes de representación declarativos adecuados, a partir de sus contrapartes operacionales. Además, las especificaciones obtenidas son más adecuadas para tareas de análisis, más precisamente para verificación acotada y ejecución simbólica generalizada utilizando lazy initialization, que especificaciones obtenidas mediante traducción con preservación de semántica. Por otro lado, EvoSpex, el algoritmo genético para inferencia de poscondiciones, es capaz de producir aserciones más precisas, con un total de 6.70 % de falsos positivos, comparados con el 17.49 % de falsos positivos de técnicas relacionadas para un conjunto de métodos elegidos aleatoriamente, y de inferir una parte importante de aserciones ricas de programas, tomadas a partir de un conjunto de casos de estudio involucrando contratos para verificación y síntesis de programas.

Nuestros algoritmos genéticos para inferencia de especificaciones abren también varias líneas de trabajo futuro. En ambos casos, nos concentramos principalmente en propiedades de implementaciones basadas en referencias, lo que limita la aplicabilidad de las técnicas (propiedades aritméticas complejas suponen una limitación). Una línea obvia de investigación es trabajar en una generalización de nuestros enfoques, de modo que permitan aprender un conjunto más rico de especificaciones. También trabajamos con conjuntos finitos de operadores genéticos; extender este conjunto de operadores y explorar algunos nuevos mecanismos podría ser necesario para incrementar el tipo de propiedades que los algoritmos son capaces de producir, especialmente al momento de lidiar con programas más sofisticados. Finalmente, las funciones de fitness de los algoritmos genéticos juegan un papel crucial en la calidad de las soluciones; adaptar la función de fitness de nuestro algoritmo para priorizar aspectos generales de las poscondiciones de métodos podría mejorar considerablemente nuestros resultados.

8.2.3. Inferencia basada en Fuzzing

El Capítulo 7 presenta SpecFuzzer, nuestra última técnica para la inferencia de especificaciones. SpecFuzzer usa una combinación de análisis estático, fuzzing basado en gramáticas, y análisis de mutación para inferir especificaciones de clase. Esencialmente, SpecFuzzer utiliza análisis estático para extraer una gramática que captura un lenguaje de especificaciones a partir de una clase objetivo; fuzzing basado en gramáticas para generar especificaciones candidatas que son provistas a un detector dinámico de invariantes; y finalmente análisis de mutaciones para eliminar especificaciones relevantes y equivalentes. Aunque varias técnicas automáticas de inferencia han sido propuestas, sufren algunas limitaciones, por ejemplo, soportan un número limitado de tipos de aserciones y no son flexibles al cambio. Nuestra evaluación muestra que SpecFuzzer tiene una performance superior en comparación con herramientas estado del arte, especialmente considerando la detección de propiedades relevantes. Además, el uso de fuzzing basado en gramáticas permite que SpecFuzzer pueda ser fácilmente adaptado a diferentes lenguajes de aserción.

Dado que hemos identificado algunas limitaciones concretas en nuestro enfoque, existen algunas líneas claras de investigación para mejorarlo. Por un lado, el uso de mutaciones para descartar aserciones débiles, en ocasiones es afectado por la ausencia de operadores de mutaciones, que impiden que la herramienta infiera propiedades deseadas. Otros mecanismos más sofisticados para lidiar con aserciones no falsificadas por ningún mutante también podrían ser incorporados (por ejemplo, técnicas basadas en restricciones). En general, la estructura modular de nuestra técnica nos permite mejorar componentes específicos, por ejemplo, generación de tests (para mejorar la precisión), fuzzing (para considerar técnicas de fuzzing más efectivas/eficientes), etc. Finalmente, en futuras extensiones de nuestra herramienta, planeamos abordar algunas limitaciones de implementación en la fase de detección dinámica, que afectan actualmente las capacidades de inferencia de aserciones de SpecFuzzer.

Bibliografía

[1] Pablo Abad, Nazareno Aguirre, Valeria S. Bengolea, Daniel Ciolek, Marcelo F. Frias, Juan P. Galeotti, Tom Maibaum, Mariano M. Moscato, Nicolás Rosner, and Ignacio Vissani, *Improving test generation under rich contracts by tight bounds and incremental SAT solving*, Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013, IEEE Computer Society, 2013, pp. 21–30. 3.4, 8

[2] Paul Ammann and Jeff Offutt, *Introduction to software testing*, 1 ed., Cambridge University Press, USA, 2008. 1.2

[3] Mike Barnett, *Code contracts for .net: Runtime verification and so much more*, Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings (Howard Barringer, Yliès Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon J. Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann, eds.), Lecture Notes in Computer Science, vol. 6418, Springer, 2010, pp. 16–17. 2.2.2

[4] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo, *The oracle problem in software testing: A survey*, IEEE Trans. Software Eng. **41** (2015), no. 5, 507–525. 1, 1.1, 6.4, 8

[5] James Bergstra and Yoshua Bengio, *Random search for hyper-parameter optimization*, J. Mach. Learn. Res. **13** (2012), 281–305. 3.2.3, 4.2.3

[6] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov, *Korat: automated testing based on java predicates*, Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2002, Roma, Italy, July 22-24, 2002 (Phyllis G. Frankl, ed.), ACM, 2002, pp. 123–133. 3.2.2, 3.3.1, 3.3.2, 12, 4.2.2, 5.1, 5.2.2, 5.3.1, 6.1, 6.2.1

[7] Pietro Braione, Giovanni Denaro, Andrea Mattavelli, and Mauro Pezzè, *Combining symbolic execution and search-based testing for programs with complex heap inputs*, Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017 (Tevfik Bultan and Koushik Sen, eds.), ACM, 2017, pp. 90–101. 2.4.2

[8] Lilian Burdy, Yoonsik Cheon, David Cok, Michael D. Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan, M. Leino, and Erik Poll, *An overview of jml tools and applications*, Electronic Notes in Theoretical Computer Science **80** (2003), 75 – 91, Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS'03). 5.1

[9] Jacob Burnim, Sudeep Juvekar, and Koushik Sen, *WISE: automated test generation for worst-case complexity*, 31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings, 2009, pp. 463–473. 2.4.2

[10] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang, *Compositional shape analysis by means of bi-abduction*, J. ACM **58** (2011), no. 6. 2.4.1

[11] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll, *Beyond assertions: Advanced specification and verification with JML and esc/java2*, Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures (Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, eds.), Lecture Notes in Computer Science, vol. 4111, Springer, 2005, pp. 342–363. 1.2, 2.2.2, 3.1, 7.1, 7.2.2, 8.2.2

[12] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel, *Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact*, Empirical Software Engineering **10** (2005), no. 4, 405–435. 3.3, 3.3.3

[13] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao, *The daikon system for dynamic detection of likely invariants*, Sci. Comput. Program. **69** (2007), no. 1-3, 35–45. 1.1.1, 2.3.3, 3.3, 3.3.2, 3.4, 5.3.5, 6.1, 7.1.1

[14] Robert W. Floyd, *Assigning meanings to programs*, Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics 19 (Providence) (J. T. Schwartz, ed.), American Mathematical Society, 1967, pp. 19–32. 1.1, 2.2.2, 5.1

[15] Gordon Fraser and Andrea Arcuri, *Evosuite: automatic test suite generation for object-oriented software*, SIGSOFT FSE, ACM, 2011, pp. 416–419. 1.1.1, 9

[16] ———, *A large-scale evaluation of automated unit test generation using evosuite*, ACM Trans. Softw. Eng. Methodol. **24** (2014), no. 2, 8:1–8:42. 6.3.1

[17] Gordon Fraser and Andreas Zeller, *Mutation-driven generation of unit tests and oracles*, ISSTA, ACM, 2010, pp. 147–158. 7.2.5

[18] Marcelo F. Frias, Juan P. Galeotti, Carlos López Pombo, and Nazareno Aguirre, *Dynalloy: upgrading alloy with actions*, 27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA (Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh, eds.), ACM, 2005, pp. 442–451. 5.3.4

[19] Juan P. Galeotti, Nicolás Rosner, Carlos Gustavo López Pombo, and Marcelo F. Frias, *TACO: efficient sat-based bounded verification using symmetry breaking and tight bounds*, IEEE Trans. Software Eng. **39** (2013), no. 9, 1283–1307. 5.1

[20] Juan P. Galeotti, Nicolás Rosner, Carlos López Pombo, and Marcelo F. Frias, *Analysis of invariants for efficient bounded verification*, Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, 2010 (Paolo Tonella and Alessandro Orso, eds.), ACM, 2010, pp. 25–36. 3.3, 3.3.3, 3.4, 4.1, 5.1, 5.3.4, 8

[21] Jaco Geldenhuys, Nazareno Aguirre, Marcelo F. Frias, and Willem Visser, *Bounded lazy initialization*, NASA Formal Methods, 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings (Guillaume Brat, Neha Rungta, and Arnaud Venet, eds.), Lecture Notes in Computer Science, vol. 7871, Springer, 2013, pp. 229–243. 5.3.4

[22] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli, *Fundamentals of software engineering*, 2nd ed., Prentice Hall PTR, Upper Saddle River, NJ, USA, 2002. 1, 2.2

[23] Adam Gibson, Chris Nicholson, Josh Patterson, Melanie Warrick, Alex D. Black, Vyacheslav Kokorin, Samuel Audet, and Susan Eraly, *Deeplearning4j: Distributed, open-source deep learning for java and scala on hadoop and spark*, May 2016. 4.2.3

[24] Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov, *Test generation through programming in UDITA*, Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010 (Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel, eds.), ACM, 2010, pp. 225–234. 3.2.1, 3.2.1

[25] Patrice Godefroid, Nils Klarlund, and Koushik Sen, *DART: directed automated random testing*, Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005, 2005, pp. 213–223. 2.4.2

[26] David Goldberg, *Genetic algorithms in search, optimization and machine learning*, Addison-Wesley, 1989. 2.3.2

[27] Ian Goodfellow, Yoshua Bengio, and Aaron Courville, *Deep learning*, MIT Press, 2016, `http://www.deeplearningbook.org`. 2.3.1, 2.3.1

[28] Google, *Google's afl*, 2021. 2.3.3

[29] ———, *Google's clusterfuzz*, 2021. 2.3.3

[30] ———, *Google's oss-fuzz*, 2021. 2.3.3

[31] Ashutosh Gupta and Andrey Rybalchenko, *Invgen: An efficient invariant generator*, Proceedings of the 21st International Conference on Computer Aided Verification (Berlin, Heidelberg), CAV '09, Springer-Verlag, 2009, p. 634–640. 1.1.1

[32] Isabelle Guyon and André Elisseeff, *An introduction to variable and feature selection*, Journal of Machine Learning Research **3** (2003), 1157–1182. 3.4, 8.2.1

[33] Charles A. R. Hoare, *An axiomatic basis for computer programming*, Commun. ACM **12** (1969), no. 10, 576–580. 2.2.2, 5.1

[34] ———, *Towards the verifying compiler*, From Object-Orientation to Formal Methods, Essays in Memory of Ole-Johan Dahl (Olaf Owe, Stein Krogdahl, and Tom Lyche, eds.), Lecture Notes in Computer Science, vol. 2635, Springer, 2004, pp. 124–136. 1.1

[35] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy, *Grammarinator: A grammar-based open source fuzzer*, Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation (New York, NY, USA), A-TEST 2018, Association for Computing Machinery, 2018, p. 45–48. 2.3.3, 7.2.3

[36] Daniel Jackson, *Software abstractions - logic, language, and analysis*, MIT Press, 2006. 5.1, 5.1, 5.2.1, 6.2.3

[37] Gunel Jahangirova, David Clark, Mark Harman, and Paolo Tonella, *Test oracle assessment and improvement*, Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016 (Andreas Zeller and Abhik Roychoudhury, eds.), ACM, 2016, pp. 247–258. 1, 1.2, 9, 6.3.1, 6.3.1, 6.3.3, 7.1.1, 7.3.5

[38] Pankaj Jalote, *An integrated approach to software engineering, third edition*, Texts in Computer Science, Springer, 2005. 1

[39] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani, *An introduction to statistical learning: With applications in r*, Springer, 2014. 4.3.1

[40] Kenneth A. De Jong, *Evolutionary computation - a unified approach*, MIT Press, 2006. 1.2, 1.2, 9

[41] René Just, Franz Schweiggert, and Gregory M. Kapfhammer, *MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler*, Proceedings of the International Conference on Automated Software Engineering (ASE), November 9–11 2011, pp. 612–615. 7.3.2

[42] Shadi Abdul Khalek, Guowei Yang, Lingming Zhang, Darko Marinov, and Sarfraz Khurshid, *Testera: A tool for testing java programs using*

alloy specifications, 26th IEEE/ACM International Conference on Automated Software Engineering, ASE 2011, Lawrence, KS, USA, November 6-10, 2011 (Perry Alexander, Corina S. Pasareanu, and John G. Hosking, eds.), IEEE Computer Society, 2011, pp. 608–611. 6.1

[43] Sarfraz Khurshid, Corina S. Pasareanu, and Willem Visser, *Generalized symbolic execution for model checking and testing*, Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings (Hubert Garavel and John Hatcliff, eds.), Lecture Notes in Computer Science, vol. 2619, Springer, 2003, pp. 553–568. 2.4.2, 2.4.2, 11, 3.2.1, 3.2.1

[44] James C. King, *Symbolic execution and program testing*, Commun. ACM **19** (1976), no. 7, 385–394. 1.2, 2.4.2

[45] Daniel Kroening and Michael Tautschnig, *Cbmc – c bounded model checker*, Tools and Algorithms for the Construction and Analysis of Systems (Berlin, Heidelberg) (Erika Ábrahám and Klaus Havelund, eds.), Springer Berlin Heidelberg, 2014, pp. 389–391. 5.1

[46] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer, *Genprog: A generic method for automatic software repair*, IEEE Trans. Software Eng. **38** (2012), no. 1, 54–72. 3.4, 8

[47] Barbara Liskov and John Guttag, *Program development in java: Abstraction, specification, and object-oriented design*, 1st ed., Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000. 5.1

[48] Barbara Liskov and John V. Guttag, *Program development in java - abstraction, specification, and object-oriented design*, Addison-Wesley, 2001. 2.2.1, 3.4, 4.2.1, 8

[49] Calvin Loncaric, Michael D. Ernst, and Emina Torlak, *Generalized data structure synthesis*, Proceedings of the 40th International Conference on Software Engineering (New York, NY, USA), ICSE '18, Association for Computing Machinery, 2018, p. 958–968. 6.3.2

[50] Victoria López, Alberto Fernández, and Francisco Herrera, *On the importance of the validation technique for classification with imbalanced*

datasets: Addressing covariate shift when data is skewed, Inf. Sci. **257** (2014), 1–13. 4.3.1

[51] Kasper Søe Luckow, Rody Kersten, and Corina S. Pasareanu, *Symbolic complexity analysis using context-preserving histories*, 2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017, 2017, pp. 58–68. 4.1

[52] ———, *Complexity vulnerability analysis using symbolic execution*, Softw. Test. Verification Reliab. **30** (2020), no. 7-8. 2.4.2

[53] Muhammad Zubair Malik, Aman Pervaiz, and Sarfraz Khurshid, *Generating representation invariants of structurally complex data*, Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings (Orna Grumberg and Michael Huth, eds.), Lecture Notes in Computer Science, vol. 4424, Springer, 2007, pp. 34–49. 5.3.5

[54] 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), 1–1. 1.2, 1.2, 2.3.3

[55] Florian Merz, Stephan Falke, and Carsten Sinz, *LLBMC: bounded model checking of C and C++ programs using a compiler IR*, Verified Software: Theories, Tools, Experiments - 4th International Conference, VSTTE 2012, Philadelphia, PA, USA, January 28-29, 2012. Proceedings (Rajeev Joshi, Peter Müller, and Andreas Podelski, eds.), Lecture Notes in Computer Science, vol. 7152, Springer, 2012, pp. 146–161. 3.2.1, 3.4, 8

[56] Bertrand Meyer, *Applying "design by contract"*, IEEE Computer **25** (1992), no. 10, 40–51. 1

[57] ———, *Object-oriented software construction, 2nd edition*, Prentice-Hall, 1997. 2.2, 2.2.2, 3.4, 6.3.2, 8

[58] ———, *Design by contract: The eiffel method*, TOOLS 1998: 26th International Conference on Technology of Object-Oriented Languages and Systems, 3-7 August 1998, Santa Barbara, CA, USA, IEEE Computer Society, 1998, p. 446. 2.2.2, 6.3.2

[59] Bertrand Meyer, Ilinca Ciupa, Andreas Leitner, and Lisa Ling Liu, *Automatic testing of object-oriented software*, SOFSEM 2007: Theory and Practice of Computer Science, 33rd Conference on Current Trends in Theory and Practice of Computer Science, Harrachov, Czech Republic, January 20-26, 2007, Proceedings (Jan van Leeuwen, Giuseppe F. Italiano, Wiebe van der Hoek, Christoph Meinel, Harald Sack, and Frantisek Plasil, eds.), Lecture Notes in Computer Science, vol. 4362, Springer, 2007, pp. 114–129. 3.2.1, 3.4, 8

[60] Barton P. Miller, *Fuzz testing of application reliability*, 2021. 2.3.3

[61] Facundo Molina, César Cornejo, Renzo Degiovanni, Germán Regis, Pablo F. Castro, Nazareno Aguirre, and Marcelo F. Frias, *An evolutionary approach to translating operational specifications into declarative specifications*, Science of Computer Programming **181** (2019), 47–63. 1.3, 5, 6.2.2, 6.2.3, 6.2.5

[62] Facundo Molina, Marcelo d'Amorim, and Nazareno Aguirre, *Fuzzing class specifications*, 2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE), 2022. 7

[63] Facundo Molina, Renzo Degiovanni, Pablo Ponzio, Germán Regis, Nazareno Aguirre, and Marcelo F. Frias, *Training binary classifiers as data structure invariants*, Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019 (Joanne M. Atlee, Tevfik Bultan, and Jon Whittle, eds.), IEEE / ACM, 2019, pp. 759–770. 1.3, 3.2.3

[64] Facundo Molina, Pablo Ponzio, Nazareno Aguirre, and Marcelo Frias, *Evospex: An evolutionary algorithm for learning postconditions*, 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), 2021, pp. 1223–1235. 1.3, 6, 7.1.1, 7.2.5, 7.3, 7.3.1, 7.3.2

[65] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli, *Solving SAT and SAT modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(*T*)*, J. ACM **53** (2006), no. 6, 937–977. 2.4.2, 11

[66] Aditya V. Nori, Sriram K. Rajamani, SaiDeep Tetali, and Aditya V. Thakur, *The yogiproject: Software property checking via static analysis*

and testing, Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings (Stefan Kowalewski and Anna Philippou, eds.), Lecture Notes in Computer Science, vol. 5505, Springer, 2009, pp. 178–181. 3.2.1

[67] P. S. Novikov, *Elements of mathematical logic*, Reading, Mass., Addison-Wesley Pub. Co., 1964. 7.2.5

[68] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball, *Feedback-directed random test generation*, 29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007, IEEE Computer Society, 2007, pp. 75–84. 1.1.1, 1.2, 3.1, 3.2.1, 3.3.1, 3.4, 12, 5.3.5, 6.2.1, 7.3.2, 8

[69] Corina S. Pasareanu and Willem Visser, *Verification of java programs using symbolic execution and invariant generation*, Model Checking Software, 11th International SPIN Workshop, Barcelona, Spain, April 1-3, 2004, Proceedings, 2004, pp. 164–181. 2.4.2

[70] Corina S. Pasareanu, Willem Visser, David H. Bushnell, Jaco Geldenhuys, Peter C. Mehlitz, and Neha Rungta, *Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis*, Autom. Softw. Eng. **20** (2013), no. 3, 391–425. 4.3

[71] Yu Pei, Carlo A. Furia, Martin Nordio, Yi Wei, Bertrand Meyer, and Andreas Zeller, *Automated fixing of programs with contracts*, IEEE Trans. Software Eng. **40** (2014), no. 5, 427–449. 3.4, 8

[72] Nadia Polikarpova, Carlo A. Furia, and Bertrand Meyer, *Specifying reusable components*, Verified Software: Theories, Tools, Experiments, Third International Conference, VSTTE 2010, Edinburgh, UK, August 16-19, 2010. Proceedings (Gary T. Leavens, Peter W. O'Hearn, and Sriram K. Rajamani, eds.), Lecture Notes in Computer Science, vol. 6217, Springer, 2010, pp. 127–141. 6.3.2

[73] Nadia Polikarpova, Carlo A. Furia, Yu Pei, Yi Wei, and Bertrand Meyer, *What good are strong specifications?*, 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26,

2013 (David Notkin, Betty H. C. Cheng, and Klaus Pohl, eds.), IEEE Computer Society, 2013, pp. 262–271. 6.4, 8

[74] Pablo Ponzio, Nazareno Aguirre, Marcelo F. Frias, and Willem Visser, *Field-exhaustive testing*, Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016 (Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su, eds.), ACM, 2016, pp. 908–919. 5.2.4

[75] Pablo Ponzio, Valeria S. Bengolea, Simón Gutiérrez Brida, Gastón Scilingo, Nazareno Aguirre, and Marcelo F. Frias, *On the effect of object redundancy elimination in randomly testing collection classes*, Proceedings of the 11th International Workshop on Search-Based Software Testing, ICSE 2018, Gothenburg, Sweden, May 28-29, 2018 (Juan Pablo Galeotti and Alessandra Gorla, eds.), ACM, 2018, pp. 67–70. 6.2.1

[76] John C Reynolds, *Separation logic: A logic for shared mutable data structures*, Proceedings 17th Annual IEEE Symposium on Logic in Computer Science, IEEE, 2002, pp. 55–74. 2.4.1

[77] Nicolás Rosner, Jaco Geldenhuys, Nazareno Aguirre, Willem Visser, and Marcelo F. Frias, *BLISS: improved symbolic execution by bounded lazy initialization with SAT support*, IEEE Trans. Software Eng. **41** (2015), no. 7, 639–660. 4.3.2, 5.3.4

[78] Stuart J. Russell and Peter Norvig, *Artificial intelligence - A modern approach (3. internat. ed.)*, Pearson Education, 2010. 1.2, 1.2, 2.3.1, 2.3.1, 3.3

[79] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini, *The graph neural network model*, IEEE Trans. Neural Networks **20** (2009), no. 1, 61–80. 4.2.2

[80] Todd W. Schiller, Kellen Donohue, Forrest Coward, and Michael D. Ernst, *Case studies and tools for contract specifications*, Proceedings of the 36th International Conference on Software Engineering (New York, NY, USA), ICSE 2014, Association for Computing Machinery, 2014, p. 596–607. 2.2

[81] Todd W. Schiller and Michael D. Ernst, *Reducing the barriers to writing verified specifications*, Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012 (Gary T. Leavens and Matthew B. Dwyer, eds.), ACM, 2012, pp. 95–112. 1, 1.2, 1.2, 2.4

[82] Anthony J. H. Simons, *Jwalk: a tool for lazy, systematic testing of java classes by design introspection and user interaction*, Autom. Softw. Eng. **14** (2007), no. 4, 369–418. 1.1.1, 3.4

[83] Mozhan Soltani, Annibale Panichella, and Arie van Deursen, *A guided genetic algorithm for automated crash reproduction*, Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017, IEEE / ACM, 2017, pp. 209–220. 9

[84] Kazuo Sugihara, *Measures for performance evaluation of genetic algorithms (extended abstract)*, Proc. 3rd Joint Conference on Information Sciences (JCIS '97, 1997, pp. 172–175. 5.3.3

[85] Valerio Terragni, Gunel Jahangirova, Paolo Tonella, and Mauro Pezzè, *Evolutionary improvement of assertion oracles*, Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (New York, NY, USA), ESEC/FSE 2020, Association for Computing Machinery, 2020, p. 1178–1189. 7.1.1, 7.2.5, 7.3, 7.3.1, 7.3.2

[86] Alaa Tharwat, *Classification assessment methods*, Applied Computing and Informatics (2018). 4.3.1

[87] Nikolai Tillmann and Jonathan de Halleux, *Pex-white box test generation for .net*, Tests and Proofs - 2nd International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings (Bernhard Beckert and Reiner Hähnle, eds.), Lecture Notes in Computer Science, vol. 4966, Springer, 2008, pp. 134–153. 7.3.6

[88] Marco Trudel, Manuel Oriol, Carlo A. Furia, and Martin Nordio, *Automated translation of java source code to eiffel*, Objects, Models, Components, Patterns - 49th International Conference, TOOLS 2011, Zurich,

Switzerland, June 28-30, 2011. Proceedings (Judith Bishop and Antonio Vallecillo, eds.), Lecture Notes in Computer Science, vol. 6705, Springer, 2011, pp. 20–35. 6.3.2

[89] Julian Tschannen, Carlo A. Furia, Martin Nordio, and Nadia Polikarpova, *Autoproof: Auto-active functional verification of object-oriented programs*, 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2015, London, UK, April 11-18, 2015, Lecture Notes in Computer Science, vol. 9035, Springer, 2015, pp. 566–580. 6.3.2

[90] Muhammad Usman, Wenxi Wang, Kaiyuan Wang, Cagdas Yelen, Nima Dini, and Sarfraz Khurshid, *A study of learning data structure invariants using off-the-shelf tools*, Model Checking Software - 26th International Symposium, SPIN 2019, Beijing, China, July 15-16, 2019, Proceedings (Fabrizio Biondi, Thomas Given-Wilson, and Axel Legay, eds.), Lecture Notes in Computer Science, vol. 11636, Springer, 2019, pp. 226–243. 4.3.1, 8.2.1

[91] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest, *Automatically finding patches using genetic programming*, 31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings, IEEE, 2009, pp. 364–374. 3.4, 8

[92] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa, *A survey on software fault localization*, IEEE Trans. Software Eng. **42** (2016), no. 8, 707–740. 3.4, 8

[93] Razieh Nokhbeh Zaeem, Divya Gopinath, Sarfraz Khurshid, and Kathryn S. McKinley, *History-aware data structure repair using SAT*, Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings (Cormac Flanagan and Barbara König, eds.), Lecture Notes in Computer Science, vol. 7214, Springer, 2012, pp. 2–17. 3.3, 3.3.3, 3.3.4

[94] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler, *The fuzzing book*, The Fuzzing Book, Saarland University, 2019, Retrieved 2019-09-09 16:42:54+02:00. 7.2.3

[95] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta, *Locating faults through automated predicate switching*, 28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006 (Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa, eds.), ACM, 2006, pp. 272–281. 3.4, 8