



Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

**Cómputo automático de workarounds
transitorios a partir de especificaciones de
programas usando SAT solving**

Tesis para optar por el título de Doctor de la Universidad de
Buenos Aires en el área Ciencias de la Computación

Marcelo Ariel Uva

Director: **Dr. Pablo Daniel Ponzio**

Codirector: **Dr. Carlos Gustavo López Pombo**

Consejero de Estudios: **Dr. Juan Pablo Galeotti**

Lugar de Trabajo: Departamento de Computación. Facultad de
Ciencias Exactas Físico-Químicas y Naturales. Universidad
Nacional de Río Cuarto. Río Cuarto, 2022

Cómputo automático de workarounds transitorios a partir de especificaciones de programas usando SAT solving

Resumen: En muchas situaciones, el software puede recuperarse de fallas en tiempo de ejecución usando *workarounds*. Un *workaround* se define como una alternativa de ejecución para un método defectuoso que permite mantener al sistema funcionando luego de la ocurrencia de una falla.

En este trabajo se presentan dos técnicas para el cómputo automático de *workarounds*. Estas técnicas emplean especificaciones formales (pre y post-condiciones de métodos, e invariantes de clases), y usan SAT solving como procedimiento de decisión. A diferencia de enfoques previos, las técnicas emplean también el estado de ejecución del sistema al momento de la falla, por lo que decimos que los *workarounds* computados por las mismas son *transitorios*.

En la primera técnica, los *workarounds transitorios* consisten de una secuencia de rutinas que satisfacen la especificación del método defectuoso, para el estado en que se produjo la falla. Se propone también un enfoque para generalizar estos *workarounds transitorios* a *esquemas de workarounds*, que sirven para acelerar la búsqueda de *workarounds transitorios* para otros estados que producen fallas en el método defectuoso.

La segunda técnica, propone la construcción de *workarounds transitorios* usando SAT solving para generar directamente un estado que satisface la postcondición de la rutina defectuosa, a partir del estado donde se produjo la falla. Para optimizar esta técnica se propone usar *predicados de rotura de simetrías* y *cotas ajustadas* (técnicas del estado del arte para mejorar la eficiencia y la escalabilidad de SAT solving), y evaluar los beneficios que estos enfoques pueden brindar en el cómputo de *workarounds*.

Las técnicas propuestas se evaluaron experimentalmente sobre una familia de casos de estudio, que incluyen implementaciones de colecciones (con estructuras de datos de distinta complejidad) y una biblioteca para aritmética de fechas. Los resultados muestran que las técnicas propuestas permiten computar *workarounds transitorios* en un número importante de casos, y en tiempos razonables para su aplicación a la recuperación de fallas de software en tiempo de ejecución.

Palabras clave: Recuperación en tiempo de ejecución, Workarounds, Especificaciones formales, SAT solving.

Automatic computation of transient workarounds from program specifications using SAT solving

Abstract: Software failures, produced by errors in source code, can often be bypassed in run time by using the so called workarounds: execution alternatives that the system can use in place of faulty routine to circumvent the failure.

In this thesis, we present two techniques for the automated computation of workarounds from Java code equipped with formal specifications (consisting of method pre and postconditions and class invariants), using SAT solving. As opposed to previous approaches, these techniques make use of the state of the system where the faulty method was executed, hence the computed workarounds are transient.

In the first technique, transient workarounds are defined as an alternative set of routines that satisfy the specification of the faulty method at the given state. We propose a mechanism to generalize these transient workarounds to schemas, which can be used to speed up the search for transient workarounds in other failing states.

The second technique directly exploits SAT solving to circumvent the failing method, automatically building a state that mimics its correct behaviour (described by the method's specification) when starting at the given state. In order to optimize and improve scalability of the second technique we compute and add symmetry breaking predicates and tight field bounds to optimize the SAT process.

For the purpose of experimentally assessing the presented techniques, we develop a number of case studies based on contract-equipped collection classes and a Java library for date arithmetic. The results of the assessment show that the techniques can effectively compute workarounds from complex contracts in an important number of cases, in times that makes them feasible to be used for run time repairs.

Keywords: Runtime recovery, Workarounds, Formal Specifications, SAT solving.

Agradecimientos

Son muchas las personas que me han ayudado de una u otra manera a concluir esta etapa la cual he disfrutado mucho.

En primer lugar quiero agradecer a Pablo Ponzio por su enorme trabajo y dedicación en la dirección de esta tesis. En Pablo he encontrado no sólo a un excelente investigador en esta Ciencia sino también a un gran amigo y persona.

Al igual que Pablo, quiero agradecer del mismo modo a Nazareno Aguirre por todos sus consejos, entrega, generosidad, dedicación y excelencia en su trabajo. Naza es una persona que siempre te ayuda a crecer.

Gracias a Charly y J.P. Galeotti por su acompañamiento y colaboración en todo este proceso.

Gracias a Marcelo Frías por todos sus aportes enriquecedores.

Gracias a todos los compañeros y amigos del Depto. de Computación: Germán, Simón, Gastón, Ceci, Pablo C., Vale, Marcelo, Marcela, Franco, Ariel, Dani, Luciano, cacho, Cesar, Marta, Sonia y seguramente me estoy olvidando de gente!!!

Dedicatoria

Este logro es para mis hijos Juan Cruz y María Victoria (que amo con locura) de los cuales aprendo todos los días a nunca darme por vencido, a luchar siempre.

Este logro es para vos Lucina, flaquita de mi alma, porque “sos mi amor mi cómplice y todo”.

Este logro es para toda mi familia especialmente para Rosa y Pedro.

Índice general

1. Introducción	2
1.1. Antecedentes	3
1.1.1. Workarounds	3
1.1.2. Especificaciones formales	5
1.1.3. Herramientas de análisis basadas en Alloy	6
1.2. Objetivos y Contribuciones	7
1.3. Organización	11
2. Preliminares	12
2.1. Análisis basados en satisfactibilidad booleana (SAT)	12
2.2. Alloy	13
2.3. Dynalloy	17
2.4. Java Modeling Language (JML)	22
2.5. Traducción de especificaciones JML a Alloy	26
2.6. Workarounds	35
2.7. Workarounds automáticos sobre aplicaciones Java	36
3. Cómputo de secuencias de rutinas para workarounds transitorios	37
3.1. Codificación del estado inicial en Alloy	38
3.2. Acciones atómicas DynAlloy a partir de especificaciones de métodos Java	40
3.2.1. Generación de parámetros de tipos primitivos	44
3.3. Programa DynAlloy para el cómputo de <i>workarounds</i>	47
3.4. Algoritmo de cómputo de workarounds transitorios	50
3.5. Ejemplo completo de búsqueda de workarounds	54
3.6. Esquemas de workarounds	58

3.6.1.	Cómputo de <i>workarounds transitorios</i> aplicando <i>esquemas de workarounds</i>	68
4.	<i>Workarounds</i> a través de estados de recuperación	70
4.1.	Cómputo de estados de recuperación	71
4.1.1.	Un ejemplo de cómputo de estados de recuperación	73
4.2.	Rotura de Simetrías	76
4.2.1.	Cómputo de predicados de rotura de simetrías	78
4.3.	Cotas ajustadas	88
4.3.1.	Algoritmo BOTTOM-UP para cómputo de cotas ajustadas	93
5.	Evaluación experimental	96
5.1.	Evaluación de técnica 1: Cómputo de secuencias de métodos para <i>workarounds transitorios</i>	99
5.1.1.	Evaluación de <i>esquemas de workarounds</i>	108
5.2.	Workarounds transitorios versus permanentes	111
5.3.	Evaluación de técnica 2: Generación de estados de recuperación	117
6.	Trabajos relacionados	128
6.1.	Redundancia en el software	128
6.2.	<i>Workarounds</i> para aplicaciones Web	129
6.3.	Secuencias equivalentes	131
6.4.	Reparación de estructuras de datos	132
6.5.	Análisis de código basado de SAT solving	134
7.	Conclusiones y trabajos futuros	136

Índice de figuras

2.1. Signaturas de <i>listas simplemente encadenadas</i>	14
2.2. Relaciones <code>head</code> , <code>size</code> , <code>data</code> y <code>next</code>	14
2.3. <i>facts</i> para <i>listas simplemente encadenadas</i>	15
2.4. Predicados <code>getFirst</code> y <code>getLast</code>	16
2.5. Aserción <code>getFirstEqGetLast</code>	17
2.6. Instancia del predicado <code>getFirstEqGetLast</code>	17
2.7. Acción atómica <code>clear</code>	19
2.8. Acción atómica de <code>asignación</code>	20
2.9. Ejemplo de <i>act</i> , <i>program</i> y <i>assertCorrectness</i> para <i>listas</i> . . .	20
2.10. Gramática DynAlloy	21
2.11. Clases <code>Node.java</code> y <code>LinkedList.java</code>	23
2.12. Función Alloy <code>fun_reach</code> para <code>LinkedList</code>	27
2.13. Signaturas <code>Null</code> , <code>LinkedList</code> y <code>Node</code>	27
2.14. (a) Cláusula JML <code>invariant</code> y (b) su traducción al predicado <code>invariant_LinkedList</code> para <code>LinkedList</code>	28
2.15. (a) Campo de la especificación <code>myseq</code> y (b) su correspondiente traducción al predicado <code>abstraction_sequence</code>	31
2.16. (a) Especificación JML para <code>set(index,e)</code> y (b) su traducción a Alloy.	34
3.1. Escenario de falla para el método <code>set(index,elem)</code>	38
3.2. Estado de ejecución Java.	39
3.3. Signaturas <code>LinkedList</code> y <code>Node</code>	39
3.4. <code>initialState</code> para <code>lis.set(i,v)</code>	40
3.5. Acción atómica a_i asociada al método m_i	41
3.6. (a) Contrato Alloy para <code>set(index,e)</code> y (b) su acción atómica <code>set</code>	42
3.7. Signatura <code>ActionExecuted</code>	43

3.8. (a) Contrato Alloy para <code>clear</code> y (b) su acción atómica <code>clear</code> .	44
3.9. (a) Contrato Alloy para <code>contains(e)</code> y (b) su acción atómica <code>contains</code> .	45
3.10. Acción <code>nonDetInt</code>	46
3.11. Acción <code>nonDetBool</code>	46
3.12. Programa de cómputo de <i>workarounds transitorios</i> para a_i	48
3.13. (a) Estado de invocación de <code>set(0,4)</code> y (b) estado esperado.	55
3.14. Estado de invocación para <code>lis.set(0,4)</code>	55
3.15. Programa de recuperación para <code>lis.set(0,4)</code>	57
3.16. (a) Estado de invocación para <code>set(1,3)</code> y (b) estado esperado.	59
3.17. Esquema de <i>workaround</i> para el método <code>set(index,elem)</code>	60
3.18. <i>Esquema de workaround</i> compuesto	61
3.19. (a) Estado de invocación para <code>boolean r = lis.contains(-7)</code> y (b) estado esperado.	61
3.20. (a) Estado de invocación para <code>boolean r = lis.contains(3)</code> y (b) estado esperado.	62
3.21. Esquema de <i>workaround</i> compuesto para el método <code>contains(e)</code>	64
3.22. Esquema de <i>workaround</i> compuesto para el método <code>isEmpty()</code>	65
3.23. Búsqueda de <i>workarounds transitorios</i> con <i>esquemas de workarounds</i> .	68
4.1. Predicado <code>recoveryState</code> para el método m_i	72
4.2. Estado de invocación s_i para <code>set(1,4)</code>	73
4.3. Estado de invocación para <code>set(1,4)</code>	74
4.4. Predicado <code>recoveryState</code> de recuperación para <code>set(1,4)</code>	75
4.5. Estado s_f para <code>set(1,4)</code>	75
4.6. Tres listas isomorfas	77
4.7. Especificación Alloy para <code>BinTree</code>	78
4.8. Funciones auxiliares <code>nextTNode</code> , <code>minTNode</code> y <code>prevstNode</code>	80
4.9. Campos generados para <code>BinTree</code>	80
4.10. Restricciones a campos generados para <code>BinTree</code>	81
4.11. Función <code>FReach</code> para <code>BinTree</code>	81
4.12. Funciones <code>globalMin</code> y <code>parentMin</code> para <code>BinTree</code>	82
4.13. Comparación de nodos a través de sus padres mínimos	82
4.14. <code>fact condicionDeOrdenTNode3</code>	83
4.15. <code>fact condicionDeOrdenTNode4</code>	84
4.16. <code>fact condicionDeOrdenTNode5</code>	84
4.17. <code>fact withoutHoles</code>	85

4.18. Funciones auxiliares <code>nextNode</code> , <code>minNode</code> y <code>prevsNode</code>	86
4.19. Función <code>Freach</code> - <code>LinkedList</code>	87
4.20. Funciones <code>globalMin</code> y <code>parentMin</code> - <code>LinkedList</code>	87
4.21. <code>fact condicionDeOrdenNoder4</code> - <code>LinkedList</code>	88
4.22. <code>fact condicionDeOrdenNoder5</code> - <code>LinkedList</code>	88
4.23. <code>fact withoutHolesNode</code> - <code>LinkedList</code>	89
4.24. Invariante de aciclicidad en <code>LinkedList</code>	90
4.25. Ejecución de <code>Bottom-up</code> para <code>LinkedList</code>	95

Índice de cuadros

2.1. Traducción de algunas operaciones de <code>JMLObjectSequence</code> a Alloy.	29
3.1. <i>Workarounds transitorios</i> para <code>set(0,4)</code>	56
3.2. Esquemas de <i>Workarounds</i> para <code>TreeSet</code>	67
4.1. Matriz de representación interna para el campo <code>c</code>	91
4.2. Matriz de representación interna para <code>next</code>	91
4.3. Matriz de representación interna para <code>next</code>	92
5.1. Caso de estudio: <code>List</code> - Técnica 1	100
5.2. Caso de estudio: <code>Set</code> - Técnica 1	102
5.3. Caso de estudio: <code>Map</code> - Técnica nro.1	103
5.4. Caso de estudio: <code>TimeOfDay</code> - 16 bits - Técnica 1	105
5.5. Caso de estudio: <code>TimeOfDay</code> - 24 bits - Técnica 1	106
5.6. Caso de estudio: <code>TimeOfDay</code> - 32 bits - Técnica 1	107
5.7. Aceleración aplicando <i>esquemas de workarounds</i> para <code>List</code> . .	109
5.8. Aceleración aplicando <i>esquemas de workarounds</i> para <code>Set</code> . .	109
5.9. Aceleración aplicando <i>esquemas de workarounds</i> para <code>Map</code> . .	110
5.10. Cómputo de <i>workarounds</i> para <code>Stack</code>	113
5.11. Cómputo de <i>workarounds</i> para <code>Vector2</code>	114
5.12. Cómputo de <i>workarounds</i> para <code>Vector3</code>	114
5.13. Cómputo de <i>workarounds</i> para <code>Edge</code>	115
5.14. Cómputo de <i>workarounds</i> para <code>Path</code>	116
5.15. Caso de estudio: <code>LinkedList</code> - Técnica 2	118
5.16. Caso de estudio: <code>ApacheList</code> - Técnica 2	119
5.17. Caso de estudio: <code>TreeSet</code> - Técnica 2	121
5.18. Caso de estudio: <code>BSTree</code> - Técnica 2	121

5.19. Caso de estudio: AVLTree - Técnica 2	122
5.20. Caso de estudio: TreeMap - Técnica 2	123
5.21. Caso de estudio: TimeOfDay - 16 bits	125
5.22. Caso de estudio: TimeOfDay - 24 bits	126
5.23. Caso de estudio: TimeOfDay - 32 bits	127

Capítulo 1

Introducción

El software forma parte de nuestra vida cotidiana. La utilización de sistemas de cómputo ha crecido notablemente en las últimas décadas volviéndose indispensable en algunos ámbitos. El software en equipamiento médico, software de navegación, software embebido en electrodomésticos, sistemas bancarios, aplicaciones de celulares, redes sociales, son algunos de los sistemas con los que interactuamos diariamente. A medida que el software se expande aumenta también la posibilidad de que aparezcan defectos o *bugs*. Caídas masivas en redes sociales como Facebook, WhatsApp [12], inconsistencias en registros, o la tan famosa BSOD (Blue Screen of Death) de Windows, entre otros, han provocado molestias y pérdidas a millones de usuarios en todo el mundo. La situación es más grave aún cuando el software tiene carácter crítico, donde un error puede ocasionar una catástrofe. La explosión del Ariane 5 [32], un cohete espacial no tripulado europeo que estalló segundos después de su lanzamiento por un error de conversión de datos produciendo pérdidas millonarias. Otro caso famoso fue el del Therac-25 [61], un dispositivo médico utilizado en tratamientos de pacientes con cáncer mediante radioterapia. Debido a un defecto en el software de control del dispositivo, en ocasiones se exponía a los pacientes a dosis letales de radiación (100 veces mayores de lo esperado) causando la muerte de seis personas entre los años 1985 y 1987. Estos son algunos ejemplos que ponen en evidencia la necesidad de desarrollar software correcto y confiable.

El desarrollo de software libre de fallas es una actividad compleja donde múltiples factores entran en juego. La constante adaptación y extensión de los sistemas, la complejidad intrínseca del software, la creciente presión por disminuir los tiempos de producción y comercialización son algunos de los

factores intervinientes [42, 76, 83]. Dentro del área de la Ingeniería de Software se han realizado diversas contribuciones con el fin de producir software confiable y de calidad. En este sentido los métodos formales han realizado grandes aportes en las diferentes etapas del ciclo de vida del software. Se han propuesto lenguajes formales de especificación y modelado ([21, 49, 75] entre otros) a fin de brindar descripciones precisas del comportamiento del software a desarrollar. Novedosas técnicas de testing automático [2, 8, 11], procedimientos de model checking [9, 26, 27] y la aplicación de métodos deductivos de prueba [30, 31] son algunas de las contribuciones en pos de incrementar la calidad y confiabilidad del software.

Sin embargo, y a pesar de los esfuerzos realizados, en muchas ocasiones los defectos siguen presentándose en etapas de producción. Las denominadas fallas de campo son aquellas vinculadas a defectos no detectados durante las etapas de testing y que son percibidas y reportadas por los usuarios. Este tipo de fallas suelen tener un período de vida extenso desde el momento en que son reportadas hasta ser corregidas [15]. Esto se debe, entre otras causas, a que están estrechamente vinculadas al ambiente de ejecución de los programas, por lo que se requiere un esfuerzo adicional importante para generar las condiciones ambientales idénticas para su detección y corrección a costos compatibles con los objetivos de los desarrolladores y usuarios.

1.1. Antecedentes

El término *self-healing* describe a sistemas, dispositivos o servicios que pueden detectar un mal funcionamiento durante su ejecución y, sin intervención humana realizar las acciones correctivas para continuar funcionando correctamente [41]. La noción de *workaround* es utilizada con el objetivo de proveer técnicas y mecanismos que permitan automáticamente encontrar alternativas de ejecución ante fallas.

1.1.1. Workarounds

Un *workaround* para una rutina defectuosa r consiste en una secuencia de una o más rutinas que producen un resultado equivalente a r de acuerdo a las especificaciones [4]. De esta forma, si al ejecutar r ocurre una falla, el *workaround* podrá ejecutarse para alcanzar un estado que permita continuar normalmente con la ejecución del programa, enmascarando de esta manera

la falla.

Trabajos previos [15,17] han propuesto una arquitectura de recuperación basada en *workarounds*. Esta arquitectura define módulos para la detección de fallas en tiempo de ejecución, la aplicación de una acción de `rollback` del sistema a un estado anterior seguro, el cómputo y selección automática de una alternativa que permita evitar la falla y su posterior ejecución a partir del estado seguro. En la literatura este proceso es definido como *workarounds automáticos* [18]. El proceso de detección de fallas en estos trabajos es realizado a través de las excepciones producidas por las rutinas involucradas (aunque podrían incorporarse otros enfoques, como los basados en especificaciones formales). Por otro lado, se requiere que el desarrollador o usuario provea explícitamente los *workarounds* a través de reglas de reescritura de términos, aunque se deja abierta la posibilidad a que puedan ser definidos a través de otros formalismos como por ejemplo máquinas de estados finito o especificaciones algebraicas. Es importante destacar que estos trabajos no definen mecanismos automáticos para el cómputo de *workarounds*. Los *workarounds* en estos enfoques deben ser proporcionados de manera explícita por el programador o usuarios expertos de la librería. El aporte principal de esta tesis es proveer mecanismos automáticos de cómputo de *workarounds transitorios* basados en las especificaciones formales de las rutinas.

Una variante de la técnica anterior utiliza *workarounds automáticos* sobre sistemas web. En este caso, el usuario forma parte del proceso de recuperación. El usuario es el encargado de la detección de las fallas, como así también es quien solicita y valida un *workaround* de recuperación. Por ejemplo, si una aplicación hace uso de la API de Google Maps para mostrar un mapa y ocurre una falla que ocasiona que el mapa no se muestre en forma correcta, el usuario podrá solicitar un *workaround* de recuperación, evaluarlo y aceptarlo o rechazarlo (si éste no brinda el comportamiento esperado). En esta propuesta los *workarounds* son derivados a partir de dos formalismos: máquinas de estados finito [4] y sistemas de reescritura de términos [56].

Posteriormente fue presentada una técnica de cómputo de métodos equivalentes aplicando algoritmos genéticos [45]. El procedimiento de búsqueda en este caso es dividido en dos etapas. En la primera se plantea la búsqueda de secuencias equivalentes candidatas a una rutina dada como objetivo. En una segunda etapa se verifica la validez de la alternativa encontrada. Esta técnica no requiere de especificaciones formales, en su lugar utiliza tests para establecer el comportamiento de la rutina involucrada y para validar las secuencias alternativas candidatas. La equivalencia de las rutinas consideradas

se da a nivel de implementación y no de sus especificaciones. La herramienta EvoSuite [35] (para la generación de casos de test) es aplicada en las dos etapas de esta técnica. Esta propuesta aplica conceptos relacionados a búsqueda y síntesis de ejecuciones equivalentes aprovechando la redundancia en el software, conceptos que comparte con las propuestas de esta tesis.

Otros antecedentes como los presentados en [55,91] plantean la reparación de estructuras de datos mediante SAT. En estos casos se utilizan especificaciones formales para definir el comportamiento de las rutinas involucradas y para la detección de las fallas. Básicamente, este enfoque propone monitorear la ejecución del programa bajo análisis, instrumentando el código en donde se accede a las estructuras de datos. Ante la identificación de una violación de un invariante que ocasione la corrupción de una estructura de datos, se aplicará un mecanismo de decisión basado en SAT solving para modificar la estructura de forma que satisfaga el invariante, y de esta manera reparar la falla.

1.1.2. Especificaciones formales

El cómputo de *workarounds* basado en especificaciones propuesto en este trabajo requiere de una definición formal del comportamiento de las rutinas del módulo de software. Las especificaciones formales han sido utilizadas en diversas propuestas que realizan análisis automático de software. Una de las más usadas es la metodología denominada *diseño por contratos* [65]. Esta metodología establece que una clase y sus clientes definen un contrato que deberán satisfacer. Los clientes deben garantizar el cumplimiento de determinadas condiciones iniciales previas a la invocación de un método m (precondición), y en contrapartida la clase dará la garantía del cumplimiento de otra serie de condiciones (postcondición) luego de la invocación de m . Además, es posible definir invariantes de clase, es decir, aserciones que deben ser garantizados por la ejecución de cada rutina. El lenguaje de programación Eiffel [64] implementa esta metodología. Los contratos son incorporados a los programas y luego son traducidos a código ejecutable. De esta manera, las violaciones de los contratos pueden detectarse, por ejemplo, en el testing o durante la ejecución del programa.

En este mismo sentido, JML(Java Modeling Language) [59] es un lenguaje declarativo que permite especificar precondiciones, postcondiciones e invariantes de clases para programas Java. A diferencia de Eiffel, JML incorpora cuantificadores y expresiones de alcanzabilidad para definir de manera

declarativa contratos muy expresivos. JML ha sido utilizado por múltiples herramientas ([20, 57] entre otras) para verificar contratos en tiempo de ejecución, realizar análisis estático de código fuente, verificación, etc.

1.1.3. Herramientas de análisis basadas en Alloy

Las técnicas propuestas en esta tesis hacen uso de los lenguajes de modelado Alloy [49] y DynAlloy [36, 37]. Alloy es un lenguaje de especificación basado en una lógica de primer orden extendida con operadores relacionales (composición de relaciones, clausura transitiva, etc.). Alloy está orientado al modelado de propiedades estructurales de sistemas. Alloy es un lenguaje simple y expresivo para construir modelos de software *estáticos*. En contraposición, Alloy no posee una forma sencilla para modelar sistemas dinámicos (que cambian de estado durante su ejecución) ni de sus propiedades.

DynAlloy [36] es una extensión de Alloy que incorpora construcciones para modelar fácilmente la noción de cambio de estado y propiedades dinámicas a verificar en un sistema. Los lenguajes Alloy y DynAlloy poseen sus respectivas herramientas de análisis automático: Alloy Analyzer y Dynalloy Analyzer. Dichas herramientas permiten una verificación exhaustiva acotada de modelos en los respectivos lenguajes. Dados dominios de datos acotados (con cotas provistas por el usuario), estas herramientas buscan de manera automática violaciones de propiedades en los modelos. Ambas herramientas utilizan SAT solving como procedimiento de decisión. En particular, DynAlloy Analyzer traduce el modelo dinámico en un modelo Alloy equivalente, que luego será analizado utilizando Alloy Analyzer.

Otra herramienta vinculada con las propuestas de esta tesis es TACO [38]. TACO es una herramienta de verificación exhaustiva acotada para programas Java anotados con contratos JML. TACO es capaz de encontrar ejecuciones de una rutina que violen su especificación de manera automática. Para ello realiza una serie de traducciones de programas JAVA y sus especificaciones JML a modelos DynAlloy, que posteriormente serán traducidos a modelos Alloy, y finalmente analizados en busca de inconsistencias mediante SAT solving. En esta tesis se utiliza el proceso de traducción de aserciones JML a Alloy presentado en [38] e implementado en TACO. El lenguaje DynAlloy es utilizado también para construir modelos para la búsqueda de *workarounds*.

1.2. Objetivos y Contribuciones

Las contribuciones de esta tesis están dirigidas al desarrollo de procedimientos de cómputo automático de *workarounds* para rutinas, basados en contratos y cuya finalidad es recuperar la ejecución de un sistema ante la ocurrencia de una falla. Se asume la existencia de un mecanismo de detección de las fallas en tiempo de ejecución y la aplicación de una acción de *rollback* que posicione al sistema en el estado de invocación de la rutina defectuosa, como el presentado en [15]. A partir de ese estado se procederá a computar los *workarounds* de recuperación.

En esta tesis se proponen dos técnicas de cómputo de *workarounds transitorios* aplicando SAT solving. Dichas técnicas son aplicadas a programas Java anotados con especificaciones JML.

Un *workaround transitorio* para una rutina defectuosa r se define como una secuencia de una o más rutinas, cuya ejecución (desde el estado de invocación corriente de r) imita el comportamiento de la rutina r (según lo establecido en las especificaciones de las rutinas).

Por ejemplo un *workaround transitorio* para la rutina `clear()` invocada desde una lista l de longitud 1 podrá obtenerse invocando el método `l.removeFirst()` el cual elimina el primer elemento de l dejándola vacía. Como se puede observar, un *workaround transitorio* puede no ser válido en otros escenarios de ejecución diferentes en donde la misma rutina falla. La invocación de `l.removeFirst()` no podrá reemplazar a `l.clear()` en los casos donde la longitud de l sea diferente de 1.

En los trabajos previos, como los mencionados en la sección de antecedentes, las alternativas computadas por los diferentes enfoques son en general alternativas permanentes. Un *workaround permanente* para una rutina defectuosa r se define como una secuencia de una o más rutinas que produce un resultado equivalente a r , para cualquier estado inicial válido de r (que satisfaga la precondition de r). Por ejemplo, para el caso de la rutina `clear()` aplicada sobre una lista l , podría utilizarse la secuencia:

$$l' = l.copy() ; l.removeAll(l')$$

donde l' es una lista auxiliar y sobre el estado inicial de l no se exige condición alguna.

En la práctica una gran cantidad de escenarios de ejecución sólo pueden repararse a través de *workarounds transitorios* que no pueden generalizarse

a *workarounds permanentes*. Por ejemplo el método `remove(e)` que elimina el elemento `e` de una estructura de datos retornándolo, en una implementación del TAD Set no posee *workarounds permanentes*. Pero dado un estado de invocación particular como por ejemplo: `c = {2,50,-100}`, la rutina `c.remove(-100)` podría ser reemplazada por `c.pollFirst()` (método que retorna y elimina el menor elemento del Set `c`). Esta alternativa sólo podrá utilizarse cuando el elemento `e` sea el menor elemento del Set. Los *workarounds transitorios* son computados en tiempo de ejecución, utilizando el contrato de la rutina defectuosa y el estado previo a su invocación. En cambio, los *workaround permanentes* pueden computarse previamente a la ejecución del sistema, ya que son independientes del estado de invocación.

Durante la etapa experimental se realizó un análisis comparativo de los *workarounds* computados para diferentes casos de estudio observando particularmente su condición de transitorio o permanente. De dicho análisis se pudo establecer que en una gran cantidad de casos no fue posible evitar las fallas con *workarounds permanentes*, siendo posible hacerlo con *workarounds transitorios* (ver Sección 5.2).

La primera técnica propuesta en esta tesis asume la existencia de redundancia intrínseca en el software [4, 14, 44]. Los *workarounds transitorios* para una rutina defectuosa `r` son computados a partir de los contratos de las rutinas del módulo. El procedimiento consiste en definir un programa DynAlloy que contemple todas las posibles secuencias de rutinas de longitud `m` (como máximo) del módulo bajo análisis, en donde se excluye a la rutina defectuosa `r` del estado de invocación (obtenido luego de aplicar una acción *rollback* sobre el estado de falla). Además de la longitud `m` de las secuencias a generar, la técnica requiere de cotas para los dominios de los tipos de datos involucrados, es decir, una cantidad máxima de elementos para cada tipo de dato. Como es usual a estas cotas se las denominan *scopes* y son provistos por el usuario. Posteriormente este programa DynAlloy es sometido al análisis de un SAT solver quién realizará el cómputo de los *workarounds transitorios*, es decir, buscará una combinación de rutinas que partiendo del estado inicial dado llegue a un estado que satisfaga la postcondición de `r`.

El problema de escalabilidad es central en este tipo de enfoques automáticos que aplican análisis de SAT ya que usualmente el espacio de estados a explorar es inmenso, y crece exponencialmente en función del incremento de los *scopes*. Por otra parte, el uso de especificaciones en lugar del código fuente de las rutinas facilita el análisis, ya que las especificaciones evitan la necesidad de modelar todos los posibles estados intermedios en la ejecución de cada

rutina (sólo se requiere la relación entrada/salida de cada una de ellas). También acelera sustancialmente la búsqueda al partir de un único estado bien definido. En la evaluación experimental de las técnicas propuestas (capítulo 5) se muestra que la técnica es capaz de encontrar *workarounds transitorios* para un gran número de casos en la práctica y en tiempos razonables para la recuperación de aplicaciones.

En pos de acelerar el procedimiento de cómputo se propuso la noción de *esquemas de workaround*. Un *esquema de workaround* se define como la generalización de un *workaround transitorio* fijando la secuencia de rutinas a ejecutar y permitiendo la variación de los valores en sus parámetros. De esta manera, si una rutina defectuosa r falla al ejecutarse en contextos diferentes, el procedimiento de recuperación intentará instanciar un *esquema de workaround* derivado previamente para la rutina, con grandes posibilidades de salvar la ejecución fallida. La búsqueda de los valores adecuados para los parámetros del esquema se realiza aplicando SAT solving. Claramente, este procedimiento es más eficiente que generar un *workaround transitorio* a partir de todo el conjunto de rutinas del módulo ya que el espacio de búsqueda se reduce sustancialmente. La incorporación de *esquemas de workarounds* fue evaluada experimentalmente. En los casos de estudio analizados se observa que a partir de la incorporación de *esquemas de workaround* el cómputo de *workarounds transitorios* es un orden de magnitud más rápido en promedio.

La segunda técnica propuesta intenta construir directamente un estado del sistema al que se hubiese arribado si la rutina defectuosa r se hubiese ejecutado de manera exitosa (respecto a su especificación). A partir de la especificación de r y su estado de invocación, el procedimiento de cómputo intentará construir un modelo Alloy que satisfaga la postcondición de r . Dicha instancia caracteriza al estado de recuperación buscado, ya que a partir de este último se podrá continuar con la ejecución normal del programa. Como la anterior técnica propuesta en esta tesis, este análisis requiere de cotas para los dominios de datos provistos por el usuario. La técnica descrita es parte de lo que en la literatura se conoce como enfoques de reparación de estructuras basada en restricciones [81, 91, 92].

A diferencia de los enfoques anteriores, la segunda técnica propuesta en esta tesis se distingue principalmente por incorporación de dos conceptos muy útiles para optimizar el análisis basados en SAT solving. Estos conceptos son predicados de rotura de simetrías y cotas ajustadas [38, 40], los mismos han sido aplicados exitosamente en verificación exhaustiva acotada (TACO) [38], generación automática de tests [72, 74, 78], entre otros. Debido a que el factor

de escalabilidad es determinante para este tipo de técnicas, la aplicación de estas optimizaciones brinda la posibilidad de incrementar el tamaño de estructuras analizadas permitiendo generar estados de recuperación en casos donde técnicas similares agotan los recursos computacionales disponibles (ver Sección 4.2, 4.3).

Con respecto a los trabajos mencionados en la sección de antecedentes, las técnicas propuestas en esta tesis computan *workarounds transitorios* utilizando contratos para la definición del comportamiento de las rutinas. En los trabajos previos los *workarounds* han sido computados a partir de reglas de reescritura de términos o bien a través de máquinas de estados finitos. La aplicación de contratos ha dejado de ser exclusiva para ambientes académicos de investigación desde hace ya varios años. Lenguajes de programación como Eiffel y de especificación como JML han extendido su aplicación a gran parte de la comunidad de desarrolladores. Los contratos han sido utilizados para diversos fines, *bug finding* en *runtime*, testing, entre otros. En particular, para el lenguaje Java (utilizado por las técnicas propuestas en esta tesis) existe un repositorio ¹ abierto en donde es posible acceder a las especificaciones JML de las bibliotecas estándar como por ejemplo `java.util`), entre otras.

Para evaluar el rendimiento de las técnicas propuestas se realizó una serie de estudios experimentales sobre varias implementaciones de colecciones, y una biblioteca Java para aritmética de fechas que fueron equipadas con contratos por el autor. Para la evaluación se generaron estados iniciales aleatoriamente, usando una herramienta de generación automática de tests (Randoop [68]). Estos estados simulan el producto de la detección de una falla y la aplicación de una acción de *rollback*. Luego se aplicaron los enfoques definidos en esta tesis para computar *workarounds transitorios* para cada rutina del módulo y cada estado inicial, como una forma de evaluar la efectividad de los enfoques propuestos. Los resultados obtenidos de la evaluación muestran que las técnicas permiten computar de manera eficiente *workarounds transitorios* en un número importante de casos, y en tiempos que hacen a las técnicas factibles de ser utilizadas para mantener sistemas funcionando a pesar de la ocurrencia de fallas en tiempo de ejecución. Además, los *workarounds transitorios* encontrados permiten recuperar a los sistemas bajo análisis en muchos casos donde los enfoques anteriores no son aplicables. Por otro lado, los resultados experimentales muestran que la aplicación de *esquemas de workarounds* permiten mejorar significativamente la eficiencia de la

¹<http://www.eecs.ucf.edu/~leavens/JML-release/javadocs/overview-summary.html>

primera técnica propuesta, como así también la incorporación de *rotura de simetrías y cotas ajustadas* mejoran notablemente la eficiencia y la escalabilidad de la segunda técnica propuesta, posibilitando computar *workarounds transitorios* para estructuras de mayor tamaño.

1.3. Organización

A continuación se presenta la organización en capítulos de esta tesis. En el capítulo 2 se realiza una revisión de los conceptos previos requeridos y aplicados en esta tesis. Los capítulos 3 y 4 presentan las técnicas de cómputo de *workarounds transitorios* propuestas en esta tesis. En el capítulo 5 se muestran los resultados de la evaluación experimental de las técnicas. En el capítulo 6 se realiza una breve reseña de trabajos relacionados relevantes a las técnicas propuestas. Finalmente, en el capítulo 7 se presentan las conclusiones y los trabajos futuros.

Capítulo 2

Preliminares

En este capítulo se realiza una revisión de los tópicos principales vinculados con esta tesis que coadyuvan a su comprensión. Se utilizan definiciones formales e informales junto con las citas bibliográficas a los trabajos completos. Esta no pretende ser una revisión acabada de las temáticas involucradas, sino más bien se hace mención a los conceptos que a criterio del autor son los más relevantes. Se presenta en primer lugar las definiciones de satisfacibilidad booleana (SAT) y la automatización de su análisis. A continuación se describen los lenguajes de especificación Alloy y DynAlloy utilizados en los enfoques propuestos en esta tesis. También se describe brevemente el lenguaje de especificación JML (Java Modeling Language) y sus principales características junto con el mecanismo de traducción de especificaciones JML al lenguaje Alloy, utilizada también por las técnicas propuestas en la tesis. Finalmente se introduce el concepto de *workaround* y se describe una arquitectura de recuperación basada en *workarounds*.

2.1. Análisis basados en satisfacibilidad booleana (SAT)

Los enfoques de cómputo de *workarounds transitorios* propuestos en esta tesis están basados en análisis de satisfacibilidad booleana (SAT). El problema de SAT consiste en determinar la existencia de una interpretación I para una fórmula proposicional F que la satisfaga. Más precisamente, la fórmula F se dice satisfacible si existe una asignación de valores de verdad para cada variable proposicional (I) que evalúe a F en verdadero. El carácter exhaustivo

del análisis de SAT conlleva a considerar todas las posibles asignaciones de valores de verdad a las variables proposicionales de F hasta encontrar una asignación que la satisfaga, o bien determinar que no existe tal asignación. Este procedimiento es realizado a través de un SAT solver, una herramienta de software que implementa la búsqueda (exhaustiva) de interpretaciones para dar solución al problema planteado [27].

Resolver el problema de SAT requiere, en el peor caso, evaluar una cantidad de interpretaciones que crece exponencialmente a medida que aumenta el número de variables proposicionales en F . La aplicación de optimizaciones para reducir la cantidad de variables en F como las empleadas en esta tesis (cotas ajustadas) son de vital importancia para mejorar la escalabilidad de los análisis automáticos basados en SAT [74, 79].

2.2. Alloy

Alloy [49] es un lenguaje de especificación basado en una lógica de primer orden relacional, y orientado al modelado de propiedades estructurales de sistemas. Además de los operadores típicos de primer orden, Alloy incluye operadores relacionales tales como el operador de composición (\cdot), clausura transitiva ($+$) y reflexo-transitiva ($*$) que facilitan la expresión de propiedades típicas de modelos de software. Alloy posee una sintaxis simple, clara y fácil de comprender. Alloy Analyzer [49] es una herramienta que implementa un análisis automático de las especificaciones Alloy, usando SAT solving como mecanismo de decisión.

A través de un ejemplo simple que define el tipo abstracto de datos *listas simplemente encadenadas*, se presentan a continuación las principales características del lenguaje Alloy y de su herramienta de análisis Alloy Analyzer.

En Alloy los tipos de datos son definidos a través de *signaturas*.

En la figura 2.1 se muestran las signaturas `Null`, `Node`, `List` e `Int` que definen la estructura de listas simplemente encadenadas. La signatura `Int` (enteros) está predefinida en Alloy. Cada signatura tendrá un conjunto de átomos que definen su dominio. La cantidad de átomos dependerá de los *scopes* establecidos para cada signatura. El modificador `one` obliga a las signaturas a tener exactamente un elemento (singleton). Este modificador permite definir constantes dentro de las especificaciones, como por ejemplo `Null`. Las signaturas pueden tener *campos*, algo similar a lo que en el paradigma orientado a objetos son los atributos. La signatura `List` del ejemplo, posee los campos

```

one sig Null { }

sig List {
  head: one Node+Null,
  size: one Int
}

sig Node {
  data: one Int,
  next: one Node+Null
}

```

Figura 2.1: Signaturas de *listas simplemente encadenadas*

`head` y `size`, que identifican el primer elemento y la cantidad de elementos en la lista, respectivamente. El campo `head` se define como una relación que mapea una lista (átomo de la signatura `List`) en un nodo (átomo de la signatura `Node`) o `Null` (figura 2.2). El modificador `one` fuerza a que toda lista posea un sólo nodo `head` o que `head` sea `Null` (en caso de que la lista estuviese vacía). El campo `size` define una relación de `List` en `Int` asociando cada lista con la cantidad de elementos que contiene.

Por otro lado en la figura 2.1, se presenta la signatura `Node` con las relaciones `data` y `next`. La primera representa el valor entero almacenado en un nodo, y la relación `next` indica el nodo con el que está enlazado (puede ser `Null`). El modificador `one` se incluye aquí también a fin de evitar que un nodo posea más de un nodo enlazado directamente (y almacene más de un valor).

$$\begin{aligned}
\text{head} &\subseteq \text{List} \times (\text{Node} + \text{Null}), \\
\text{size} &\subseteq \text{List} \times \text{Int}, \\
\text{data} &\subseteq \text{Node} \times \text{Int}, \\
\text{next} &\subseteq \text{Node} \times (\text{Node} + \text{Null}).
\end{aligned}$$
Figura 2.2: Relaciones `head`, `size`, `data` y `next`

Además, Alloy permite definir restricciones, predicados y aserciones. Mediante la palabra reservada `fact` se definen propiedades que deben cumplirse en la especificación. Por ejemplo, el valor de `size` y la cantidad de nodos alcanzables a partir del campo `head` por `next` deben coincidir (figura 4.24(a)). Notar que esta definición se basa en $\hat{\text{next}}$, que denota la clausura transitiva de la relación `next`. El operador `#` denota el cardinal del conjunto de nodos alcanzables de la lista. El operador `.` compone relaciones, y puede in-

interpretarse intuitivamente de manera similar al acceso a campos en lenguajes orientados a objetos. Por otro lado, se desea que las listas cumplan la propiedad de aciclicidad por lo que se incluye un segundo `fact` (figura 4.24(b)). Este último exige que todo nodo `n` no sea alcanzable a través de una o más aplicaciones de `next` partiendo de si mismo, es decir, que no se formen ciclos.

```
(a) fact listLength{
    List.size= #(List.head.^next)
}

(b) fact acyclicLists {
    no n:Node | n in n.^next
}
```

Figura 2.3: *facts* para *listas simplemente encadenadas*.

Alloy permite la definición de predicados a través de fórmulas parametrizadas mediante las cuales es posible expresar propiedades. En el ejemplo presentado, los predicados pueden ser útiles para capturar el comportamiento de las operaciones sobre listas. En la figura 2.4 se muestran los predicados correspondientes a las operaciones `getFirst()` (retorna el primer elemento de una lista no vacía) y `getLast()` (retorna el último elemento de una lista no vacía). Ambos predicados reciben como parámetros la lista `l` sobre la cual se aplica la operación y un átomo entero `result` donde se almacena el resultado. Para el caso de `getFirst()`, la fórmula establece que la lista a la que se aplica no debe ser vacía y define como `result` al valor del nodo alcanzable a través del `head` de `l`. Por otro lado, para `getLast()` se exige, al igual que para el caso anterior, que `l` sea no sea vacía y se define a `result` como el valor del nodo `n` alcanzable desde el `head` de `l` a través de la relación `next`, tal que `n.next=Null` (el último de la lista).

Por otra parte, las aserciones son propiedades pretendidas, es decir, propiedades que el usuario espera que se cumplan en el modelo. Por ejemplo, se podría chequear que, si una lista tiene un `size = 1`, entonces los métodos `getFirst()` y `getLast()` deberían retornar el mismo valor. La aserción `getFirstEqGetLast` (figura 2.5) permite realizar el chequeo (acotado) de esta propiedad.

Alloy Analyzer es una herramienta que implementa un análisis exhaustivo acotado de especificaciones Alloy. Utilizando SAT solving como procedimien-

```

pred getFirst[l: List, result': Int] {
  l.head != Null and result' = l.head.data
}

pred getLast[l: List, result': Int] {
  l.head != Null and
  some n: Node | n in l.head.*next and
  n.next=Null and result'=n.data
}

```

Figura 2.4: Predicados `getFirst` y `getLast`

to de decisión permite buscar modelos que satisfagan predicados, o modelos que exhiban violaciones de aserciones. El análisis basado en SAT requiere que se fijen los *scopes* para cada signatura determinando así límites máximos en la cantidad de elementos a considerar. Recordemos que todo análisis de este tipo es acotado y que a medida que los *scopes* se incrementan la cantidad de variables involucradas en el proceso de SAT crece exponencialmente implicando mayores recursos computacionales. El chequeo de una aserción se realiza invocando el comando `check`. Para la aserción `getFirstEqGetLast` (figura 2.5) se establecieron los siguientes *scopes*: 1 List, 5 Int, 5 Node. Para la signatura `Int` de Alloy, el *scope* indica la cantidad de bits utilizados para representar enteros en complemento a la base; en este caso los enteros considerados están en el rango `[-16,15]`. Alloy Analyzer intentará producir una instancia que no satisfaga la aserción y en caso de encontrarla la exhibirá a modo de contraejemplo. En caso contrario, no se podrá asegurar que la propiedad sea válida ya que el análisis es acotado, pero permitirá incrementar la confianza en el cumplimiento de la propiedad.

En la figura 2.5 también se muestran dos ejemplos de comandos `run`. En ambos casos se emplea SAT solving para construir instancias con a lo sumo 1 lista, 5 nodos y enteros en el rango `[-16,15]`. La ejecución de los comandos `run` intentarán generar instancias (acotadas) que satisfagan los predicados `getFirst` y `getLast`, respectivamente. Un ejemplo de la salida del comando `run getFirst for 5 but 1 List, 5 Int` se muestra en la figura 2.6. En la figura se observa una lista de tamaño 2 (`size=2`) compuesta por los nodos `Node1` y `Node2`. El `head` de la lista es `Node1`. El último elemento de la lista es `Node2` (`Node2.next=Null`). El `result'` del predicado `getFirst` se

corresponde con el campo `data` de `Node1`, es decir el valor entero 12.

```
assert getFirstEqGetLast {
  all l: List | all i, j: Int |
    l.size = 1 and getFirst[l,i] and getLast[l,j] => i = j
}
```

```
check getFirstEqGetLast for 5 but 1 List, 5 Int, 5 Node
```

```
run getFirst for 5 but 1 List, 5 Int, 5 Node
```

```
run getLast for 5 but 1 List, 5 Int, 5 Node
```

Figura 2.5: Aserción `getFirstEqGetLast`

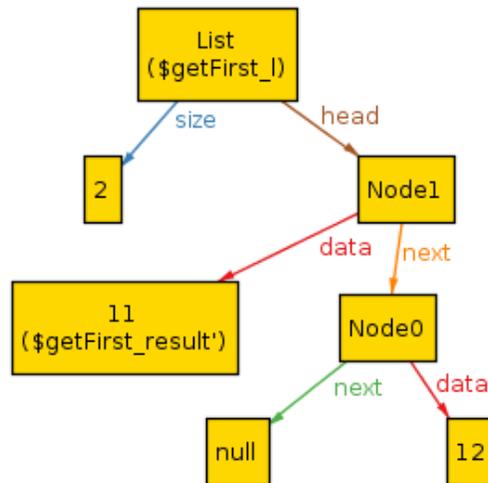


Figura 2.6: Instancia del predicado `getFirstEqGetLast`

2.3. Dynalloy

Alloy es un lenguaje de especificación simple y expresivo para construir modelos de software estáticos. En contraposición, Alloy no posee una forma sencilla para modelar sistemas dinámicos que capturen la ejecución de

sistemas mediante cambios de estado. DynAlloy [36] es una extensión de Alloy que incorpora constructores para modelar fácilmente la noción de cambio de estado. La sintaxis y la semántica de DynAlloy está basada sobre una lógica dinámica. En la figura 2.10 se muestra la gramática de Dynalloy definida por M. Frias y otros [37]. DynAlloy extiende Alloy con acciones atómicas (**act**), programas (**program**), y aserciones de corrección parcial (**assertCorrectness**). Las acciones son definidas mediante contratos (precondiciones y postcondiciones) incluidos en cada acción a través de los predicados **pre** y **post**.

Continuando con el ejemplo de listas presentado en la sección anterior y para ejemplificar el uso de acciones atómicas, se define **clear**(figura 2.7). Esta acción atómica elimina todos los elementos de una lista dada. La acción recibe como parámetros a **this** que hace referencia a la lista corriente sobre la cual se aplica la acción y dos relaciones binarias **head** y **size** las cuales mapean a una lista con su primer elemento (o **Null**) y la cantidad de elementos en la lista, respectivamente. El contrato de **clear** no requiere precondición alguna para ejecutarse por lo que es igual a verdadero. El predicado que define la postcondición **post** utiliza el operador **++** para sobrescribir valores en las relaciones **head** y **size**, de forma tal que el **head** de la lista **this** quede en **Null**(lista vacía) y el tamaño (**size**) de **this** sea igual a 0. Las variables primadas denotan a los valores de las variables después de la ejecución de cada acción, permitiendo modelar el cambio de estado. Un detalle a observar es que no es necesario definir las variables primadas **head'** y **size'** explícitamente como argumentos de la acción. Cuando una variable primada no es mencionada en la postcondición se asume que la acción no ha producido cambios en ella. En este caso, la variable **this** no se modifica luego de la ejecución de **clear**.

Los programas DynAlloy se construyen utilizando los operadores de asignación (**:=**), **skip**, **tests** y acciones atómicas combinadas mediante la aplicación de composición secuencial (**;**), elección no determinística (**+**) e iteración (*****). El operador de asignación (**:=**) se define como la acción atómica **assign** (ver figura 2.8) que recibe los parámetros **v** y **value** representado a la variable y el valor a asignar respectivamente. En la postcondición se establece que en **v'** (la variable **v** luego de la ejecución de **assign**) queda almacenado **value**. En la figura 2.9 se presenta el programa DynAlloy **choose** en donde se ejemplifica el uso de los operadores anteriores. El programa **choose** itera sobre una lista de enteros y de forma no determinística selecciona uno de sus elementos. Los parámetros del programa son: la lista **l** de enteros y **result**

```
act clear[ thiz: List, head: List -> one (Node+Null),
           size: List -> one Int] {
  pre {}
  post { head' = head ++ (thiz -> Null)
        and size' = size ++ (thiz -> 0) }
}
```

Figura 2.7: Acción atómica `clear`.

donde es almacenado el valor seleccionado. En el cuerpo del programa se definen dos variables locales `chosen` (para indicar si un elemento de `l` ha sido seleccionado o no) y `curr` para almacenar el nodo corriente de `l`. Inicialmente a la variable `chosen` se le asigna falso y a `curr` el `head` de `l`. Luego, el programa itera hasta un máximo de 5 veces (ver comando `run choose for 5 but 1 List, 5 Int, 5 lurs` y discusión sobre `loop unrolls` más abajo) mientras se cumpla el test `[curr!=Null]?`, es decir, mientras no se haya alcanzado el final de `l`. Notar que el test sólo permite continuar con las ejecuciones que satisfagan el predicado del mismo. En cada paso de la iteración el programa no determinísticamente (+) ejecuta secuencialmente las siguientes asignaciones `result:=curr.elem ; chosen:=true` guardando un valor de la lista en `result` y seteando a `chosen` en verdadero, o bien ejecutando la acción `skip` que no produce cambio alguno. Seguidamente se actualiza el nodo corriente mediante la asignación `curr:=curr.next`. El programa finaliza cuando se ha recorrido `l` (`[curr=Null]`) y `chosen` haya sido seteado en verdadero (por el test final `[chosen = true]?`), lo que implica que en `result` se ha almacenado un entero de `l`.

Cabe mencionar que el número de iteraciones en un programa DynAlloy es determinado por la cantidad de `loop unrolls` fijado al momento de su traducción al modelo Alloy. Del mismo modo que son necesarios los *scopes* para acotar la cantidad de elementos en los modelos Alloy, en DynAlloy, se requiere además establecer el máximo de iteraciones o `loop unrolls` considerados para las (*) del programa. Por ejemplo, el programa DynAlloy `P*` para 3 `loop unrolls` podría reescribirse como `P + (P;P) + (P;P;P)`, esto significa que el cuerpo del ciclo `P` se ejecuta una, dos, o tres veces. La cantidad de `loop unrolls` es definida en los comandos `run` o `check`.

```

act assign[v,value: C]{
  pre = { true }
  post = { v' = value }
}

```

Figura 2.8: Acción atómica de asignación.

```

program choose[l: List, result: Int] {
  local [chosen: Boolean, curr: Node+Null]
  chosen := false;
  curr := l.head;
  ([curr!=Null]?;
  (
  (result:=curr.elem; chosen:=true)+skip);
  curr:=curr.next
  )*;
  [chosen = true]?
}

assertCorrectness chooseIsCorrect[l: List, result: Int]
{
  pre { l.size>0 and repOK[l] }
  program = choose[l, result]
  post { some e: l.head.*next.elem | e=result' }
}

```

check chooseIsCorrect for 5 but 1 List, 5 Int, 5 lurs

run choose for 5 but 1 List, 5 Int, 5 lurs

Figura 2.9: Ejemplo de *act*, *program* y *assertCorrectness* para *listas*.

Un programa DynAlloy puede ser enriquecido con aserciones de corrección parcial. Por ejemplo, para verificar si el programa `choose` se comporta de acuerdo a lo establecido en su especificación, se define la aserción `chooseIsCorrect` (figura 2.9). La aserción establece que dada una lista `l` de

```

formula ::= ... | formula program formula
           ‘‘partial correctness’’
program ::= < formula, formula > (x̄)
           ‘‘atomic action’’
           | formula?
           ‘‘test’’
           | program + program
           ‘‘non-deterministic choice’’
           | program ; program
           ‘‘sequential composition’’
           | program*
           ‘‘iteration’’
           | < program > (x̄)
           ‘‘invoke program’’

```

Figura 2.10: Gramática DynAlloy

enteros no vacía que satisfaga el invariante de representación de listas (predicado `repOK`) y luego de ejecutar el programa `choose`, en `result` quedará almacenado uno de los valores enteros contenidos en `l`. Las aserciones permiten verificar el cumplimiento de propiedades en modelos acotados. El chequeo de `chooseIsCorrect` se realiza mediante el comando `check chooseIsCorrect for 5 but 1 List, 5 Int, 5 lurs`.

En la figura 2.9 también se incluye el comando `run choose for 5 but 1 List, 5 Int, 5 lurs` mediante el cual se busca obtener una instancia de ejecución del programa `choose` para una lista con un máximo de 5 enteros considerados en un rango `[-16,15]` y para un máximo de 5 iteraciones. Previa a la verificación de las propiedades definidas en las especificaciones DynAlloy, estas son traducidas a Alloy (el proceso de traducción es definido en [37]). Sobre estas últimas se aplica el análisis de SAT ya descrito en la sección anterior.

Un punto importante a considerar es que a medida que los *scopes* y *loops unrolls* se incrementan, las fórmulas resultantes serán cada vez más grandes y complejas, lo que puede afectar la viabilidad del análisis de SAT.

Alloy y DynAlloy son lo suficientemente expresivos como para representar programas Java y sus especificaciones JML. Éstos han sido utilizados como lenguajes intermedios para diversos análisis, incluida la verificación acotada

y la generación de casos de tests de programas Java con anotaciones JML [2, 38, 40].

2.4. Java Modeling Language (JML)

JML [20, 57] es un lenguaje de especificación formal que permite definir el comportamiento de clases e interfaces escritas en lenguaje Java. Las especificaciones JML son definidas mediante precondiciones, postcondiciones e invariantes lo que posibilita una aplicación directa de la metodología de Diseño por Contratos [65]. Las especificaciones pueden ser incorporadas dentro del mismo código fuente Java a través de anotaciones JML (en comentarios delimitados por `/*@ . . . @*/`) o bien en archivos separados (con extensión `.jml`).

En la figura 2.11 se presenta una implementación de la `interface java.util.List`¹ mediante listas simplemente encadenadas de enteros. La figura contiene las clases `LinkedList` y `Node` con sus respectivos métodos y contratos JML. No se incluye la totalidad de los métodos de las clases debido a que no son necesarios a fin de ejemplificar las características más relevantes de JML. Tampoco se incluye el código fuente de los métodos ya que los mismos son omitidos por los enfoques de cómputo de *workarounds transitorios* presentados en esta tesis. Las técnicas propuestas asumen que el comportamiento de los métodos se define a través de sus contratos.

La clase `Node` posee dos atributos: `value` (valor del entero almacenado en el nodo), y `next` (el nodo siguiente con el cual está vinculado, pudiendo ser `Null`). Por su parte la clase `LinkedList` define al atributo `head` de tipo `Node` que hace referencia al primer nodo de la lista.

Dentro de las especificaciones JML también es posible definir propiedades que caractericen a los objetos de la clase a través de la cláusula `invariant`. La clase `LinkedList` define su cláusula `invariant` (líneas 9-12) a través de un cuantificador universal (`\forall`).

En JML el cuantificador universal tiene la forma:

$$\forall x:T;P(x);H(x)$$

y expresa que para todo `x` de tipo `T` que satisface `P(x)`, se debe verificar `H(x)`. También se utiliza la expresión

¹<https://docs.oracle.com/javase/6/docs/api/java/util/List.html>

```

1  public class Node {
2      public Node next;
3      public int value;
4  }
5
6  public class LinkedList{
7      public Node head;
8
9      /*@ invariant (\forallall Node n;
10         @ \reach(this.head, Node, next).has(n);
11         @ !\reach(n.next, Node, next).has(n));
12         @*/
13
14     /*@ public model instance JMLObjectSequence myseq;
15         @ public represents this.myseq \such_that
16         @ this.myseq.int_size()==\reach(this.head,Node,next).int_size()
17         @ this.myseq.int_size() > 0 ==> (this.myseq.get(0) == this.head \&\&
18         @ (\forallall int j; 0 <= j \&\& j< this.myseq.int_size()-1 \&\&
19         @ this.myseq.get(j).next == this.myseq.get(j+1)));
20         @*/
21
22     /*@ ensures (\exists int i; 0<=i \&\& i<this.myseq.int_size() \&\&
23         @ this.myseq.get(i).value==e)<==>\result==true)
24         @*/
25     public boolean contains(int e){
26         // Retorna verdadero si 'e' esta en la lista y falso en otro caso.
27     }
28
29     /*@ ensures (this.myseq.int_size()==0 \&\& this.myseq[0]==Null)
30         @*/
31     public void clear(){
32         // vacia la lista.
33     }
34
35     /*@ requires 0>=index \&\& index < this.myseq.int_size();
36         @ ensures this.myseq.int_size()==\old(this.myseq).int_size()\&\&
37         @ (\forallall int i ; 0 <= i \&\& i < this.myseq.int_size() ;
38         @ (((i==index)==>
39             this.myseq.get(i).value==elem)\&\&
40         @ (!i==index)==>
41         @ (thiz.myseq.get(i).value==\old(thiz.myseq).get(i).value)))
42         @ \result==\old(thiz.myseq).get(index).value ;
43         @*/
44     int set(int index, int elem){
45         // asigna elem en la pos. index retornando el valor contenido inicialmente.
46     }

```

Figura 2.11: Clases Node.java y LinkedList.java

$$\backslash\text{reach}(x;T;f)$$

para denotar al conjunto de nodos de tipo T alcanzables desde x a través del campo f . La operación $\text{has}(n)$ es aplicada a un conjunto de nodos para

indicar la pertenencia de `n` a dicho conjunto.

El invariante definido para `LinkedList` expresa que para todo nodo `n` de tipo `Node`

$$\backslash\text{forall Node } n$$

perteneciente al conjunto de nodos alcanzables desde el primer elemento de la lista

$$\backslash\text{reach}(\text{this.head};\text{Node};\text{next}).\text{has}(n)$$

se cumple que el mismo nodo `n` no es alcanzable desde su sucesor

$$\!\backslash\text{reach}(n.\text{next};\text{Node};\text{next}).\text{has}(n)$$

Es decir, el invariante establece la condición de aciclicidad de la lista.

Dentro de la clase `LinkedList` también se incluye el atributo `myseq` que define una secuencia de objetos JML (línea 14) de tipo `JMLObjectSequence`. Para esto se usa el modificador `model`. El atributo `myseq` es utilizado para propósitos de la especificación. Seguidamente, se encuentra la expresión `represents myseq such_that` (líneas 15-20) la cual asocia a cada lista concreta `LinkedList` con la secuencia de nodos que representa (`myseq`). Para su definición se utilizan las expresiones `\reach` y `\forall`, mencionadas anteriormente, junto a `\int_size()` que al aplicarse a una colección retorna la cantidad de elementos que contiene. Inicialmente se establece que el tamaño de la secuencia debe coincidir con la cardinalidad del conjunto de nodos alcanzables desde el primer elemento de la lista asociada (línea 16). Luego, si el tamaño de `myseq` es mayor a cero, entonces el primer elemento de la secuencia coincide con el `head` de la lista `this` (línea 17). Además, si la lista no es vacía también se debe verificar que para todo nodo perteneciente a `myseq` (a excepción del último), su campo `next` coincide con el nodo ubicado en la posición siguiente en `myseq` (líneas 18 y 19). Esta es la función típica de abstracción que representa a una lista encadenada como la secuencia de nodos que contiene [63]. Dicha función de abstracción simplifica las especificaciones de los métodos de la clase pudiendo así éstos definirse mediante operaciones sobre la secuencia de nodos `myseq` vinculada a lista concreta `this`.

Para la definición de los contratos de los métodos se utilizan las cláusulas JML `requires` para las precondiciones, y `ensures` para las postcondiciones. En el ejemplo de `LinkedList` se pueden observar estos contratos previo a

las definiciones de los métodos. Aquellos métodos que no requieren condición alguna para ejecutarse, no poseen cláusula `requires`. En las postcondiciones se utilizan expresiones tales como

$$\backslash\text{exists } T \ x; P(x); H(x)$$

(cuantificador existencial con una semántica dual a `\forall`).

Para denotar el valor de la variable previo a la ejecución del método se utiliza `\old` y para referenciar el valor de retorno del método se utiliza `\result`, entre otros. JML también permite especificar comportamientos excepcionales, e importar y refinar especificaciones. Para mayores detalles del lenguaje se recomienda acceder a la descripción completa de JML [60].

Veamos a modo de ejemplo el contrato definido para el método `set(int index, int elem)` de `LinkedList` (líneas 35-41). Este método actualiza el valor almacenado en el nodo ubicado en la posición `index` de la lista por `elem`, retornando el valor almacenado inicialmente en esa posición. La precondition expresa que `index` debe ser una posición válida en la lista (línea 35).

La postcondición establece que la cantidad de nodos en la lista se mantiene igual (línea 36). Además se asegura que el valor almacenado en la posición `index` de la lista es actualizado con `elem` (línea 38) y que el resto de los valores almacenados en la lista no se modifican (línea 39). Finalmente (línea 40) se establece que el valor de retorno (`\result`) será el valor original almacenado en la posición `index`.

En sus comienzos JML surgió con el objetivo de proveer soporte para chequear en tiempo de ejecución el cumplimiento de los contratos de un programa [23]. A continuación se mencionan algunas herramientas que realizan chequeo estático de aserciones JML soportando diferentes niveles de automatización y de expresividad. ESC/Java y ESC/Java2 [13, 24] son verificadores de contratos en tiempo de ejecución de programas Java. Estas herramientas pueden automáticamente detectar ciertos tipos de errores comunes en el código fuente Java verificando el cumplimiento de aserciones simples. JACK [6] provee un ambiente de verificación de programas Java y Java Card [5] utilizando anotaciones JML. JACK Implementa un cálculo automático de *Weakest Precondition* (precondición más débil) con el objeto de generar obligaciones de prueba en programas Java a partir de anotaciones JML. LOOP [46, 53] es otra herramienta que traduce código Java anotado con expresiones JML a obligaciones de prueba. Una vez generadas estas obligaciones es posible utilizar algún probador de teoremas para la verificación de propiedades. La misma se realiza aplicando la lógica de Hoare [52] y cálculo

de *Weakest Precondition* [51]. KeY [10] es otra herramienta de verificación de programas Java con especificaciones JML. En este caso se utiliza una lógica dinámica para probar la corrección de propiedades derivadas de las especificaciones JML.

2.5. Traducción de especificaciones JML a Alloy

Como se ha explicado en las secciones anteriores, las técnicas propuestas en esta tesis aplican SAT solving sobre modelos Alloy para computar *workarounds transitorios* que posibiliten la recuperación de rutinas defectuosas en módulos de software. Para la construcción de estos modelos se requiere traducir las especificaciones JML incluidas en los programas Java a fórmulas Alloy. Este procedimiento fue presentado en trabajos previos [28, 67] enfocados en la verificación formal (acotada) de programas Java a través de SAT.

A continuación se retoma el ejemplo de `LinkedList` de la sección anterior (figura 2.11) mediante el cual se ilustrará el procedimiento de traducción.

En primer lugar la traducción genera las firmas, predicados y funciones básicas comunes a todos los modelos. Se incluyen las firmas para booleanos, enteros, secuencias, etc. También se definen predicados y funciones para expresiones específicas JML. Por ejemplo la expresión de alcanzabilidad:

$$\backslash\text{reach}(n, \text{Node}, \text{next})$$

es traducida a la función Alloy a:

$$\text{fun_reach}[n, \text{Node}, \text{next}] \text{ (figura 2.12)}$$

La función de alcanzabilidad queda instanciada como $n.*\text{next} \ \& \ \text{Node}$, es decir, por un lado se computa la composición (\cdot) de n con la clausura reflexo-transitiva de next ($*\text{next}$) y luego se realiza la intersección ($\&$) con el conjunto `Node` (esto es para descartar a `Null`). Al invocarse la función:

$$\text{fun_reach}[\text{this.head}, \text{Node}, \text{next}]$$

a partir del primer elemento de la lista `this.head`, el resultado será el conjunto de todos los nodos alcanzables en dicha lista excluyendo a `Null`.

Siguiendo con el ejemplo de `LinkedList` durante la traducción se generan las firmas `Node` y `LinkedList` (figura 2.13). Se define también la firma `Null` con el modificador `one` (para la constante `Null`).

```

fun fun_reach[ h:Node,
              type:set Node,
              field:Node -> (Node+Null)]:set Node{
    h.*(field ) & type
}

```

Figura 2.12: Función Alloy `fun_reach` para `LinkedList`.

```

one sig Null {}

sig LinkedList{}

sig Node{}

```

Figura 2.13: Signaturas `Null`, `LinkedList` y `Node`.

En las figuras 2.14a y 2.14b se muestra la cláusula JML `invariant` con su correspondiente traducción al predicado `invariantLinkedList`.

El predicado `invariantLinkedList` recibe como parámetros a `this` (correspondiente a la referencia `this` de Java) y las relaciones `head` y `next`. Los atributos `next` y `value` (pertenecientes a la clase `Node`) y `head` (perteneciente a la clase `LinkedList`) son mapeados a relaciones que se pasan como parámetros a los predicados. Para el caso de `invariantLinkedList` el parámetro `value` no es requerido ya que el predicado define la condición de aciclicidad de los nodos pertenecientes a las listas, no haciendo referencia a los valores contenidos en esos nodos.

El cuantificador universal (`\forall Node n`) incluido en la figura 2.14a se traduce a (`all n:Node`). Luego la siguiente línea:

```
!\reach(this.head,Node,next).has(n)
```

es traducido al predicado Alloy:

```
not (n in fun_reach[this.head,Node,next])
```

en donde `has(n)` (que define la pertenencia de `n` a una colección) es correspondido a una inclusión (`in`) en Alloy, el operador de navegación (`.`) de un

```
/*@ invariant (\forall Node n;  
  @ \reach(this.head, Node, next).has(n);  
  @ !\reach(n.next, Node, next).has(n));  
  @*/
```

(a) Cláusula JML invariant

```
pred invariant_LinkedList[thiz:LinkedList,  
  head:LinkedList->one(Node+Null),  
  next:Node->one(Node+Null)]{  
  
  all n:Node | {  
    n in fun_reach[thiz.head,Node,next]  
    implies  
      not ( n in fun_reach[n.next,Node,next])  
  }  
}
```

(b) Predicado Alloy invariant_LinkedList

Figura 2.14: (a) Cláusula JML invariant y (b) su traducción al predicado invariant_LinkedList para LinkedList.

Secuencias JML (JMLObjectSequence)	Secuencias Alloy (Seq)	Descripción
s.int_size()	#s	Tamaño de la secuencia.
s.get(i)	s[i]	Se obtiene el elemento en la posición i de la secuencia.
s.insertBeforeIndex(index,e)	s.insert[index,e]	Inserta e en la posición index.
s.isEmpty()	s.isEmpty	verdadero si y sólo si la secuencia es vacía.
s.has(e)	e in s.elems	verdadero si y sólo si e pertenece a la secuencia.

Cuadro 2.1: Traducción de algunas operaciones de `JMLObjectSequence` a Alloy.

objeto se traducido a una composición relacional (`.`), y la negación de una expresión JML (`!`) es traducida a la negación en Alloy (`not`). Como se mencionó en la sección anterior, el invariante expresa la condición de aciclicidad en donde todo nodo alcanzable desde el primer elemento de una lista a través de `next` no deberá ser alcanzable desde si mismo mediante `next`.

En la clase `LinkedList` también se incluye el campo `myseq` sobre el que se define la función de abstracción de `LinkedList` a `JMLObjectSequence` (secuencia de objetos JML) (figura 2.15a). El procedimiento de traducción genera en este caso el predicado Alloy `abstraction_sequence` (figura 2.15b). Este predicado recibe como parámetros a `this`, `head` (utilizada para determinar el primer elemento de la lista), `next` (para establecer el encadenamiento de los nodos de la lista) y `value` (para definir los valores almacenados en los nodos). Además, recibe el parámetro adicional `myseq`, una relación que vincula a cada `LinkedList` con su abstracción como una secuencia de nodos en Alloy (de tipo `Seq Node`). Intuitivamente, este predicado define a la abstracción de una lista como la secuencia de nodos alcanzables de la lista, donde los nodos aparecen en la secuencia en el mismo orden que en la lista original. La signatura `Seq` de Alloy permite definir secuencias de elementos y aplicar operaciones ya definidas sobre ellas, lo que facilita la traducción de las especificaciones. En el cuadro 2.1 podemos observar algunas de las operaciones sobre `JMLObjectSequence` y su correspondiente operación en Alloy.

En el cuerpo del predicado se establecen las condiciones de la relación (funcional) `myseq`. La primera de ellas resulta de la traducción de la expresión JML:

```
this.myseq.int_size()==\reach(this.head,Node,next).int_size()
```

en el predicado Alloy:

```
#thiz.myseq = #fun_reach[thiz.head,Node,next]
```

Esto define que la cantidad de nodos contenidos en la secuencia `thiz.myseq` debe ser igual a la cantidad de nodos alcanzables desde la `LinkedList` `thiz` asociada. Como se puede observar `int_size()` se traduce en `#` (cuadro 2.1). El operador `#` es aplicado sobre la secuencia `thiz.myseq` y sobre el set obtenido a través de `fun_reach`. Como secuencias y conjuntos se traducen a relaciones en Alloy, puede utilizarse el mismo operador.

La segunda condición incluida en el predicado `abstraction_sequence` resulta de la traducción de la expresión:

```
(this.myseq.int_size()>0) ==> (this.myseq.get(0)==this.head)
```

en la expresión Alloy:

```
(#thiz.myseq > 0) implies (thiz.myseq[0]=thiz.head)
```

En este caso se muestra que la implicación `==>` JML es transformada a su operador lógico equivalente en Alloy `implies`, y la expresión `this.myseq.get(0)` (cuadro 2.1) es traducida en el predicado `thiz.myseq[0]`.

Por último, dentro del predicado `abstraction_sequence` se incluye la traducción de:

```
(\forall int j; 0 <= j && j < this.myseq.int_size()-1 ;
  this.myseq.get(j).next == this.myseq.get(j+1));
```

como la siguiente expresión Alloy:

```
all j:Int | 0<=j and j< #thiz.myseq-1
  thiz.myseq[j].next= thiz.myseq[j+1]
```

El predicado es generado aplicando las traducciones de los operadores ya mencionados anteriormente. El predicado establece que en toda posición `j` comprendida entre 0 y el tamaño de la secuencia menos 1 se verifica que el nodo ubicado en la posición `j+1` en la secuencia debe ser igual al nodo siguiente de `thiz.myseq[j]`. Esta última condición establece que el orden de los nodos en la `LinkedList` se mantiene en la secuencia.

```

/*@ public model instance JMLObjectSequence myseq;
  @ public represents this.myseq \such_that
  @(this.myseq.int_size()==\reach(this.head,Node,next).int_size())
  @(this.myseq.int_size() > 0 ==> this.myseq.get(0) == this.head )
  @(\forall int j; 0 <= j && j< this.myseq.int_size()-1 ;
  @ this.myseq.get(j).next == this.myseq.get(j+1));
  @*/

```

(a) Función de abstracción JML para LinkedList

```

pred abstraction_sequence [
  thiz: LinkedList,
  myseq: LinkedList->one (seq Node),
  head: LinkedList ->one(Node + Null),
  next: Node ->one(Node + Null),
  value:Node->one(Int) ]{

  #thiz.myseq = #fun_reach[thiz.head,Node,next] and
  (thiz.head = Null implies #thiz.myseq = 0) and
  #thiz.myseq > 0 implies (thiz.myseq[0]=thiz.head and
  all j:Int | (0<=j and j< #thiz.myseq-1)
    implies
      thiz.myseq[j+1] = thiz.myseq[j].next
  )
}

```

(b) Predicado de abstracción abstraction_sequence

Figura 2.15: (a) Campo de la especificación myseq y (b) su correspondiente traducción al predicado abstraction_sequence.

Finalmente se presenta la traducción del contrato JML definido para el método

```
int set(int index, int elem)
```

de LinkedList. Recordemos que este método actualiza el valor del nodo

ubicado en la posición `index` con el valor `elem`, retornando el valor inicial almacenado en esa posición. Veamos a continuación la traducción del contrato de este método. La cláusula `requires` del método `set` se define con la expresión:

```
index => 0 && index < this.myseq.int_size()
```

la cual es traducida a la expresión Alloy:

```
index >= 0 and index < #this.myseq
```

El resultado de esta traducción es presentado en el predicado `pre_set` (figura 2.16a). En este se establece que `index` debe ser una posición válida en la secuencia. Notar como el uso de la función de abstracción `myseq` simplifica la especificación del método. Este es en general el propósito de las funciones de abstracción.

Por otro lado, para la cláusula `ensures` se genera el predicado `post_set` (figura 2.16b). Este predicado recibe como parámetros a `this`, `elem`, `myseq`, `myseq'`, `value`, `value'`, `index` y `return` (valor de retorno del método). Recordemos que las variables primadas en Alloy hacen referencia al estado de la relación luego de la ejecución del método asociado. Por lo tanto, se agrega en la traducción una variable primada como parámetro por cada campo de la clase que el método modifique (en este caso `myseq'` y `value'`). El objetivo es modelar el estado inicial de los objetos antes (variables sin primar) y después (variables primadas) de la ejecución del método. El predicado `post_set` es generado a partir de las siguientes expresiones JML:

```
this.myseq.int_size() == \old(this.myseq).int_size()
```

traducida a:

```
#this.myseq' = #this.myseq
```

La expresión JML `\old` puede ser utilizada dentro de las especificaciones de las postcondiciones de los métodos y hace referencia al valor de la variable previo a la ejecución del método. Es por esto que `\old(v)` se traduce a `v` (sin primar) en Alloy, mientras que cuando la variable `v` es utilizada sin `\old` en una postcondición hace referencia al valor obtenido luego de ejecutar el método y se traduce a `v'`. En este caso la expresión `\old(this.myseq)` es

traducida a la expresión `this.myseq`, y la expresión `this.myseq` es traducida a la expresión `this.myseq'`. La condición aquí definida establece que luego de ejecutar el método, la cantidad de nodos de la secuencia no se modifica.

A continuación la expresión:

```
thiz.myseq.get(index).value==elem
```

es traducida a:

```
thiz.myseq'[index].value'=elem
```

La traducción nuevamente asocia a `this.myseq.get(index)` con la expresión `this.myseq'[index]` para acceder al nodo almacenado en la posición `index`. Luego mediante `value` se accede al valor almacenado en esa posición el cual es actualizado con `elem`. Luego la expresión:

```
\forall int i; 0<=i && i<this.myseq.int_size();
    !(i==index)==>
thiz.myseq.get(i).value==\old(thiz.myseq).get(i).value
```

es traducida a:

```
all i:int | { (0<=i and i<#this.myseq and
    !(i=index)) implies
thiz.myseq'[i].value'=thiz.myseq[i].value}
```

El predicado resultante establece que para toda posición válida en la secuencia distinta de `index`, el valor almacenado en esa posición no cambia.

Finalmente, la última expresión JML en la postcondición define que el valor de retorno será el valor almacenado en el nodo antes de ser actualizado por `elem`.

```
\result==\old(thiz.myseq).get(index).value
```

Para la expresión anterior se genera:

```
return=thiz.myseq[index].value
```

En este caso, el valor de retorno `\result` es traducido al parámetro `return`.

```

/*@ requires 0 >= index && index < this.myseq.int_size();
   @ ensures this.myseq.int_size() == \old(this.myseq).int_size() &&
   @ this.myseq.get(index).value == elem &&
   @ (\forall int i; 0 <= i && i < this.myseq.int_size();
   @ (! (i == index) ==>
   @ (this.myseq.get(i).value == (\old(this.myseq).get(i)).value)) &&
   @ \result == (\old(this.myseq).get(index)).value;
   @*/
int set(int index, int elem){
    ...
}

```

(a) Especificación JML para `set(int index, int elem)`

```

pred pre_set[ thiz:LinkedList,
              myseq:LinkedList->(seq Node),
              index:Int]{
    index >= 0 and index < #thiz.myseq
}

pred post_set[myseq,myseq':LinkedList->(Seq Node),
              value,value':Nodo->lone( Int),
              index,elem:int,
              thiz:LinkedList,
              return:int]{
    #thiz.myseq=#thiz.myseq' and

    thiz.myseq'[index].value'=elem

    all i:int | {
        (!(i==index) implies thiz.myseq'[i].value'=thiz.myseq[i].value)
    } and

    return=thiz.myseq[index].value
}

```

34

(b) Predicados `pre_set` y `post_set`.

Figura 2.16: (a) Especificación JML para `set(index,e)` y (b) su traducción a Alloy.

2.6. Workarounds

Un *workaround* para una rutina defectuosa r se define como una secuencia de una o más rutinas que produce un resultado equivalente a r de acuerdo con su especificación. De esta forma si falla la ejecución de la acción r , mediante un *workaround* se dará una alternativa de ejecución que partiendo del estado de invocación de r , pueda obtenerse un estado que satisfaga su postcondición. El uso de *workarounds* en la recuperación de sistemas ha sido ya utilizado por otros autores [4, 14], ellos presentan enfoques vinculados a sistemas *self-healing* [54] y tolerantes a fallas [77].

El mecanismo de recuperación de sistemas mediante *workarounds* presume la existencia de redundancia intrínseca en los módulos de software bajo análisis [18, 44]. Este tipo de redundancia es propia de sistemas modulares donde la reusabilidad es una de las características más deseadas. Por otro lado muchos componentes son diseñados de acuerdo a las necesidades específicas de diversas aplicaciones clientes. Esto provoca que puedan existir variantes de una misma funcionalidad a fin de atender las necesidades particulares de cada cliente. En otros casos los desarrolladores mantienen varias implementaciones de una misma funcionalidad en bibliotecas para asegurar su compatibilidad con versiones anteriores. Por ejemplo la biblioteca estándar de Java 6 ² contiene 45 clases y 365 métodos `deprecated` que replican total o parcialmente funcionalidades de métodos contenidos en clases nuevas. La redundancia en el software también suele presentarse como consecuencia de la implementación de interfaces y/o especialización de clases. Un ejemplo es el caso de `java.util.Stack` ³ de Java que hereda de `java.util.Vector` ⁴ métodos de inserción y eliminación de elementos, y luego en su implementación, define otros específicos para el tipo abstracto de datos (TAD) *Stack* como `push(e)` y `pop()`. La eficiencia es otro de los factores por lo que en muchas ocasiones los desarrollares incorporan redundancia en el software. Ejemplo de ello es la implementación de diferentes algoritmos de ordenamiento en un misma biblioteca, o bien diferentes implementaciones de TADs como por ejemplo `java.util.HashSet`, `java.util.TreeSet` de Java para la *interface Set* ⁵.

La automatización del proceso de recuperación a través de *workarounds* involucra: la detección de fallas en tiempo de ejecución, la recuperación del

²<https://www.doc.ic.ac.uk/csg-old/java/jdk6docs>

³<https://docs.oracle.com/javase/6/docs/api/java/util/Stack.html>

⁴<https://docs.oracle.com/javase/6/docs/api/java/util/Vector.html>

⁵<https://docs.oracle.com/javase/6/docs/api/java/util/Set.html>

sistema a un estado previo libre de fallas, el cómputo y la selección automática de una alternativa que permita evitar la falla y su posterior ejecución a partir del estado recuperado. Este proceso es definido en la literatura como *workarounds automáticos* [18].

2.7. Workarounds automáticos sobre aplicaciones Java

En [15] se presenta ARMOR, una herramienta que implementa *workarounds automáticos* para recuperar programas Java ante la ocurrencia de fallas. ARMOR toma como entrada un programa Java y una API. Inicialmente la herramienta identifica en el código fuente las llamadas a las rutinas de la API que podrían ser recuperadas eventualmente mediante *workarounds automáticos*. Luego, el programa es instrumentado agregando *Roll-Back Areas (RBAs)* sobre las invocaciones de esas rutinas. Cada *RBA* agrega puntos de recuperación de modo tal que si la ejecución del programa produce una falla se podrá realizar una acción de *rollback* a un estado previo del sistema desde donde se ejecutará un *workaround*.

Para la detección de fallas ARMOR utiliza las excepciones producidas por las mismas rutinas (aunque podrían incorporarse mecanismos basados en especificaciones formales fácilmente). ARMOR asume que los *workarounds* vienen dados por el desarrollador o usuario de la herramienta. Si bien en el artículo que presenta ARMOR los *workarounds* han sido expresados a través de un conjunto de reglas de reescritura de términos, los mismos podrían ser definidos a través de otros formalismos como por ejemplo máquinas de estados o especificaciones algebraicas.

ARMOR implementa un proceso de recuperación donde las técnicas propuestas en esta tesis podrían integrarse sin problemas. La idea es que luego de hacer *rollback* a al estado de invocación de la rutina que provocó la falla, ARMOR puede invocar al procedimiento de cómputo de *workarounds transitorios* propuesto en la técnica 1 (capítulo 3), con el estado de invocación mencionado, para computar una secuencia de métodos alternativa cuya ejecución sirva para continuar con la ejecución del sistema. Otra opción sería invocar a la técnica 2 propuesta (capítulo 4), para la construcción de un estado de recuperación que permita continuar con la ejecución.

Capítulo 3

Cómputo de secuencias de rutinas para workarounds transitorios

En este capítulo se presenta la primera técnica de cómputo de *workarounds transitorios* a partir de especificaciones de programas. Esta técnica se aplica sobre módulos Java anotados con especificaciones JML que describen el comportamiento de las rutinas componentes. La propuesta hace uso de la información presente en el contexto de ejecución del sistema previo a la invocación de un método defectuoso. El procedimiento de recuperación genera automáticamente un programa Dynalloy que aplica SAT como procedimiento de decisión permitiendo encontrar *workarounds transitorios*. Los tiempos de cómputo de *workarounds* obtenidos en la evaluación experimental (capítulo 5) son satisfactorios y fundamentan su aplicación en tiempo de ejecución.

Para ilustrar la técnica se presenta un escenario de ejecución en donde un programa Java invoca a un método de la API `LinkedList` (figura 2.11) produciéndose una falla. Luego, se aplica una acción de *rollback* al estado previo a la ejecución del método defectuoso y se ejecuta el procedimiento de cómputo de *workarounds transitorios* propuesto.

Finalmente se presenta una optimización a la técnica basada en *esquemas de workarounds* obteniendo mejoras significativas en los tiempos de búsqueda de *workarounds* de acuerdo a lo relevado en la etapa experimental.

3.1. Codificación del estado inicial en Alloy

Las técnicas propuestas en esta tesis computan *workarounds transitorios* a partir de un estado concreto de ejecución. Para ello es necesario codificar el estado del programa Java a recuperar previo a la llamada de la rutina defectuosa. Una vez detectada la violación del contrato del método y luego de una acción de *rollback* al estado de invocación, se aplica Java Reflection [34] con el fin de inspeccionar los objetos creados en tiempo de ejecución y generar un predicado Alloy que caracterice a dicho estado. El predicado resultante se denomina `initialState` y sus parámetros son relaciones que se extraen automáticamente a partir de los campos de las clases involucradas. Para el ejemplo de `LinkedList` visto en la sección 2.5, se generan las relaciones `this`, `head`, `next` y `value`. En la figura 3.1 se muestra el código fuente de un programa Java que se asume falla al ejecutar el método `set(index,elem)` en la línea 9, debido a un defecto en el código del método. El estado de la `LinkedList` `lis` previo a la invocación de `set` se muestra en la figura 3.2.

```

1  import LinkedList;
2  public class Principal{
3      public static void main(String [] args){
4          LinkedList lis = new LinkedList();
5          int i = 0;
6          int v = 4;
7          lis.add(10);
8          lis.add(12);
9          int res = lis.set(i,v);
10     }
11 }
```

Figura 3.1: Escenario de falla para el método `set(index,elem)`.

El predicado `initialState` (figura 3.4) es construido a partir de los objetos Java creados al momento de la invocación del método defectuoso. Para ello se recorren estos objetos aplicando un algoritmo de recorrido *Breadth First (BF)*. Durante el recorrido se etiquetan los objetos con identificadores únicos l_0, l_1, \dots, l_i para objetos de tipo `LinkedList`, y n_0, n_1, \dots, n_i para objetos de tipo `Node` (figura 3.2) a fin de obtener una versión canónica de la estructura [29,38]. Para cada identificador asignado a un objeto se define una

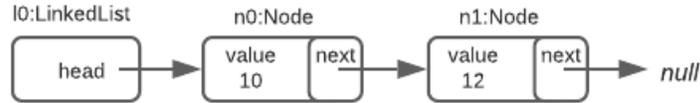


Figura 3.2: Estado de ejecución Java.

constante en la signatura asociada en el modelo Alloy. Por ejemplo para los objetos de tipo `LinkedList` se definen constantes de signatura `LinkedList`, para los objetos de tipo `Node` se definen constantes de signatura `Node`, etc. Los nombres de las constantes siguen la misma lógica de las etiquetas asignadas a los objetos, es decir, para los objetos n_0, n_1, \dots, n_i se definen las constantes N_0, N_1, \dots, N_i (figura 3.3) en el modelo Alloy.

Al mismo tiempo que se realiza el recorrido, se obtienen los pares ordenados a insertar en las relaciones Alloy correspondientes a los campos de la estructura. El acceso a los valores almacenados en los objetos es realizado mediante técnicas de `Reflection` [34, 62].

En el ejemplo presentado en la figura 3.2 se recorre la estructura a partir del objeto raíz etiquetado con `l0` generando la constante `L0` (figura 3.3) en la signatura `LinkedList`. Luego, examinando `l0` se encuentra el campo `head` que hace referencia a un objeto de tipo `Node` con etiqueta `n0`. Esto se agrega a `initialState` como:

```
head = L0->N0 and this = L0
```

donde `N0` es una constante de signatura `Node` definida para el objeto `n0`. Del mismo modo se definirá a `N1` como otra constante de signatura `Node` para el objeto `n1` (figura 3.3).

```
sig LinkedList{}
one sig L0 extends LinkedList{}
sig Node{}
one sig N0,N1 extends Node{}
```

Figura 3.3: Signaturas `LinkedList` y `Node`.

El objeto `n0` posee en su campo `value` el valor entero 10 y en su campo `next` la referencia al objeto `n1`. Como consecuencia se incluyen los pares (`N0->10`) en la relación `value` y (`N0->N1`) en la relación `next`. Continuando con el recorrido, se analiza el objeto `n1` encontrando en su campo `value` el valor 12 y en su campo `next` el valor `null`. Esto incorpora los pares (`N1->12`) y (`N1->null`) en las relaciones `value` y `next` respectivamente. Al finalizar el recorrido, las relaciones `value` y `next` quedan definidas como:

```
value = (N0->10)+(N1->12)
next = (N0->N1)+(N1->null)
```

Finalmente para los parámetros de invocación `i=0` y `v=4` (figura 3.1) se definen a `pint0` y `pint1` (de signatura `Int`) incluidos en `initialState` como:

```
pint0 = 0 and pint1 = 4
```

```
pred initialState[ thiz:LinkedList,
    value:Node -> one (Int),
    head:LinkedList->one(Node+null),
    next: Node->one(Node+null),
    pint0:Int,
    pint1:Int] {
  head = L0 ->N0 and thiz=L0 and
  value = (N0->10) + (N1->12) and
  next = (N0->N1) + (N1->null) and
  pint0 = 0 and pint1 = 4
}
```

Figura 3.4: `initialState` para `lis.set(i,v)`.

3.2. Acciones atómicas DynAlloy a partir de especificaciones de métodos Java

La técnica de cómputo de *workarounds* propuesta en este capítulo está basada en la construcción de un programa DynAlloy. Para ello se requiere de

las traducciones de las especificaciones de los contratos JML de cada método $m_i[\vec{p}]$ de la clase (explicada en la sección 2.5) donde \vec{p} denota a la secuencia de parámetros p_0, \dots, p_n . De las traducciones se obtienen los predicados:

$$\text{pre}_{m_i}[\text{this}, \vec{f}, \vec{x}_i, \text{abs}] \text{ y } \text{post}_{m_i}[\text{this}, \vec{f}, \vec{f}', \vec{x}_i, \vec{x}_i', \text{ret}', \text{abs}]$$

que luego serán utilizados como precondition y postcondition de la acción atómica a_i (figura 3.5).

```

act  a_i [this,  $\vec{f}$ ,  $\vec{x}_i$ , ret, abs, action] {
    pre { premi[this,  $\vec{f}$ ,  $\vec{x}_i$ , abs] }
    post { postmi[this,  $\vec{f}$ ,  $\vec{f}'$ ,  $\vec{x}_i$ ,  $\vec{x}_i'$ , ret', abs] and
          action'=ai-action}
}

```

Figura 3.5: Acción atómica a_i asociada al método m_i

A los fines de simplificar la presentación de la acción atómica a_i se omiten los tipos de los parámetros. Como fue explicado en la sección 2.3 las variables primadas en a_i son utilizadas para hacer referencia al estado de las mismas luego de ejecutar la acción.

Los parámetros $this$ y \vec{f} en a_i modelan el objeto `this` y los campos de los objetos no primitivos relacionados al método m_i . En el ejemplo de `LinkedList`, $this$ es de tipo `LinkedList`, y en \vec{f} se encuentra la relación `value` que dado un `Node` establece el valor entero almacenado en el mismo.

Para los parámetros \vec{p} (propios de m_i) se generan los parámetros \vec{x}_i (donde $\vec{x}_i = x_0, \dots, x_n$). En la acción `set` (figura 3.6b) \vec{x}_i se corresponde a los parámetros `index` y `elem` (los que determinan el índice y el elemento a insertar).

El valor de retorno de a_i se almacena en el parámetro `ret`. Las acciones que no retornan un valor, como por ejemplo `clear` (figura 3.8b), no cuentan con este parámetro.

La acción a_i también recibe como parámetro a la función de abstracción `abs` (explicada en la sección 2.5). En el caso de `LinkedList` la abstracción queda establecida por `abs` que vincula a una `LinkedList` con una secuencia de nodos (`seq Node`).

El último parámetro en a_i es `action`, de tipo `ActionExecuted`, donde se registra la ejecución de la acción (esto queda expresado en la postcondition de a_i). Dicho registro permite construir la secuencia de métodos Java

```

pred pre_set[ thiz:LinkedList,
              abs:LinkedList->(seq Node),
              index:Int]{
  index >=0 and index < #thiz.abs
}

pred post_set[abs,abs':LinkedList->(Seq Node),
              value,value': Nodo->lone( Int),
              index,elem:int,
              thiz:LinkedList,
              ret_int:int]{
#thiz.abs=#thiz.abs' and
thiz.abs'[index].value'=elem
all i:int | {
  (!(i=index) implies thiz.abs'[i].value'=thiz.abs[i].value)
} and
ret_int=thiz.abs[index].value
}
    
```

(a) Contrato Alloy para set(int index, int elem)

```

act set[thiz:LinkedList,
        abs:LinkedList->(Seq Node),
        value: Nodo->lone( Int),
        index,elem:int,
        ret_int:int,
        action:ActionExecuted] {
  pre { pre_set[thiz,abs,index]}
  post { post_set[abs,abs',value,value',
                  index,elem,thiz,ret_int'] and
        action'=set_action
  }
}
    
```

(b) Acción set

Figura 3.6: (a) Contrato Alloy para set(index,e) y (b) su acción atómica set.

al final del proceso de búsqueda de *workarounds*. Durante la generación de las acciones se incorpora al modelo Alloy la signatura `ActionExecuted` (figura 3.7) desde donde se extenderán los distintos tipos de acciones (una por cada método). Cada acción atómica agrega en su postcondición el seteo del parámetro `action`. Por ejemplo la acción `set` agrega en su postcondición `action'=set_action`. Recordemos que el procedimiento de búsqueda de *workarounds* brindará como resultado una secuencia de acciones atómicas DynAlloy que deberá ser transformada a una secuencia de métodos Java. Este último paso consiste en transformar un modelo de una fórmula Alloy en un programa Java [3,39], para ello es muy útil el registro de las acciones que componen el *workaround*.

Retomando el ejemplo de `LinkedList`, en la figura 3.6 se muestran los predicados `pre_set`, `post_set` (figura 3.6a) y la acción atómica `set` (figura 3.6b) construidas automáticamente para el método `set(index,elem)` que reemplaza en la lista el entero ubicado en la posición `index` por `elem`.

En las figuras 3.8 y 3.9 se incluyen dos ejemplos adicionales correspondientes a las acciones `clear` y `contains`. Como se puede observar en la definición de las acciones de `LinkedList`, las relaciones `head` y `next` no son incluidas como parámetros. Esto se debe a que mediante `abs` se realiza la abstracción de `LinkedList` a `seq Node` y sobre esta última se definen las especificaciones.

```

abstract sig ActionExecuted{
    one sig set_action extends ActionExecuted{}
    one sig clear_action extends ActionExecuted{}
    one sig contains_action extends ActionExecuted{}
}
    
```

Figura 3.7: Signatura `ActionExecuted`

Cabe aclarar que las técnicas propuestas en esta tesis no realizan traducciones de las implementaciones de los métodos, sólo se procesan sus contratos. Se asume que el comportamiento de cada rutina o método queda establecido estrictamente por su especificación. Esto permite mejorar la eficiencia de la búsqueda de *workarounds transitorios* basada en SAT ya que no se requiere modelar los estados intermedios de ejecución de cada método. Por otro lado, en este trabajo estamos interesados en *workarounds* que se comporten como indica la especificación un método, independientemente de su implementación.

```

pred pre_clear[] { true_pred[] }

pred post_clear [ thiz:LinkedList,
                  abs, abs':LinkedList->Seq Node ] {
    #thiz.abs' = 0
}
    
```

(a) Contrato Alloy para clear

```

act clear [ thiz:LinkedList,
            abs:LinkedList->Seq Node,
            action:ActionExecuted ] {
    pre { pre_clear[] }
    post { post_clear [ thiz, abs, abs' ] and
          action' = clear_action
    }
}
    
```

(b) Acción clear

Figura 3.8: (a) Contrato Alloy para clear y (b) su acción atómica clear.

3.2.1. Generación de parámetros de tipos primitivos

Durante el procedimiento de búsqueda de *workarounds transitorios* para un método pueden requerirse de parámetros específicos de tipo primitivo en las acciones atómicas a_i involucradas en el procedimiento de cómputo. Por ejemplo, supóngase que dada la `LinkedList` presentada en la figura 3.2 se invoca el método `lis.getFirst()` el cual falla. Este método podría recuperarse ejecutando `lis.get(0)`. El parámetro actual para el método `get` no podría ser otro que 0. Supongamos ahora que se desea recuperar la invocación fallida de `lis.isEmpty()`. Un *workaround transitorio* para este caso podría darse ejecutando `contains(e)`, donde el parámetro entero `e` no debería pertenecer a `lis` para que el resultado sea equivalente a `lis.isEmpty()`.

Para que las acciones a_i puedan ser invocadas con los valores apropiados (dentro de los *scopes*) y que posibiliten la recuperación de la acción fallida, se definen las acciones atómicas `nonDetInt [i: Int]` y `nonDetBool [b: boolean]`

```
pred pre_contains[] {true_pred[] }

pred post_contains[thiz:LinkedList,
                  abs,abs':LinkedList->Seq Node,
                  value,value':Node->one int,
                  e:int,
                  ret_bool:boolean]{
  (ret_bool=true iff
   some i:Int|{0<=i and
              i<#thiz.abs and value[thiz.abs[i]]=e})
   and value=value' and thiz.abs=thiz.abs'
}
```

(a) Contrato Alloy para contains

```
act contains[ thiz:LinkedList,
             abs:LinkedList->Seq Node,
             value:Node->one int,
             e:int,
             ret_bool:boolean,
             action:ActionExecuted] {
  pre { pre_contains[] }
  post { post_contains[thiz,abs,abs', value,value',e,ret_bool']
        and action'=contains_action }
}
```

(b) Acción contains

Figura 3.9: (a) Contrato Alloy para contains(e) y (b) su acción atómica contains.

para la generación de valores enteros y booleanos respectivamente (figuras 3.10 y 3.11).

```
act nonDetInt[i:Int] {
    pre { }
    post {some x:int | i' = x}
}
```

Figura 3.10: Acción nonDetInt

```
act nonDetBool[b:boolean] {
    pre { }
    post { some x:boolean | b' = x}
}
```

Figura 3.11: Acción nonDetBool

Un *workaround* para la invocación fallida de `lis.isEmpty()` sobre la `LinkedList` presentada en la figura 3.2 podrá obtenerse invocando el programa:

```
nonDetInt[elem];
contains[thiz,abs,value,elem,ret_bool,contains_action]
```

La precondition y postcondition para este programa es determinada por el contrato a que deberá satisfacer la invocación de `lis.isEmpty()`. Es decir, `True` como precondition y `ret_bool=False` como postcondition, ya que el resultado de ejecutar `lis.isEmpty()` (siendo `lis` no vacía) debe retornar falso. El valor del parámetro `elem` es generado a través de `nonDetInt` y luego es pasado a la acción `contains`. Para cumplir con la postcondition, `elem` no deberá pertenecer a `lis`.

Se presenta a continuación un segundo ejemplo en donde la acción atómica involucrada requiere generar dos parámetros enteros. En este caso el *workaround transitorio* es computado para la invocación fallida de `lis.getFirst()` que retorna el elemento ubicado en la primera posición de `lis` (figura 3.2). El *workaround transitorio* utiliza la acción atómica `set(index,elem)`:

```
nonDetInt[index];nonDetInt[elem];
set[thiz,abs,value,index,elem,ret_int,action]
```

El contrato para este programa requiere que inicialmente la lista `lis` no sea vacía y que luego de su ejecución, el retorno `ret_int=10`. Los valores para los parámetros `index` y `elem` se generan al ejecutar `nonDetInt` en dos oportunidades. Para verificar el contrato, los valores generados deben ser: `index=0` y `elem=10`. De esta forma `set` actualiza el valor ubicado en la primera posición con el mismo valor contenido (lo que no produce cambios en `lis`) retornando este mismo valor.

Las valores generados para los parámetros dependen de las firmas del modelo y sus *scopes*. Para el tipo de dato entero involucrado en los casos de estudio se utilizó una implementación diferente a la provista por Alloy. La implementación de enteros utilizada permite representar enteros con una precisión de 32 bits y fue propuesta por Abad et al. [2]. Esto implica que todo parámetro entero requerido por una acción atómica será analizado para todo entero representable en 32 bits.

3.3. Programa DynAlloy para el cómputo de *workarounds*

En esta sección se presenta la definición del mecanismo de búsqueda de *workarounds transitorios*, parte central de la técnica.

Bajo el supuesto de la ocurrencia de una falla al ejecutar un método m_i en un estado dado (caracterizado por el predicado `initialState`), la técnica buscará construir una secuencia de acciones que se comporte según lo especificado en el contrato de la acción defectuosa a_i (asociada a m_i). En esta secuencia de acciones se descarta a a_i debido a que justamente se quiere obtener una alternativa de ejecución válida que excluya al método m_i . Una vez generadas las acciones atómicas a_0, a_1, \dots, a_n (sección 3.2) junto al estado inicial `initialState`, se construye el programa de cómputo de *workarounds* (figura 3.12).

El programa de cómputo de *workarounds* se define mediante la aserción DynAlloy `wac_finder`. Durante la verificación de esta aserción se intentará generar una secuencia de acciones tales que partiendo de la precondition y luego de ejecutar la secuencia, se verifique la postcondition. Debido a que la postcondition de `wac_finder` se define como la negación de la acción defectuosa, el contraejemplo obtenido será una alternativa de ejecución a la misma.

```

assertCorrectness
wac_finder[thiz,  $\vec{f}$ ,  $\vec{X}$ , ret_int, ret_bool, abs, aci,  $\vec{p}\vec{s}$ ] {
  pre={
    initState[thiz,  $\vec{f}$ ,  $\vec{p}\vec{s}$ ]  $\wedge$  premi[thiz,  $\vec{f}$ ,  $\vec{x}_i$ ]  $\wedge$ 
    abstraction[thiz,  $\vec{f}$ , abs]}
  program={ (
    nonDet[ $\vec{x}_0$ ]; a0[t, ret_int,  $\vec{f}$ ,  $\vec{x}_0$ , abs, ac] +
    nonDet[ $\vec{x}_1$ ]; a1[t, ret_bool,  $\vec{f}$ ,  $\vec{x}_1$ , abs, ac] +
    ...
    nonDet[ $\vec{x}_n$ ]; an[t,  $\vec{f}$ ,  $\vec{x}_n$ , abs, ac] ) * }
    donde ai  $\notin$  {a0, a1, ..., an}
  post={  $\neg$  postmi[thiz,  $\vec{f}$ ,  $\vec{x}_i$ , ret',  $\vec{f}'$ ,  $\vec{x}'_i$ ]  $\wedge$ 
    abstraction[thiz,  $\vec{f}'$ , abs']  $\wedge$ 
    invariant[thiz,  $\vec{f}'$ ] }
}
check wac_finder for <m> lurs <n>
    
```

Figura 3.12: Programa de cómputo de *workarounds transitorios* para a_i

A continuación se detallan los elementos incluidos en `wac_finder`. El parámetro `thiz` de la aserción se corresponde al objeto desde donde es invocada la acción defectuosa y \vec{f} hace referencia a los campos de los objetos relacionados. Los parámetros propios de cada acción se incluyen en `wac_finder` como $\vec{X} = [\vec{x}_0, \dots, \vec{x}_n]$, donde $\vec{x}_0, \dots, \vec{x}_n$ identifican a los parámetros de las acciones a_0, \dots, a_n . Para evitar posibles colisiones de nombres, los parámetros de las acciones serán renombrados en un paso de procesamiento previo a la búsqueda de *workarounds*. Por otro lado, el retorno se incluye mediante `ret_int` o `ret_bool` (dependiendo del tipo de retorno de la acción defectuosa). El parámetro `abs` establece la función de abstracción, y la acción defectuosa es definida en `aci` con el objetivo de que una vez encontrado el *workaround transitorio*, pueda reconstruirse la secuencia de acciones. Finalmente en $\vec{p}\vec{s}$ son almacenados los valores de invocación de la acción defectuosa. Estos valores son utilizados en `initialState`.

El contrato de la aserción es fijado por los predicados `pre` y `post` (figura 3.12). En la precondition se establecen las condiciones iniciales de a_i . Recordemos que se asume que a_i ocasionó una falla durante su ejecución, por lo tanto, su precondition es verdadera. Por otro lado, mediante el predicado

`initialState` se establece el estado de la estructura y los parámetros de invocación. Otro elemento que se exige en la precodición de la aserción es el cumplimiento del predicado `abstraction`. Como se explicó anteriormente, el predicado de abstracción permite simplificar las especificaciones de las acciones atómicas. En el ejemplo de `LinkedList` la función de abstracción vincula listas con secuencias, lo que permite la definición de los contratos utilizando secuencias.

La postcondición de `wac_finder` establece la negación de la postcondición de `ai` y el cumplimiento de los predicados `invariant` y de `abstraction`. El predicado `invariant` es requerido debido a que la estructura resultante debe verificar el invariante de representación. Por otro lado el predicado `abstraction` establece la correspondencia entre la instancia abstracta obtenida y una instancia concreta.

En el cuerpo de la aserción se define el programa mediante los operadores de iteración acotada (`*`) y de elección no determinística (`+`). El operador (`*`) establece un bucle de cero o más iteraciones. Dentro de ese bucle se busca una combinación de acciones de forma no determinística que sirva como *workaround* para la acción defectuosa. Cada acción dentro del programa es antecedida por una o más acciones `nonDet` (`nonDetInt` o `nonDetBool`) para la generación de los parámetros primitivos (en caso de que la acción lo requiera). La búsqueda de los *workarounds* se realiza en el momento en que DynAlloy Analyzer ejecuta el comando

```
check wac_finder for <m> lurs <n>
```

donde `n` y `m` definen los *scopes* para las firmas y los *loop unrolls*, respectivamente.

Debido al carácter acotado del análisis de SAT la no detección de contraejemplos no garantiza la inexistencia de *workarounds transitorios*. Una cantidad de *loop unrolls* muy pequeña podría no ser suficiente como para encontrar una combinación apropiada de acciones, mientras que una cantidad muy grande de *loop unrolls* puede ocasionar que el análisis de SAT no termine. A medida que la cantidad de *loop unrolls* crece, las fórmulas generadas crecen de manera exponencial haciendo en ocasiones que las mismas se vuelvan inmanejables a nivel de procesamiento computacional. Por este motivo es que el prototipo que implementa esta técnica ejecuta el chequeo de la aserción de manera incremental. Otro elemento que colabora en el proceso es la reducción del espacio de búsqueda al fijar el estado de invocación a

través del predicado `initialState`. A pesar de no tener garantías de encontrar un *workaround*, en los experimentos realizados se observa que en una gran cantidad de casos es posible identificar ejecuciones alternativas usando este enfoque (capítulo 5).

3.4. Algoritmo de cómputo de workarounds transitorios

En esta sección se presenta un pseudocódigo (algoritmo 1) del procedimiento de búsqueda completo. Para la evaluación experimental de la técnica fue desarrollado un prototipo en lenguaje Python. Las técnicas resultantes de esta tesis asumen un mecanismo de monitoreo, detección de fallas y aplicación de una acción de *rollback* a un estado previo seguro (como el descrito en ARMOR [15]).

Para una mejor comprensión del algoritmo se describen a continuación las funciones incluidas:

- `translate_to_alloy(jml_exp)`: Función encargada de traducir una expresión JML a un predicado Alloy (sección 2.5).
- `generate_alloy_state(run_state, clase_Java)`: Extrae del estado de ejecución Java (`run_state`) los objetos creados en el instante previo a la invocación de la acción defectuosa.
- `get_params(p)`: Extrae del predicado Alloy `p`, resultante de la traducción de un método Java anotado en JML, las variables correspondientes a los parámetros de dicho método.
- `get_pre(p)` y `get_post(p)`: Dado un predicado Alloy `p` correspondiente a un método Java, la función extrae su precondition y postcondition respectivamente (ejemplo en figura 3.6a).
- `gen_dynalloy_action(m, params, pre, post)`: Esta función toma como entrada al método `m`, sus parámetros `params`, el contrato de `m` (`pre` y `post`) y construye la acción atómica DynAlloy asociada (sección 3.2).
- `gen_dynalloy_prog(assert_params, actions, mi, initial_state, abs_alloy, inv_alloy)`: Esta función es la encargada de construir la

aserción `wag_finder` (sección 3.3). Para generar la aserción, la función utiliza las acciones atómicas Dynalloy `actions` (resultantes de la traducción de los contratos de los métodos de la clase). La función también recibe como parámetro al método defectuoso `mi`, los predicados `initial_state`, `abs_alloy` e `inv_alloy` los cuales establecen las condiciones que debe cumplir el estado inicial, la función de abstracción y el predicado invariante de la clase. Finalmente `assert_params` establece los parámetros de `wag_finder` (estructura de datos corriente y campos de la misma, junto a los parámetros propios de las acciones).

- `run_dynalloy_translator(dynalloy_program, curr_unrolls)`: Invoca a DynAlloy Translator para generar un modelo Alloy fijando la cantidad de `loop unrolls` en `curr_unrolls`.
- `run_alloy_analyzer(alloy_spec)`: Función que invoca a Alloy Analyzer retornando la tupla `<res, model>` siendo `res` igual a `SAT` o `UNSAT`. En el caso que `res` sea igual a `SAT`, en `model` quedará almacenada la instancia que satisface el modelo.
- `translate_to_java(model, java_class, actions)`: Construye el programa Java (`workaround`) a partir de una instancia (`model`) Alloy asociada a un contraejemplo obtenido al intentar verificar `wac_finder` [3, 39].

El algoritmo 1 recibe como entrada a la clase Java con sus campos, métodos y especificaciones JML. También toma como entrada: el método `mi` que ha provocado la falla, el estado de ejecución Java previo a la ocurrencia de la misma, los `unrolls` (cantidad máxima de iteraciones a considerar), `scope` (alcance o cantidad de átomos a considerar por cada signatura Alloy) y `timeout` (tiempo máximo aceptable para el análisis).

El procedimiento implementado por el algoritmo 1 genera inicialmente a través de `translate_to_alloy` los predicados Alloy para la función de abstracción y el invariante (líneas 1 y 2). En la línea siguiente se construye el predicado Alloy `initial_state`. Luego se inicializa la lista `actions` donde se almacenarán las acciones atómicas vinculadas a los métodos de la clase, y el conjunto `assert_params` en donde se registrarán los parámetros de las acciones involucradas (líneas 4 y 5).

El ciclo `foreach` comprendido entre las líneas 6 y 12 recorre la lista de métodos de la clase (`m0, . . . , mn`) traduciendo los contratos JML de cada uno

```

Input : Clase C con campos  $f_0, \dots, f_n$ , invariante inv en
          JML, func.de abstracción abs, métodos  $m_0, \dots, m_j$ 
          anotados en JML, método fallido  $m_i \in m_0, \dots, m_j$ ,
          estado inicial java_state, unrolls, scope, timeout;
Output: Workaround wk + UNSAT
1 abs_alloy = translate_to_alloy(abs);
2 inv_alloy = translate_to_alloy(inv);
3 initial_state = generate_alloy_state(java_state, C);
4 actions = [];
5 assert_params =  $\emptyset$ ;
6 foreach (m in  $m_0, \dots, m_n$ ) do
7   r = translate_to_alloy(m);
8   params = get_params(r);
9   a = gen_dynalloy_action(m, params, get_pre(r),
   get_post(r));
10  actions = actions + [a];
11  assert_params = assert_params  $\cup$  params;
12 end
13 curr_unrolls=1;
14 while (!timeout() && curr_unrolls<=unrolls) do
15   dynalloy_program = gen_dynalloy_prog(assert_params,
   actions,  $m_i$ , initial_state, abs_alloy, inv_alloy,
   curr_unrolls);
16   alloy_spec = run_dynalloy_translator(dynalloy_program);
17   <res,model> = run_alloy_analyzer(alloy_spec);
18   if (res == SAT) then
19     wk = translate_to_java(model, C, actions);
20     return wk
21   end
22   else
23     curr_unrolls++;
24   end
25 end
26 return UNSAT;

```

Algoritmo 1: Cómputo de *workarounds transitorios*

de ellos en predicados Alloy (`translate_to_alloy` línea 7), luego se extraen sus parámetros (`get_params` línea 8) y se genera la acción atómica correspondiente (`gen_dynalloy_action` línea 9). Las acciones atómicas junto a sus parámetros son almacenadas (líneas 10 y 11) para luego generar la aserción `wag_finder`.

El ciclo `while` comprendido entre las líneas 14 y 25 ejecuta el procedimiento de búsqueda de *workarounds transitorios* propiamente dicho de manera incremental. Inicialmente se buscarán secuencias de acciones de longitud 1 que imiten el comportamiento de m_i . Para ello se inicializa la variable `curr_unrolls` en 1 (línea 13). Si no se logra encontrar un *workaround* de este tamaño, se va incrementando en 1 el tamaño de la secuencia mediante la sentencia `curr_unrolls++` (línea 23). Dentro del ciclo `while` en la línea 15 se genera, mediante la función `gen_dynalloy_prog`, la aserción `wac_finder`. Luego en la línea 16 se realiza la traducción del programa DynAlloy a Alloy aplicando `run_dynalloy_translator`.

En la línea 17 sobre el modelo obtenido de la traducción se ejecuta Alloy Analyzer a través de `run_alloy_analyzer`. En caso de encontrarse una instancia del modelo Alloy, ésta será transformada a una secuencia de métodos Java aplicando `translate_to_java` en la línea 19. El ciclo `while` tiene como condición de continuidad el no haber alcanzado el límite máximo de `unrolls` y que se no se haya agotado el tiempo límite `timeout`. Si alguna de estas dos condiciones no se cumple, el resultado será `UNSAT`, es decir no se habrá podido encontrar una solución alternativa para el método m_i .

La condición transitoria de los *workarounds* computados por esta técnica es una de sus principales ventajas. La técnica brinda la posibilidad de recuperar al sistema en aquellos casos donde no existen alternativas permanentes. Esta no solo aprovecha la redundancia intrínseca del módulo, sino que también utiliza la información proveniente del estado de invocación del método defectuoso. La aplicación de SAT solving como método de búsqueda implica un uso eficiente de los recursos computacionales.

De los resultados obtenidos en la evaluación experimental (capítulo 5) y de lo analizado en otros trabajos [4, 17, 45] que aplican el concepto de reparación a través de *workarounds*, se pudo observar que en la gran mayoría de los casos las alternativas de recuperación encontradas tienen como máximo 3 o 4 rutinas. Esto es muy favorable al mecanismo de cómputo propuesto, ya que a medida que se incrementan los *loop unrolls* las fórmulas crecen exponencialmente en tamaño necesitando más tiempo para su análisis, y más aún si se plantea que la técnica permitirá dar soluciones a fallas en tiempo

de ejecución.

A fin de mejorar el proceso de búsqueda de *workarounds transitorios*, en la sección 3.6 se plantea una optimización a la técnica utilizando *esquemas de workarounds*.

3.5. Ejemplo completo de búsqueda de workarounds

En esta sección se muestra el procedimiento de cómputo de *workarounds transitorios* propuesto aplicado a un ejemplo concreto. En este caso se computa una alternativa a una invocación fallida del método `set` de la API de `LinkedList`. Recordemos que el método `set(index,elem)` inserta un elemento `elem` en la posición `index` de una `LinkedList`, retornando el valor almacenado previamente en esa posición.

Para este ejemplo asumimos que durante la ejecución de un programa Java que hace uso de listas (de tipo `LinkedList`) en un determinado momento ejecuta la sentencia:

```
Integer v = lis.set(0,4)
```

produciéndose una falla. Esta última es capturada y luego de aplicarse un procedimiento de *rollback* al estado de invocación del método `set`, se pone en marcha el mecanismo de cómputo de *workarounds transitorios*.

La `LinkedList lis` es una lista creada previamente en el programa y posee el estado mostrado en la figura 3.13a. Por otro lado en la figura 3.13b se presenta el estado esperado luego de la ejecución de la sentencia.

En primer lugar el procedimiento construye el predicado `initialState` (figura 3.14) a partir del análisis del estado de invocación del método `set` (figura 3.13a). Luego son generadas las acciones atómicas correspondientes a los métodos de la clase `LinkedList` mediante las cuales se intentará generar un *workaround transitorio*. En las figuras 3.8b y 3.9b se muestran, a modo de ejemplo, dos de las acciones atómicas generadas (acción `clear()` y acción `contains(e)`).

Finalmente a partir de estas acciones se construye el programa de recuperación definido en la aserción `wac_finder` (figura 3.15). La precondition de `wac_finder` exige que se cumplan los predicados `initialState` (para definir el estado de invocación de `set`), `abstraction_sequence` (que establece la

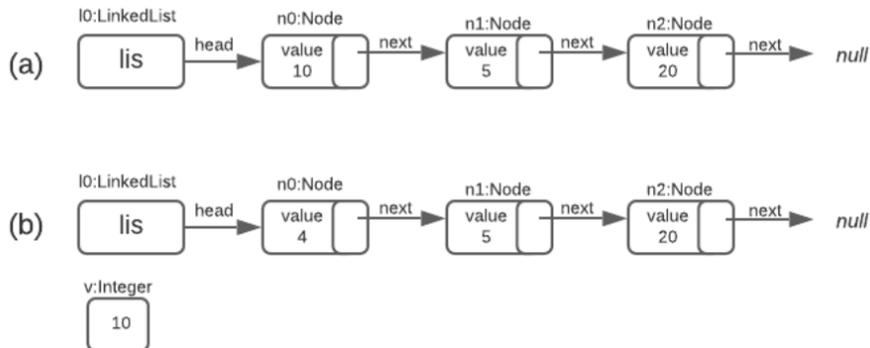


Figura 3.13: (a) Estado de invocación de `set(0,4)` y (b) estado esperado.

```

one sig N0,N1,N2 extends Node
pred initialState[ thiz:LinkedList,
    value:Node -> one (Int),
    head:LinkedList->one(Node+null),
    next: Node->one(Node+null),
    pint0, pint1:Int] {
value = (N0->10)+(N1->5)+(N2->2) and
thiz.head = N0 and
next = (N0->N1)+(N1->N2)+(N2->>null) and++
pint0 = 0
pint1 = 4
}

```

Figura 3.14: Estado de invocación para `lis.set(0,4)`

abstracción de listas a secuencias) y la precondition de `set` (`pre_set` figura 2.16b).

El programa (`program`) definido en `wac_finder` hace uso de las acciones atómicas generadas para la API de `LinkedList`. A fines de simplificar la presentación de este ejemplo se ha incluido un subconjunto de las acciones generadas para la API. Como fue explicado en la sección 3.2.1, cada acción que requiera de parámetros de invocación (enteros o booleanos) es antecedida

por una o más acciones de generación de valores primitivos. Los parámetros de las acciones involucradas quedan renombrados como `i0`, `i1`, ..., `i7` de tipo `Int`.

En la postcondición `post` de `wac_finder` se establece la negación de la postcondición de `set` (`post_set`) junto con el predicado de abstracción (`abstraction_sequence`) y el predicado invariante (`invariant_LinkedList`).

Si durante la verificación de la aserción `wac_finder` se logra construir un contraejemplo, se habrá encontrado una secuencia de acciones que permite llegar al estado deseado 3.13b partiendo de 3.13a y sin ejecutar la sentencia `set`. El contraejemplo es computado como una instancia de una fórmula Alloy que deberá ser traducida a una secuencia concreta de rutinas o métodos de la API `LinkedList` con sus respectivos parámetros. Este procedimiento fue tomado de trabajos anteriores [3, 39].

En el cuadro 3.1 se presentan algunos de los *workarounds transitorios* computados para este ejemplo.

Invocación a recuperar: <code>Integer v=lis.set(0,4)</code>	
Estado inicial: <code>{lis = [10,5,20]}</code>	
Estado final: <code>{v=10 , lis= [4,5,20]}</code>	
unrolls(*)	Workarounds transitorios
1	No existen workarounds de longitud 1.
2	<code>Integer v=lis.removeIndex(0);lis.addIndexElem(0,4);</code>
	<code>Integer v=lis.remove(10);lis.addFirst(4);</code>
	<code>lis.addFirst(4);Integer v=remove();</code>
	...
3	...
4	...
5	<code>lis.clear();lis.addFirst(20);lis.addFirst(5);</code>
	<code>lis.addFirst(4);Integer v=lis.get(0)</code>

Cuadro 3.1: *Workarounds transitorios* para `set(0,4)`.

```

assertCorrectness wac_finder[thiz:LinkedList,
                        abs:LinkedList->(Seq Node),
                        head: LinkedList->one (Node+null),
                        next: Node->one(Node+null),
                        value:Node -> one Int,
                        index,elem,ret_int,pint0,pint1:Int,
                        i0,i1,i2,i3,i4,i5,i6,i7:Int,
                        ret_bool:boolean, ac:ActionExecuted]{
  pre{
    initialState[thiz,value,head,next,index,elem,pint0,pint1]
    and abstraction_sequence [thiz,abs,head,next,value]
    and pre_set[thiz,abs,index]
  }
  program
  {(nonDetInt[i0];add[thiz,abs,i0,ac])+
   (nonDetInt[i1];addFirst[thiz,abs,i1,ac])+
   (nonDetInt[i2];nonDetInt[i3];
    add_index_elem[thiz,abs,i2,i3,ac])+
   (nonDetInt[i4];get[thiz,abs,i4,ret_int,ac])+
   (nonDetInt[i5];remove_index[thiz,abs,i5,ret_int,ac])+
   (nonDetInt[i6];remove_elem[thiz,abs,i6,ret_bool,ac])+
   (clear[thiz,abs,ac])+
   (nonDetInt[i7];contains[thiz,abs,i7,ret_bool,ac])+
   (isEmpty[thiz,abs,ret_bool,ac]))*
  }
  post{
    not(
      post_set[abs,abs',value,value',index,elem,thiz,ret_int'])
    and abstraction_sequence [thiz,abs',head',next',value']
    and invariant_LinkedList[thiz,head',next']
  }
}

```

Figura 3.15: Programa de recuperación para `lis.set(0,4)`

Cualquiera de los *workarounds transitorios* computados podría ser utilizado para recuperar la invocación fallida de `set(0,4)`. El algoritmo de

cómputo siempre opta por una alternativa que involucre la menor cantidad de acciones. Esto sucede por la forma incremental en que está definido el algoritmo de cómputo de *workarounds transitorios*.

3.6. Esquemas de workarounds

Durante la evaluación experimental de la técnica presentada en las secciones anteriores se observó que muchos de los *workarounds transitorios* computados para un método sobre un estado particular pueden ser fácilmente generalizados para recuperar al mismo método en otras situaciones con sólo modificar sus valores de entrada.

Retomaremos el ejemplo de la sección 3.5 a fin de introducir el concepto de *esquema de workaround*. En la tabla 3.1 se muestran los *workarounds transitorios* computados para la ejecución fallida del método `set(0,4)` en el estado de invocación mostrado en la figura 3.13a. El primero de los *workarounds transitorios* incluidos en la tabla 3.1 es:

```
Integer v=lis.removeIndex(0);addIndexElem(0,4)
```

En este caso para imitar el comportamiento de la invocación del método `set(0,4)`, el *workaround* computado utiliza en primer lugar al método `removeIndex(0)` (elimina el elemento que se encuentra en la posición 0 de la lista `lis` y lo almacena en la variable `v`) y luego el método `addIndexElem(0,4)` (inserta el entero 4 en la posición 0 de la lista). Al ejecutar estas dos sentencias se alcanza el estado esperado 3.13b.

Supongamos ahora que el método `set` falla nuevamente al ejecutarse la sentencia:

```
Integer v = lis.set(1,3)
```

en un estado de invocación diferente (figura 3.16a).

Luego de aplicar el procedimiento de búsqueda de *workarounds transitorios* se encuentra la siguiente alternativa de ejecución:

```
Integer v = lis.removeIndex(1);addIndexElem(1,3)
```

En este caso el método `removeIndex(1)` elimina el elemento ubicado en la posición 1 de la lista almacenando el valor removido en la variable `v`. Luego el

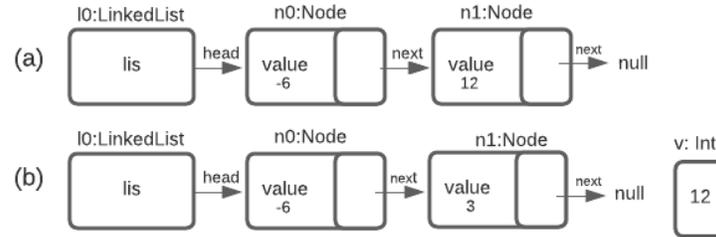


Figura 3.16: (a) Estado de invocación para `set(1,3)` y (b) estado esperado.

método `addIndexElem(1,3)` inserta el entero 3 en la posición 1. Al ejecutar estas sentencias el estado resultante coincide con el estado esperado (figura 3.16b).

En los dos casos anteriores se identifica la misma secuencia de acciones en los *workarounds transitorios* computados pero con distintos parámetros de invocación. Ambos casos son instancias diferentes de lo que hemos definido como un *esquema de workaround*. Un *esquema de workaround* generaliza un *workaround transitorio* fijando la secuencia de métodos y reemplazando los valores concretos en los parámetros por variables. Estos últimos serán instanciados por el SAT-Solver.

En la figura 3.17 presentamos la aserción `instance_schema_set` con el *esquema de workaround*, derivado de los *workarounds transitorios* anteriores, para el método `set(index,elem)`. Podemos observar aquí que el programa `DynAlloy` de búsqueda definido en la aserción, se reduce significativamente en comparación con el de `wac_finder` (figura 3.15). En este caso, las acciones atómicas involucradas son sólo aquellas incluidas en el *workaround transitorio* generalizado (más las acciones de generación de valores primitivos). También se eliminan las composiciones no determinística (+) e iterativa (*) y se fija el orden de las acciones lo que simplifica sustancialmente el análisis a nivel de SAT-Solver. Los parámetros de `instance_schema_set` coinciden con los de `wac_finder` difiriendo sólo en aquellos propios de las acciones. Es decir en `instance_schema_set` sólo se incluyen los parámetros propios de las acciones involucradas en el esquema, en este caso `i0, i1` e `i2` de tipo `Int`.

La aplicación del *esquema de workaround* (figura 3.17) para recuperar al método `set(index,elem)` (en la etapa experimental) permitió encontrar *workarounds* en todos los escenarios en los que fue evaluado. Si bien la aplica-

```

assertCorrectness instance_schema_set [thiz:LinkedList,
    abs:LinkedList->(Seq Node),
    head: LinkedList->one (Node+null),
    next: Node->one(Node+null),
    value:Node -> one Int,
    index,elem,pint0,pint1:Int,
    i0,i1,i2:Int,
    ret_int:Int, ac:ActionExecuted]{
pre{
  initialState [thiz,value,head,next,index,elem,pint0,pint1]
  and abstraction_sequence [thiz,abs,head,next,value]
  and pre_set [thiz,abs,index]
}
program
{
  nonDetInt [i0];
  remove_index [thiz,abs,i0,ret_int,ac];
  nonDetInt [i1]; nonDetInt [i2];
  add_index_elem [thiz,abs,i1,i2,ac])
}
post{
  not(
  post_set [abs,abs',value,value',index,elem,thiz,ret_int'])
  and abstraction_sequence [thiz,abs',head',next',value']
  and invariant_LinkedList [thiz,head',next']
}
}

```

Figura 3.17: Esquema de *workaround* para el método `set(index,elem)`

ción de un *esquema de workaround* no garantiza la recuperación del método que falla, brinda mayores posibilidades de tener éxito.

Nótese también que para algunos métodos se requiere más de un *esquema de workaround*. En estos casos se define un *esquema de workaround* compuesto (figura 3.18). Observemos a continuación el procedimiento de derivación del *esquema de workaround* compuesto para el método `contains(e)`

de `LinkedList`.

$$(esquema_1 + esquema_2 + \dots + esquema_i)$$

Figura 3.18: *Esquema de workaround* compuesto

En primer lugar asumiremos que un programa Java falla al ejecutar la sentencia:

```
boolean r = lis.contains(-7)
```

donde `lis` es una `LinkedList` definida en el programa. El estado de invocación de la sentencia es mostrado en la figura 3.19a. La ejecución de la sentencia (de ser exitosa) debería dejar al sistema en el estado mostrado en la figura 3.19b ya que el entero `-7` se encuentra contenido en `lis`.

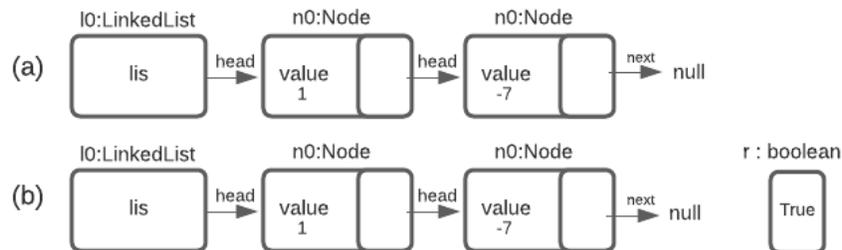


Figura 3.19: (a) Estado de invocación para `boolean r = lis.contains(-7)` y (b) estado esperado.

Luego de aplicar el procedimiento de búsqueda de *workarounds transitorios* se encuentra la siguiente alternativa de ejecución:

```
lis.add(0);boolean r = lis.removeElem(0)
```

En este caso el método `add(0)` inserta en `lis` el entero `0` (que no se encuentra previamente en la lista) y luego el método `removeElem(0)` lo elimina retornando `True` en `r`. Al ejecutar estas sentencias el estado resultante coincide con el estado esperado (figura 3.19b).

En segundo lugar asumiremos que un programa Java falla al ejecutar la sentencia:

```
boolean r = lis.contains(3)
```

El estado de invocación de la sentencia se muestra en la figura 3.20a y la ejecución de la misma (de ser exitosa) debería dejar al sistema en el estado presentado en la figura 3.20b ya que el entero 3 no se encuentra contenido en `lis`.

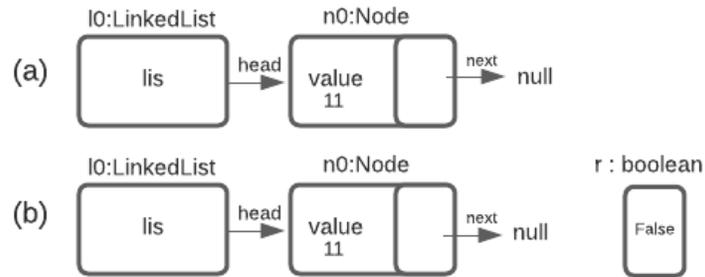


Figura 3.20: (a) Estado de invocación para `boolean r = lis.contains(3)` y (b) estado esperado.

Luego de aplicar el procedimiento de búsqueda de *workarounds transitorios* se encuentra la siguiente alternativa de ejecución:

```
boolean r = lis.isEmpty()
```

En este caso el método `isEmpty()` retornará `False` en `r` lo cual coincide con el estado esperado (figura 3.20b).

Analizando los *workarounds transitorios* computados para `contains(e)` es posible identificar dos *esquemas de workarounds* diferentes. El primer esquema derivado es aplicable cuando el elemento `e` no pertenece a la lista y el segundo para cuando `e` si pertenece a la lista. En la figura 3.21 se presenta la aserción `instance.schema.contains` en donde se define el *esquema de workaround* compuesto para `contains(e)` como la composición no determinística (+) de los dos esquemas derivados. Al igual que en el primer esquema presentado para `set(index,elem)`, se incluyen en el programa sólo las acciones atómicas de los *workarounds transitorios* involucrados (más las acciones de generación de valores primitivos).

```

assertCorrectness instance_schema_contains [thiz:LinkedList,
      abs:LinkedList->(Seq Node),
      head: LinkedList->one (Node+null),
      next: Node->one(Node+null),
      value:Node -> one Int,
      index,elem,ret_int,pint0,pint1:Int,
      i0,i1:Int,
      ret_bool:boolean, ac:ActionExecuted]{
pre{
  initialState [thiz,value,head,next,index,elem,pint0,pint1]
  and abstraction_sequence [thiz,abs,head,next,value]
  and pre_contains [thiz,abs,elem]
}
program
{
  (
    nonDetInt [i0];add [thiz,abs,i0,ac];
    nonDetInt [i1];remove_elem [thiz,abs,i1,ret_bool,ac]
  )
  +
  isEmpty [thiz,abs,ret_bool,ac]

}
post{
  not(
    post_contains [abs,abs',value,value',elem,elem,thiz,ret_bool' ])
  and abstraction_sequence [thiz,abs',head',next',value']
  and invariant_LinkedList [thiz,head',next']
}
}

```

Figura 3.21: Esquema de *workaround* compuesto para el método `contains(e)`

A pesar de que en los *esquemas de workaround* compuestos se incorpora la composición no determinística (+) de los diferentes *esquemas de worka-*

round de un método, la reducción en la cantidad de alternativas es muy importante respecto del programa de cómputo de *workarounds transitorios* (usualmente hay unos pocos esquemas por método, ver sección experimental 5.1.1), y el beneficio obtenido a nivel de análisis de SAT-Solving es sustancial.

```

assertCorrectness instance_schema_isEmpty[thiz:LinkedList,
    abs:LinkedList->(Seq Node),
    head: LinkedList->one (Node+null),
    next: Node->one(Node+null),
    value:Node -> one Int,
    pint0,pint1:Int,
    i0,i1,i2:Int,
    ret_bool:boolean, ac:ActionExecuted]{
pre{
  initialState[thiz,value,head,next,index,elem,pint0,pint1]
  and abstraction_sequence[thiz,abs,head,next,value]
  and pre_isEmpty[thiz,abs]
}
program
{
  (nonDetInt[i0];remove_elem[thiz,abs,i0,ret_bool,ac])
  +
  (nonDetInt[i1];add[thiz,abs,i1,ac];
  nonDetInt[i2];remove_elem[thiz,abs,i2,ret_bool,ac])
}
post{
  not(
  post_isEmpty[abs,abs',value,value',thiz,ret_bool'])
  and abstraction_sequence [thiz,abs',head',next',value']
  and invariant_LinkedList[thiz,head',next']
  )
}
}

```

Figura 3.22: Esquema de *workaround* compuesto para el método `isEmpty()`

En la figura 3.22 se muestra la aserción `instance_schema_isEmpty`, otro *esquema de workaround* compuesto, esta vez correspondiente al método

`isEmpty()`. El esquema derivado posee dos alternativas de ejecución. La primera de ellas es aplicable sobre listas no vacías que invocan a `isEmpty()`. El programa DynAlloy en este caso genera un entero no perteneciente a la lista (`nonDetInt`) e intenta eliminarlo (`remove_elem`) retornando `False`. La segunda alternativa es aplicable en los casos donde la lista que invoca a `isEmpty()` es vacía. El programa DynAlloy derivado en este caso, genera un entero (`nonDetInt`), lo inserta (`add`) en la lista y luego lo elimina (`remove_elem`) retornando `True`.

En el cuadro 3.2 se muestran algunos de los *esquemas de workarounds* computados para `TreeSet`¹ otro de los casos de estudio analizados en este trabajo. Un `TreeSet` es una implementación de la interfaz `Set`. Recordemos que los *workarounds transitorios* son computados para reparar estados específicos. Cada esquema posee un grado diferente de generalidad y podrá aplicarse a un conjunto diferente de escenarios. Estos esquemas no pueden ser considerados como *workarounds permanentes* debido a que dependen de la generación de parámetros específicos. En el cuadro 3.2 se muestra de una manera simplificada para algunos métodos de `TreeSet` un *workaround transitorio* computado, el *esquema de workaround* derivado y la condición bajo la cual será posible su aplicación. Por ejemplo, para recuperar al método `add` sólo podrá aplicarse el *esquema* presentado si se cumple que el elemento a insertar ya se encuentra en el conjunto. Otro ejemplo es mostrado para el método `remove`, el esquema computado podrá aplicarse si el elemento que se desea eliminar es el más pequeño o el más grande del conjunto ya que se hace uso de los métodos `pollFirst()` y `pollLast`.

¹<https://docs.oracle.com/javase/7/docs/api/java/util/TreeSet.html>

Método	Wa. transitorio	Esquema de wa.
add(e)	$\{1,7,10\}.add(10) == \{1,7,10\}.contains(-1)$	$s.add(X) == s.contains(Y)$, si $X \in s$
ceiling(e)	$\{1,7,10\}.ceiling(10) == \{1,7,10\}.floor(10)$	$s.ceiling(X) == s.floor(Y)$
contains(e)	$\{2,4\}.contains(5) == \{2,4\}.isEmpty()$	$s.contains(X) == s.isEmpty()$, si $s \neq \emptyset$ y $X \notin s$
	$\{2,4\}.contains(4) == \{2,4\}.add(5);$ $\{2,4,5\}.remove(5)$	$s.contains(X) == s.add(Y); s.remove(Y)$, si $Y \notin s$ y $X \in s$
first()	$\{2,3,4\}.first() == \{2,3,4\}.floor(2)$	$s.first() == s.floor(X)$
floor(e)	$\{1,7,3,10\}.floor(10) == \{1,7,3,10\}.ceiling(10)$	$s.floor(X) == s.ceiling(Y)$
higher(e)	$\{2,3,5\}.higher(4) == \{2,3,5\}.floor(5)$	$s.higher(X) == s.floor(Y)$
last()	$\{2,3,4\}.last() == \{2,3,4\}.floor(4)$	$s.last() == s.floor(X)$
remove(e)	$\{2,5,9\}.remove(2) == \{2,5,9\}.pollFirst()$	$s.remove(X) == s.pollFirst()$, si X es mín de s .
	$\{2,5,9\}.remove(9) == \{2,5,9\}.pollLast()$	$s.remove(X) == s.pollLast()$, si X es máx. de s .
pollFirst()	$\{2,4,7\}.pollFirst() == \{2,4,7\}.first();$ $\{2,4,7\}.remove(2)$	$s.pollFirst() == s.first(); s.remove(X)$, si X es mín. de s .
pollLast()	$\{2,4,7\}.pollLast() == \{2,4,7\}.last(); \{2,4,7\}.remove(7)$	$s.pollLast() == s.last(); s.remove(X)$, si X es máx. de s .
isEmpty()	$\{2,3,4\}.isEmpty() == \{2,3,4\}.contains(10)$	$s.isEmpty() == s.contains(X)$, si $s \neq \emptyset$ y $X \notin s$
	$\{\}.isEmpty() == \{\}.add(1); \{1\}.remove(1)$	$s.isEmpty() == s.add(X); remove(X)$, si $s = \emptyset$
lower(e)	$\{1,3,24\}.lower(-1) == \{1,3,24\}.floor(-2)$	$s.lower(X) == s.floor(Y)$

Cuadro 3.2: Esquemas de Workarounds para TreeSet

El procedimiento de búsqueda de *workarounds transitorios* se optimiza notablemente con la aplicación de *esquemas de workarounds*. En la sección 3.6.1 se explica cómo se incluyen los *esquemas de workarounds* al proceso de búsqueda global propuesto. Por otro lado, en el capítulo 5 se muestran los resultados obtenidos al aplicar *esquemas de workarounds* mostrando la reducción de los tiempos de recuperación.

A pesar de la naturaleza transitoria de los *esquemas de workarounds*, en algunos casos se tiene la certeza de que un *esquema de workaround* podrá recuperar al método defectuoso. Es decir, siempre podrá encontrarse una valuación apropiada para los parámetros de las acciones atómicas del esquema a fin de recuperar al método defectuoso. Por ejemplo el esquema computado para `set(index, elem)` (figura 3.17) podrá siempre ser utilizado para recuperar las invocaciones fallidas de `set`. Se tiene la seguridad de que el SAT-Solver encontrará el entero adecuado a eliminar (dentro de la secuencia) junto con la posición (`index`) y el nuevo entero (`elem`) a insertar para imitar el comportamiento de `set`. En parte, esto se debe a que la representación utilizada permite soportar cualquier entero representable en 32 bits.

3.6.1. Cómputo de *workarounds transitorios* aplicando *esquemas de workarounds*

El procedimiento de cómputo de *workarounds transitorios* presentado en la sección 3.4 es optimizado incorporando *esquemas de workarounds*.

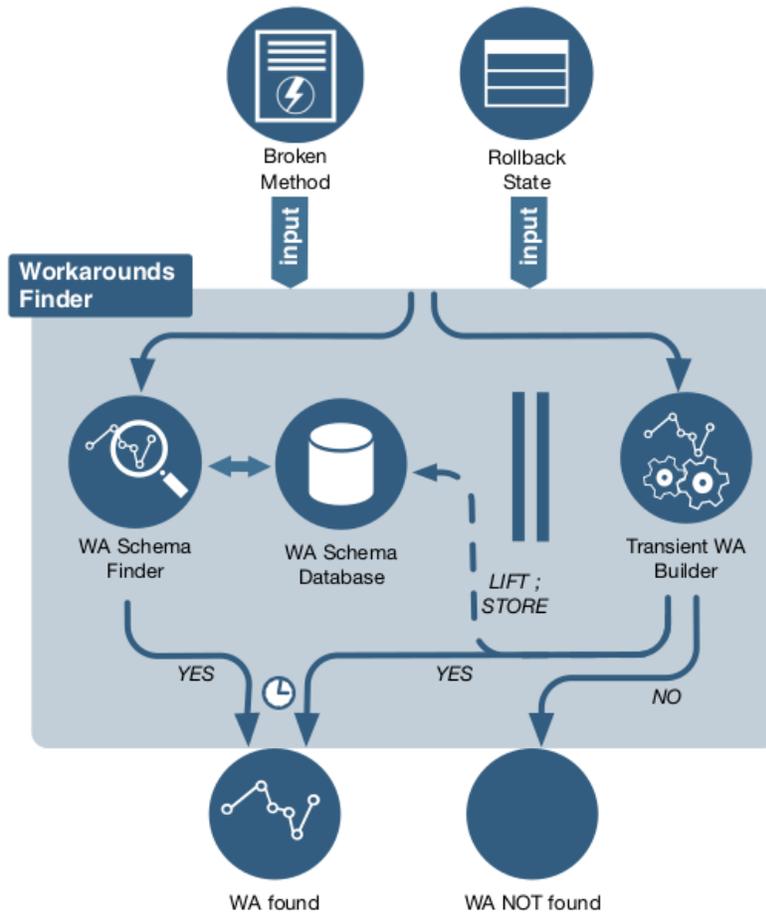


Figura 3.23: Búsqueda de *workarounds transitorios* con *esquemas de workarounds*.

En la figura 3.23 se muestra el procedimiento completo de búsqueda de *workarounds transitorios* incluyendo *esquemas de workaround*.

Dentro del componente *Workarounds Finder* se define un repositorio (*WA Schema Database*) donde son almacenados los *esquemas de workarounds* derivados para cada método. El procedimiento de búsqueda permite ir sumando nuevos esquemas o complementando los ya existentes.

El componente *Workarounds Finder* toma como entrada al método que ha ocasionado la falla (*Broken Method*) junto al estado de invocación (*Roll-back State*) desde donde se ha ejecutado dicho método. Luego se ejecutan dos procesos independientes en paralelo (posibles de ejecutar también de manera distribuida): por un lado *WA Schema Finder* intentará construir un *workaround transitorio* a partir de la instanciación de un *esquema de workaround* almacenado en la *WA Schema Database*. Por otro lado se ejecuta *Transient WA Builder* iniciando el proceso de búsqueda de *workarounds transitorios* descrito en la sección 3.4.

En el caso de lograr instanciar alguno de los *esquemas de workarounds* existentes para el método defectuoso, el proceso completo finaliza retornando el *workaround* encontrado. Si en cambio el segundo proceso finaliza en primer lugar encontrando un *workaround transitorio*, se detiene la ejecución del primer proceso retornando el *workaround transitorio* encontrado. En este último caso, a partir del *workaround* encontrado se deriva un nuevo *esquema de workaround* (luego de aplicar el procedimiento de generalización descrito anteriormente). Este esquema es incorporado a la *WA Schema Database* como un esquema nuevo, si es que no se encuentra ninguno para el método defectuoso, o bien se lo compone no determinísticamente (+) a un esquema ya existente para dicho método (figura 3.18).

Si ninguno de los procesos logra encontrar un *workaround transitorio* o bien se agota el tiempo máximo establecido para la búsqueda, los procesos finalizarán no pudiendo encontrar una alternativa de recuperación.

En los casos de estudio evaluados (sección 5.1.1) se observó que la cantidad de esquemas encontrados para cada método nunca es mayor a tres, por lo que no tuvimos que lidiar con una degradación importante en la performance de nuestro enfoque debido a un crecimiento importante en la cantidad de combinaciones no determinísticas de múltiples esquemas. Si este problema apareciera, se podría imponer una cota en la máxima cantidad de esquemas distintos a almacenar por cada método, para evitar perder los beneficios del enfoque basado en esquemas.

Capítulo 4

Workarounds a través de estados de recuperación

La primera técnica, presentada en el capítulo anterior, propone computar *workarounds transitorios* a través de invocaciones a rutinas de la API del módulo que respeten la especificación del método defectuoso m_i . En este capítulo se presenta una segunda alternativa para recuperar la ejecución fallida de m_i . La propuesta está centrada en aplicar SAT-Solving para construir directamente (en caso de ser posible) un estado de recuperación s_f el cual se pretendía alcanzar si m_i se hubiese ejecutado exitosamente. En la sección 6.4 fueron mencionados algunos trabajos de otros autores [55,92] que han utilizado SAT-Solving para la generación de estados de recuperación de programas. El enfoque presentado aquí es una extensión de estos trabajos.

Nuestra contribución principal reside en la incorporación de predicados de rotura de simetrías [80] y cotas ajustadas [74] a fin de optimizar el cómputo de estados de recuperación. Predicados de rotura de simetrías como así también cotas ajustadas han sido utilizados en diversas áreas de trabajo como por ejemplo verificación acotada de código [38], inferencia de modelos a través del cómputo del MinDFA (mínimo autómata finito determinístico) [93], generación de casos de test [2,79], etc., pero nunca han sido aplicados en la generación de estados de recuperación. La incorporación de estos conceptos mejoran notablemente la eficiencia de la propuesta permitiendo escalar en tamaño y complejidad los estados a generar. Al igual que para la primera técnica, se requiere que el programa Java sobre el cual se aplicará el mecanismo de recuperación incluya especificaciones JML (pre y postcondiciones para los métodos). También se asume que se cuenta con un monitor que ve-

rifica en *runtime* el cumplimiento de dichas especificaciones JML y ante la detección de una violación aplica una acción de *rollback*, dejando al sistema en el estado de invocación del método defectuoso m_i [15]. La recuperación de m_i se realiza a través de un modelo Alloy (generado automáticamente). Este modelo Alloy incluye al predicado `initialState` (sección 3.1) con el que se caracteriza al estado concreto de invocación de m_i . También se requieren de las traducciones Alloy de las especificaciones de m_i , de la función de abstracción y del invariante de representación (definidos en la sección 2.5).

En las secciones siguientes se presenta el modelo Alloy cuyas instancias permitirán construir estados de recuperación para la ejecución fallida de m_i , luego se detalla cómo este modelo es optimizado incorporando predicados de rotura de simetrías y cotas ajustadas.

4.1. Cómputo de estados de recuperación

Con el objetivo de recuperar la ejecución fallida de un método m_i se aplica SAT-Solving para generar un estado de recuperación donde se verifique la postcondición de m_i . Para ello se construye automáticamente un modelo Alloy utilizando las traducciones de las especificaciones JML incluidas en la API Java (sección 2.5). Se incorpora también el predicado `initialState` junto con los predicados de abstracción (`abstraction`) e invariante (`invariant`). Por otro lado, se requiere de los predicados `pre m_i` y `post m_i` obtenidos de la traducción del contrato JML de m_i .

Para computar el estado de recuperación se define el predicado Alloy `recoveryState` (figura 4.1). Este predicado recibe como parámetros a `this` (el objeto desde donde es invocado m_i), \vec{f} y \vec{f}' (campos de los objetos relacionados), \vec{x}_i y \vec{x}'_i (parámetros propios de m_i), $\vec{p}s$ (valores de invocación de m_i), `abs` y `abs'` (resultados de aplicar la función de abstracción, ver abajo) y `ret'` (valor de retorno).

En el cuerpo del predicado `recoveryState`, `initialState` fija el estado inicial de la estructura mediante `this`, \vec{f} y $\vec{p}s$. Luego `abstraction` construye una abstracción del mismo a través de `abs`. Para poder computar un estado de recuperación, dicho estado abstracto debe cumplir con la precondición de m_i . Seguidamente, `recoveryState` busca un estado abstracto `abs'` que satisfaga la postcondición de m_i . Finalmente `abs'` se traduce en una estructura concreta (determinada por `this` y \vec{f}') utilizando `abstraction`. La estructura resultante deberá verificar también el invariante de representación

```
invariant.  
  
pred recoveryState[thiz,  $\vec{f}$ ,  $\vec{f}'$ ,  $\vec{x}_i$ ,  $\vec{x}_i'$ ,  $\vec{p}s$ , abs, abs', ret'] {  
  initState[thiz,  $\vec{f}$ ,  $\vec{p}s$ ]  $\wedge$   
  abstraction[thiz,  $\vec{f}$ , abs]  $\wedge$   
  premi[thiz,  $\vec{f}$ ,  $\vec{x}_i$ , abs]  $\wedge$   
  postmi[thiz,  $\vec{f}$ ,  $\vec{f}'$ ,  $\vec{x}_i$ ,  $\vec{x}_i'$ , abs, abs', ret']  $\wedge$   
  abstraction[thiz,  $\vec{f}'$ , abs']  $\wedge$   
  invariant[thiz,  $\vec{f}'$ ]  
}  
run recoveryState
```

Figura 4.1: Predicado `recoveryState` para el método m_i

La ejecución del comando `run recoveryState` debería (de ser posible) generar una instancia a partir de la cual se construye un estado concreto Java de recuperación, para luego continuar con la ejecución del programa.

4.1.1. Un ejemplo de cómputo de estados de recuperación

Para ilustrar el mecanismo de recuperación retomamos el caso de estudio `LinkedList` donde asumimos una falla en el método `set` al ejecutar la sentencia:

```
int res=lis.set(1,4)
```

en el estado de invocación s_i mostrado en la figura 4.2 y caracterizado por el predicado `initialState` definido en la figura 4.3.

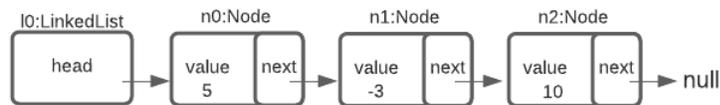


Figura 4.2: Estado de invocación s_i para `set(1,4)`.

```

one sig N0,N1,N2 extends Node
pred initial_state[ thiz:LinkedList,
                    value:Node -> one (Int),
                    head:LinkedList->one(Node+Null),
                    next: Node->one(Node+Null),
                    pint1, pint2:Int] {
  value = (N0->5)+(N1->-3)+(N2->10) and
  thiz.head = N0 and
  next = (N0->N1)+(N1->N2)+(N2->Null) and
  pint1 = 1 and
  pint2 = 4
}

```

Figura 4.3: Estado de invocación para `set(1,4)`

El predicado de recuperación `recoveryState` (figura 4.4) incluye a los predicados `initialState` junto a `pre_set` y `post_set` que establecen el contrato del método `set` (figura 3.6a). También se incluyen a los predicados `abstraction_sequence` (figura 2.15b) e `invariant_LinkedList` (figura 2.14b) con la función de abstracción y el invariante de representación.

En la figura 4.5 se muestra una instancia obtenida al ejecutar el comando

```
run recoveryState for 1 LinkedList, 3 Node
```

el cual determina un estado de recuperación para nuestro ejemplo.

```

pred recoveryState[ thiz:LinkedList,
                    head,head':LinkedList->Node+Null,
                    value,value':Node->Int,
                    next,next':Node->Node+Null,
                    myseq,myseq':LinkedList->seq Node,
                    index,elem:Int,
                    ret':Int ]{

  index=1 and elem=4 and
  initialState[thiz,value,head,next,index,elem] and
  abstraction_sequence[next,head,myseq,thiz,value] and

  pre_set[thiz,myseq,index]
  post_set[myseq,myseq',value,value',index,elem,thiz,ret']

  abstraction_sequence[next',head',myseq',thiz,value'] and
  invariant_LinkedList[next',head',thiz]

}

run recoveryState for 1 LinkedList, 3 Node

```

Figura 4.4: Predicado `recoveryState` de recuperación para `set(1,4)`.

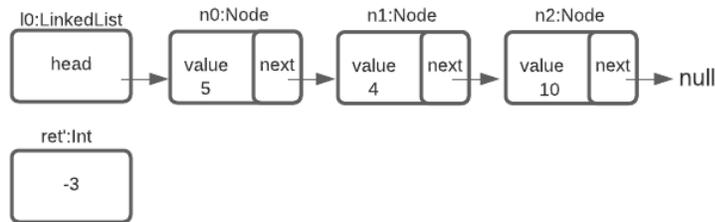


Figura 4.5: Estado s_f para `set(1,4)`.

4.2. Rotura de Simetrías

Una mejora al procedimiento de recuperación presentado en la sección anterior se obtiene mediante la incorporación de predicados de rotura de simetrías. El tamaño y complejidad de las especificaciones pueden determinar el éxito o fracaso del procedimiento de recuperación. Propiedades tales como escalabilidad y rendimiento se vuelven críticas al momento de aplicar análisis de SAT-Solving. Es por esto que se requiere de enfoques eficientes posibles de ser aplicados en tiempo de ejecución.

El procedimiento de generación de estados de recuperación via SAT buscará un modelo que satisfaga el predicado `recoveryState` (figura 4.1) dentro de una gran cantidad de modelos posibles. Los predicados de rotura de simetrías ayudan a reducir esta cantidad eliminando aquellos modelos que se correspondan con estructuras isomorfas. Observemos el ejemplo de `LinkedList` y las posibles estructuras isomorfas que pueden darse al intentar recuperar un método fallido. Recordemos que una `LinkedList` se define mediante las relaciones:

```
thiz:LinkedList
head:LinkedList->Node+Null
next:Node->Node+Null
value:Node->Int
```

Para un scope de exactamente tres nodos, el SAT-Solver podría generar para una misma lista de enteros hasta seis estructuras diferentes, obtenidas de las permutaciones de los identificadores de los nodos de la lista (en general, para n nodos hay $n!$ permutaciones). Notar que en aquellos lenguajes que no permiten al desarrollador manejar la memoria de manera explícita (como Java), los identificadores de los nodos son irrelevantes para los programas, y es posible descartar estas estructuras isomorfas sin afectar la completitud (acotada) del análisis de SAT. En la figura 4.6 se presentan tres `LinkedList` diferentes que representan a la misma lista de enteros.

Para eliminar estas estructuras isomorfas es necesario computar predicados de rotura de simetrías y luego instrumentar las especificaciones Alloy. Estos predicados tienen por finalidad imponer una representación canónica para las estructuras del *heap* eliminando aquellas isomorfas. Para el caso de `LinkedList` un predicado que fuerce a utilizar los nodos en un orden determinado ($N0, N1, N2, \dots$) eliminaría las estructuras isomorfas dejando a una sola

como representante. Por ejemplo de las estructuras mostradas en la figura 4.6 se eliminan todas a excepción de la lista mostrada en la figura 4.6(a) ya que cumple con el orden establecido para los identificadores de los nodos.

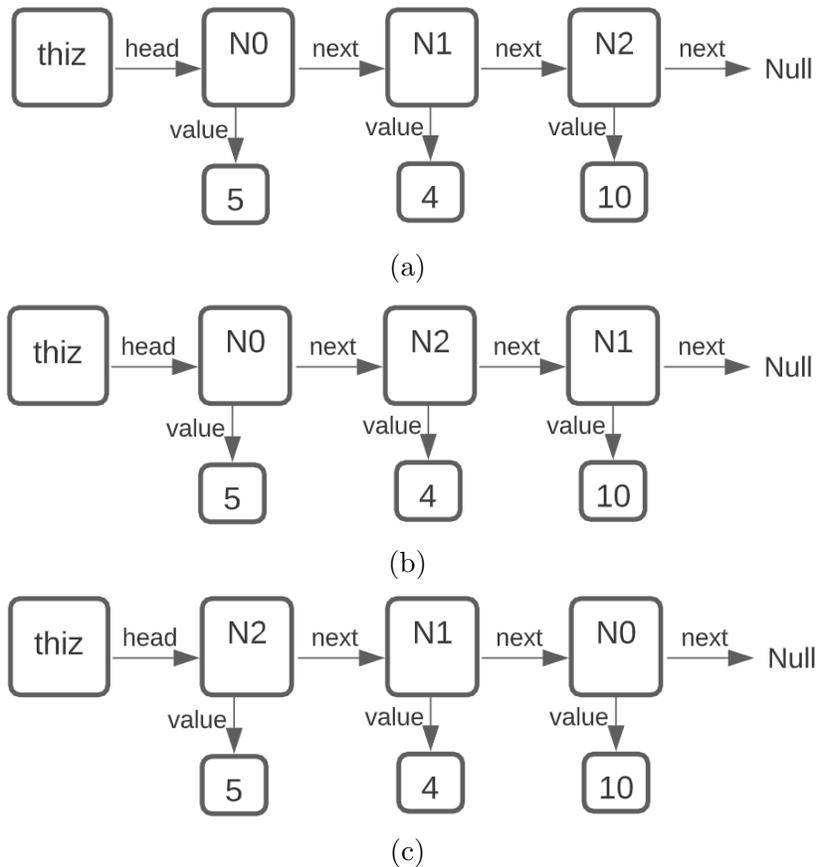


Figura 4.6: Tres listas isomorfas

El procedimiento de cómputo de predicados de rotura de simetrías es tomado de trabajos previos [39, 40] y puede implementarse de manera completamente automática.

A pesar de que Alloy Analyzer posee predicados de rotura de simetrías de propósito general [50], éstos no poseen la capacidad de identificar parte de las estructuras isomorfas que aparecen cuando las especificaciones modelan estructuras de datos complejas que provienen de código fuente. Los resultados experimentales obtenidos al incorporar estos predicados adicionales eviden-

cion que la eliminación de simetrías produce una mejora significativa en la *performance* del análisis de SAT de especificaciones que modelan estructuras de datos complejas (sección 5.3). Por otro lado, además de hacer más eficiente el análisis de SAT-solving, posibilita el cómputo de cotas ajustas más precisas [71, 74].

4.2.1. Cómputo de predicados de rotura de simetrías

El procedimiento utilizado para computar los predicados de rotura de simetrías fue tomado de [39]. Para ilustrar este procedimiento se incluye el caso de estudio `BinTree`. Al final de esta sección también se incluyen los predicados de rotura de simetrías para el caso de estudio `LinkedList`.

En la figura 4.7 se presenta un modelo Alloy para `BinTree` obtenido de la traducción de una implementación Java de árboles binarios.

```

one sig Null {}

sig BinTree {
    root: TNode + Null
}

sig TNode {
    left: TNode + Null,
    right: TNode + Null
}

```

Figura 4.7: Especificación Alloy para `BinTree`

Por simplicidad se omiten del modelo los campos de tipo primitivo, ya que no se rompen simetrías para los mismos. El procedimiento de cómputo de los predicados de rotura de simetrías es dependiente de los *scopes* fijados al momento de ejecutar los comandos *run* o *check* en Alloy Analyzer. Para el ejemplo de `BinTree` se establecen los siguientes *scopes*:

```

exactly 1 BinTree, 5 TNode

```

El modelo de *memory heap* utilizado es definido como un grafo

$$\langle N, E, L, R \rangle$$

donde N , (conjunto de nodos del *heap*) está formado por elementos pertenecientes a las firmas involucradas en el modelo. El conjunto de arcos E está formado por pares de nodos $\langle n_1, n_2 \rangle \in N \times N$. Los arcos corresponden a valores de los campos de la clase, y L es una función que asigna a cada arco el nombre de un atributo. Un arco entre dos nodos $\langle n_1, n_2 \rangle$ etiquetado con f_i , se interpreta como $n_1.f_i = n_2$. Finalmente, $R \subseteq N$ es el conjunto de nodos raíz, los cuales pueden ser argumentos de métodos, atributos estáticos o de tipo `Object`.

Luego de definir los *scopes*, el modelo Alloy es instrumentado con k constantes $T_0 \dots T_{k-1}$ de tipo T , siendo k el *scope* de la firma T . Para los *scopes* definidos en `BinTree`, el modelo Alloy es instrumentado con las siguientes firmas:

```
one sig BinTree0 extends BinTree
one sig TNode0, TNode1, TNode2, TNode3, TNode4 extends TNode
```

El procedimiento de instrumentación requiere de la definición de las funciones auxiliares `nextT`, `minT` y `prevsT` para cada tipo T en el modelo. La función `nextT` establece un orden lineal entre los elementos del mismo tipo T . La función `minT` retorna el menor objeto de un subconjunto de entrada de acuerdo al orden establecido por `nextT`. Finalmente la función `prevsT` retorna el conjunto de nodos menores a un nodo pasado como parámetro. En la figura 4.8 se muestra la definición de dichas funciones auxiliares para el tipo `TNode`.

A continuación cada campo recursivo $r:S \rightarrow (S+Null)$ del modelo es reemplazado por dos funciones parciales:

$$fr:S \rightarrow lone(S+Null) \text{ y } br:S \rightarrow lone(S+Null)$$

Un campo es considerado recursivo si su dominio y codominio (excluyendo al valor `Null`) coinciden.

La función `fr`(*forward*) mapea un objeto en otro estrictamente mayor o nulo y `br`(*backward*) mapea un objeto en otro menor o igual, de acuerdo al orden establecido por `nextT`.

Para el ejemplo de `BinTree` los campos resultantes se muestran en la figura 4.9.

```

fun nextTNode[]: TNode -> lone TNode {
    TNode0->TNode1 + TNode1->TNode2 +
    TNode2->TNode3 + TNode3->TNode4
}

fun minTNode[ns: set TNode]: lone TNode {
    ns - ns.^(nextTNode[])
}

fun prevsTNode[n : TNode]: set TNode {
    n.^(~nextTNode[])
}

```

Figura 4.8: Funciones auxiliares `nextTNode`, `minTNode` y `prevsTNode`

```

root: BinTree -> one(TNode+Null)
fleft: TNode -> lone(TNode+Null)
bleft: TNode -> lone(TNode+Null)
fright: TNode -> lone(TNode+Null)
bright: TNode -> lone(TNode+Null)

```

Figura 4.9: Campos generados para `BinTree`

Los campos *forward* y *backward* deben establecer particiones de sus campos originales por lo que se requiere agregar a la especificación las restricciones mostradas en la figura 4.10.

En la especificación de `BinTree` las definiciones de `left` y `right` son reemplazadas por `fleft + bleft` y `fright + bright`, respectivamente.

En nuestros modelos Alloy es importante hacer referencia al conjunto de nodos alcanzables desde un nodo raíz. Por este motivo se define una función auxiliar `FReach` de alcanzabilidad. Para el caso de `BinTree` la función `FReach` queda definida en la figura 4.11.

```

fact {
    // Los dominios de fleft y bleft son disjuntos
    // la unión es igual a TNode
    no ((fleft.univ) & (bleft.univ)) &&
    TNode = fleft.univ + bleft.univ &&

    // Los dominios de fright y bright son disjuntos
    // la unión es igual a TNode
    no ((fright.univ) & (bright.univ)) &&
    TNode = fright.univ + bright.univ
}

```

Figura 4.10: Restricciones a campos generados para BinTree

```

fun FReach[this:BinTree]: set TNode {
    this.*(root + fleft + fright) - Null
}

```

Figura 4.11: Función FReach para BinTree

El enfoque presentado en [39] propone establecer un orden global entre los nodos del *heap* observando la relación con sus padres. Recordemos que el *heap* es representado como un grafo. Para determinar el orden de un nodo, sólo se deberá tener en cuenta el mínimo padre dentro de un orden global de identificadores. Para ello se definen las funciones `globalMin` y `parentMin`. La función `globalMin` impone este orden, y así permite seleccionar el mínimo objeto de un conjunto de dado de objetos. Para el ejemplo de `BinTree` `globalMin` define que `BinTree0` es anterior a todos los nodos de tipo `TNode`, y que para estos se respeta el orden fijado por `nextTNode`. La función `parentMin` retorna el mínimo padre de un nodo, obteniendo todos sus padres a través de sus campos, y luego aplicando `globalMin`. En la figura 4.12 se presentan estas funciones para el caso de estudio.

El predicado `parentMin` retorna el mínimo padre de un nodo. El orden global entre dos identificadores `n1` y `n2` pertenecientes a una signatura `T` queda definido al comparar sus padres a través de la relación `parentMin`. Dados dos nodos de tipo `T` se consideran los casos mostrados en la figura

```

fun globalMin[s: set Object]: Object {
  s - s.^(BinTree0 -> TNode0 + TNode0 -> TNode1 +
    TNode1 -> TNode2 + TNode2 -> TNode3 + TNode3 -> TNode4)
}

fun parentMin[o: Object]: Object {
  globalMin[(root + fleft + fright).o]
}

```

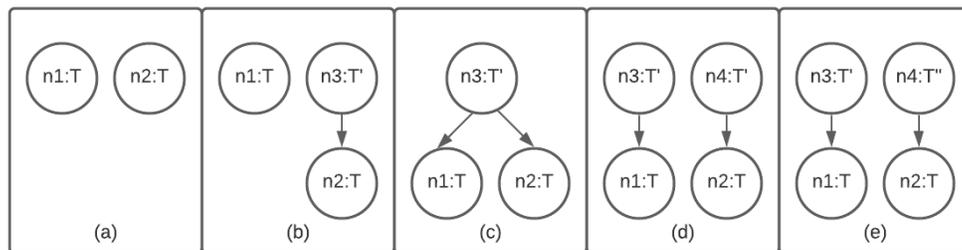
Figura 4.12: Funciones `globalMin` y `parentMin` para `BinTree`

Figura 4.13: Comparación de nodos a través de sus padres mínimos

4.13. El orden queda establecido por las siguientes condiciones:

- Condición 1 (figura 4.13a): si `n1` y `n2` son nodos raíces (asociados a parámetros del método bajo análisis), entonces `n1` recibirá un identificador menor al identificador de `n2` en el caso de que el parámetro al que está asociado `n1` aparezca antes que el asociado a `n2` en la definición del método a analizar.
- Condición 2 (figura 4.13b): si `n1` es un nodo raíz y `n2` no lo es, `n1` deberá tener un identificador menor al identificador de `n2`.

Para los *scopes* definidos en el ejemplo, el único identificador perteneciente a un nodo raíz es `BinTree0` de tipo `BinTree` y está definido como un *singleton*. Las condiciones 1 y 2 no generan ninguna fórmula para `BinTree`.

- Condición 3 (figura 4.13c): si $n1$ y $n2$ no son nodos raíces y además $\text{parentMin}[n1]=\text{parentMin}[n2]=n$, existen los campos $f1$ y $f2$ tal que $n.f1 = n1$ y $n.f2 = n2$. El orden quedará definido de acuerdo al orden en que los campos $f1$ y $f2$ hayan sido definidos en la clase Java de la que provienen.

Si dos nodos distintos $n1$ y $n2$ comparten el mismo “mínimo padre” n a través de los campos `left` y `right`, respectivamente, y `left` se define en el código Java antes que `right`, entonces el identificador de $n2$ deberá ser el siguiente al identificador $n1$ de acuerdo al orden establecido por `nextT`. En el caso de `BinTree` la condición nro. 3 genera la restricción mostrada en la figura 4.14.

```
fact condicionDeOrdenTNode3[] {
  all disj n1, n2: TNode | (n1+n2 in FReach[BinTree0] =>
    ((some n: TNode | n = parentMin[n1] &&
      n = parentMin[n2] &&
      n1 = n.fleft && n2 = n.fright) =>
      n2 = n1.nextTNode[]))
}
```

Figura 4.14: fact condicionDeOrdenTNode3

- Condición 4 (figura 4.13d): si $n1$ y $n2$ no son nodos raíces, siendo $\text{parentMin}[n1]=n3$ y $\text{parentMin}[n2]=n4$ pertenecientes a la misma signatura T , y $n3 \neq n4$, entonces los identificadores $n1$ y $n2$ quedan ordenados de acuerdo al orden de $n1$ y $n2$ en T .

Para el caso de `BinTree`, dados dos nodos $n1$ y $n2$ distintos de tipo `TNode` cuyos padres son $n3$ y $n4$ ($n3 \neq n4$) respectivamente de tipo `TNode`, y el identificador de $n3$ es menor que el de $n4$ ($n3 \in \text{prevsNode}[n4]$), entonces el identificador de $n1$ deberá ser menor al identificador de $n2$ ($n1 \in \text{prevsNode}[n2]$). La condición nro. 4 genera el fact `condicionDeOrdenTNode4` (figura 4.15).

- Condición 5 (figura 4.13e): si $n1$ y $n2$ no son nodos raíces, siendo $\text{parentMin}[n1]=n3 \in T'$ y $\text{parentMin}[n2]=n4 \in T''$, siendo $T' \neq$

```

fact condicionDeOrdenTNoder4[] {
  all disj n1, n2: TNode | (n1+n2 in FReach[BinTree0] =>
    ((some disj n3, n4: TNode |
      n3 = parentMin[n1] && n4 = parentMin[n2] &&
      n3 in prevsNode[n4]) =>
      n1 in prevsTNode[n2]
    )
  )
}

```

Figura 4.15: fact condicionDeOrdenTNoder4

T'', entonces el orden de los identificadores $n1$ y $n2$ queda definido en función del orden que hayan sido definidas las clases de las cuales derivan las firmas T' y T'' .

Esta última condición para el caso de `BinTree` establece el orden entre dos elementos $n1$ y $n2$ de tipo `TNode`, donde $n1$ tiene como padre a un objeto de tipo `BinTree` y $n2$ a un objeto de tipo `TNode` (figura 4.16). Quedará definido entonces el identificador de $n1$ anterior al de $n2$ debido a que la clase `BinTree` es definida antes que la clase `TNode` en el código fuente.

```

fact condicionDeOrdenNoder5[] {
  all disj n1, n2: TNode | (n1+n2 in FReach[BenTree0] =>
    ((some a: BinTree, b: TNode | a = minP[n1] &&
      b = minP[n2]) => n1 in prevsTNode[n2]))
}

```

Figura 4.16: fact condicionDeOrdenNoder5

Por último se instrumenta a las especificaciones con un `fact` que evita una numeración discontinua de los identificadores. Por ejemplo, para el caso de `TNode` si un identificador Ni es utilizado, también deberían haber sido utilizados todos los anteriores. No podrá darse el caso, por ejemplo, de tener un `BinTree` con nodos $N0$, $N2$ y $N3$ sin tener a $N1$. Esta condición queda expresada por el `fact` de la figura 4.17:

```

fact withoutHoles {
  all n: TNode | n in FReach[BinTree0] =>
    prevsTNode[n] in FReach[BinTree0]
}

```

Figura 4.17: fact withoutHoles

Predicados de rotura simetría para LinkedList

Para el caso de estudio `LinkedList` se muestra a continuación el resultado del cómputo de los predicados de rotura de simetría siguiendo el procedimiento descrito en [39]. Como se ha presentado anteriormente, los campos `head: LinkedList -> Node+Null` y `next: Node-> Node+Null` de la signatura `LinkedList` son definidos mediante relaciones que luego serán incorporadas a los predicados como parámetros.

En este caso se fijan los siguientes *scopes* para los comandos `run` y `check`:

```

exactly 1 LinkedList, 3 Node

```

Luego y de acuerdo al procedimiento descrito en la sección anterior, el modelo Alloy es instrumentado con las siguientes signaturas:

```

one sig LinkedList0 extends LinkedList{}
one sig Node0, Node1, Node2 extends Node{}

```

Las funciones auxiliares `nextNode`, `minNode` y `prevsNode` generadas para este caso de estudio son presentadas en la figura 4.18.

```

fun nextNode[]: Node -> lone Node {
    Node0->Node1 + Node1->Node2
}

fun minNode[ns:set Node]: lone Node {
    ns - ns.^(nextNode[])
}

fun prevsNode[n:Node]: set Node {
    n.^(~nextNode[])
}

```

Figura 4.18: Funciones auxiliares `nextNode`, `minNode` y `prevsNode`

El único campo recursivo no me gusta la denominación de recursivo. Me vas a tener que convencer cuando nos reunamos de `LinkedList` es `next`. Este campo es reemplazado por las relaciones:

```

fnext: Node -> lone(Node+Null)
bnext: Node -> lone(Node+Null)

```

Luego se agrega al modelo:

```

fact {
    no ((fnext.univ) & (bnext.univ)) &&
    Node = fnext.univ + bnext.univ
}

```

La función auxiliar `FReach` de alcanzabilidad queda definida en la figura 4.19.

El procedimiento requiere de la definición de las funciones `globalMin` y `parentMin` (figura 4.20).

```

fun FReach[this:LinkedList]: set Node {
  this.*(head + fnext) - Null
}

```

Figura 4.19: Función Freach - LinkedList

```

fun globalMin[s: set Object]: Object {
  s - s.^(LinkedList0 -> Node0 + Node0 -> Node1 +
  Node1 -> Node2)
}

```

```

fun parentMin[o: Object]: Object {
  globalMin[(head + fnext).o]
}

```

Figura 4.20: Funciones globalMin y parentMin - LinkedList

El orden global lineal entre los elementos del *heap* es definido observando la relación con sus padres. En este caso, y fijados los *scopes* la condición nro. 1 no genera ninguna fórmula dentro del modelo ya que el único nodo raíz posible es `LinkedList0`. Del mismo modo las instancias del modelo sólo tendrán exactamente un objeto de tipo `LinkedList` por lo que la condición nro. 2 tampoco genera fórmula alguna.

Por otro lado, la condición nro. 3 define el orden entre objetos que coinciden en el mismo padre mínimo. Esta situación tampoco se presenta `LinkedList` debido a que cada objeto puede tener a lo sumo un padre.

La condición nro. 4 define el orden entre dos objetos de tipo `Node` cuyos padres son de tipo `Node` (figura 4.21).

La condición nro. 5 establece el orden entre dos nodos del mismo tipo que poseen padres mínimos de diferente tipo, en este caso el padre de un nodo es de tipo `LinkedList` y otro es de tipo `Node`. La fórmula resultante es mostrada en la figura 4.22.

Por último se instrumenta el modelo con el `fact withoutHolesNode` (figura 4.23) para evitar una numeración discontinua en los identificadores:

```

fact condicionDeOrdenNode4[] {
  all disj n1, n2: Node | (n1+n2 in FReach[LinkedList0] =>
    ((some disj n3, n4: Node |
      n3 = parentMin[n1] && n4 = parentMin[n2]
        && n3 in prevsNode[n4])
      => n1 in prevsNode[n2]))
}

```

Figura 4.21: fact condicionDeOrdenNoder4 - LinkedList

```

fact condicionDeOrdenNoder5[] {
  all disj n1, n2: Node | (n1+n2 in FReach[LinkedList0] =>
    ((some a: LinkedList, n: Node | a = minP[n1] &&
      b = minP[n2]) => n1 in prevsNode[n2]))
}

```

Figura 4.22: fact condicionDeOrdenNoder5 - LinkedList

4.3. Cotas ajustadas

La segunda optimización para computar estados de recuperación ante la ejecución fallida de un método m_i consiste en la incorporación de cotas ajustadas [71, 74].

Las cotas ajustadas pueden computarse previamente al análisis de SAT, en base al invariante de representación, los predicados de rotura de simetrías y los scopes. Una vez computadas, las cotas ajustadas pueden usarse para eliminar variables de la fórmula proposicional que codifica el problema de búsqueda de un estado de recuperación. Las variables proposicionales a eliminar son aquellas que se corresponden con valores que no pertenecen a las cotas ajustadas.

Las cotas ajustadas permiten reducir la cantidad de variables proposicionales en la fórmula que codifica el estado de invocación del método fallido, lo que produce un impacto positivo al análisis de SAT.

Las fórmulas que codifican los estados (acotados) de un programa deben satisfacer un invariante de representación junto con las restricciones impuestas por los predicados de rotura de simetrías, esto implica que muchas de

```

fact withoutHolesNode {
  all n:Node | n in FReach[LinkedList0] =>
    prevsNode[n] in FReach[LinkedList0]
}

```

Figura 4.23: fact withoutHolesNode- LinkedList

las variables proposicionales que forman parte de las fórmulas nunca sean evaluadas a verdadero.

Las cotas ajustadas definen los conjuntos de asignaciones válidas de valores a campos de la estructura dentro de las restricciones impuestas por el invariante de representación, los predicados de rotura de simetrías y los scopes. En base a estas restricciones es posible computar las cotas ajustadas previamente a la búsqueda de un estado de recuperación. Incluso, las cotas ajustadas pueden precomputarse y almacenarse en un repositorio antes de ejecutar el software bajo análisis.

Una vez que se cuenta con cotas ajustadas, estas pueden usarse para eliminar variables de la fórmula proposicional que codifica el problema de búsqueda de un estado de recuperación. Las variables proposicionales a eliminar serán aquellas que se corresponden con valores que no pertenezcan a las cotas ajustadas. Por ejemplo, la condición de aciclicidad para `LinkedList` imposibilita que las variables proposicionales utilizadas para relacionar a un nodo consigo mismo sean evaluadas a verdadero, por lo que pueden ser eliminadas.

Para ilustrar el concepto de cotas ajustadas se retoma el ejemplo de `LinkedList`. La especificación Alloy de `LinkedList` define a una lista de enteros mediante las relaciones:

```

thiz:LinkedList
head:LinkedList->Node+Null
next:Node->Node+Null
value:Node->Int

```

Ante la detección de una falla en un método `mi` y luego de aplicar una acción de *rollback*, la técnica de recuperación genera el predicado `recoveryState` (figura 4.1). En el caso de `LinkedList` cada uno de estos modelos es definido a través de las relaciones `thiz`, `head`, `next` y `value`. `LinkedList` también

cuenta con un invariante de representación que establece que las listas no pueden tener ciclos (figura 4.24).

```
fact acyclicLinkedList {
    no n:Node | n in n.^next
}
```

Figura 4.24: Invariante de aciclicidad en `LinkedList`

Como ya se ha explicado en la sección de los preliminares el problema de SAT-Solving se reduce a encontrar una valuación que satisfaga una fórmula proposicional que caracteriza, en nuestro ejemplo, todas las posibles listas dentro de los *scopes* establecidos que satisfacen las restricciones dadas (invariante de representación). El número de variables proposicionales utilizadas dependerá de los *scopes* fijados al momento de ejecutar los comandos `check` o `run` (en el caso de Alloy Analyzer).

Las cotas ajustadas permiten disminuir significativamente la cantidad de variables proposicionales utilizadas de una fórmula. Las cotas permiten eliminar aquellas variables que de acuerdo a las restricciones establecidas por el invariante de representación, nunca podrían ser evaluadas a verdadero. En el caso de `LinkedList` por ejemplo, el invariante no permite listas con ciclos, esto implica que las variables proposicionales utilizadas para expresar la relación de un nodo N_i consigo mismo a través de `next` podrían ser descartadas.

Como fue mencionado en la sección 2.2 el lenguaje Alloy posee una semántica relacional, en consecuencia, los campos pertenecientes a las especificaciones son traducidos a relaciones. Dada una especificación Alloy con las firmas R y S , y donde se establecen *scopes* i y j para R y S respectivamente, se definen las constantes:

$$\begin{aligned} r_0, \dots, r_{i-1} &\in R \\ s_0, \dots, s_{j-1} &\in S. \end{aligned}$$

para identificar a los átomos pertenecientes a dichas firmas.

Dado un campo $c:R \rightarrow \text{one}(S + \text{null})$ este es representado con la matriz M_c de j filas por $i+1$ columnas [84] (cuadro 4.1).

La matriz M_c está compuesta de variables proposicionales, donde cada variable $p_{m,n}$ $0 \leq m < j$; $0 \leq n \leq i$ indica la presencia (variable en verdadero) o

M_c	r_0	r_1	..	r_{i-1}	null
s_0	$p_{0,0}$	$p_{0,1}$...	$p_{0,i-1}$	$p_{0,i}$
s_1	$p_{1,0}$	$p_{1,1}$...	$p_{1,i-1}$	$p_{1,i}$
...
s_{j-1}	$p_{j-1,0}$	$p_{j-1,1}$...	$p_{j-1,i-1}$	$p_{j-1,i}$

Cuadro 4.1: Matriz de representación interna para el campo c .

ausencia (variable en False) de un componente en la relación asociada al campo c . Es decir si $p_{1,1}=\text{verdadero}$ significa que el par (s_1, r_1) pertenece a la relación c . Es simple de percibir que a medida que los *scopes* se incrementan, la cantidad de variables proposicionales necesarias crece cuadráticamente (la cantidad de variables en la matriz es $\#R \times \#S+1$).

En el ejemplo de `LinkedList` un scope de 3 para la signature `Node`

```
run recoveryState for 1 LinkedList, 3 Node
```

determina la matriz de representación mostrada en el cuadro 4.2 para la relación `next`. Del mismo modo se generan las variables proposicionales para las relaciones `this`, `next` y `value`.

next	N0	N1	N2	null
N0	$next_{0,0}$	$next_{0,1}$	$next_{0,2}$	$next_{0,3}$
N1	$next_{1,0}$	$next_{1,1}$	$next_{1,2}$	$next_{1,3}$
N2	$next_{2,0}$	$next_{2,1}$	$next_{2,2}$	$next_{2,3}$

Cuadro 4.2: Matriz de representación interna para `next`

Bajo el supuesto de la ocurrencia de una falla y ante la necesidad de computar un estado de recuperación para un método de `LinkedList`, existen pares de nodos que no pueden pertenecer a `next` en un estado inicial del método defectuoso. El invariante de aciclicidad para los *scopes* fijados en este ejemplo establece que los pares $(N0,N0)$, $(N1,N1)$ y $(N2,N2)$ nunca van a ocurrir en `next`. Esto permite eliminar las variables proposicionales $next_{0,0}$, $next_{1,1}$, $next_{2,2}$ (cuadro 4.3) sin comprometer la completitud acotada del análisis de SAT. Por otro lado como ya se mencionó en la sección

anterior, los predicados de rotura de simetrías canonizan las estructuras a generar permitiendo eliminar otras variables proposicionales. Recordemos que estos predicados etiquetan los nodos del *heap* siguiendo un recorrido primero en anchura (Breadth First Search), esto implica la eliminación de las variables proposicionales $next_{1,0}, next_{2,0}, next_{2,1}$ y $next_{0,2}$. En el cuadro 4.3 se muestran las variables proposicionales que definen la cota ajustada para *next* (notar que se excluyen aquellas mencionadas anteriormente).

next	N0	N1	N2	null
N0	$next_{0,0}$	$next_{0,1}$	$next_{0,2}$	$next_{0,null}$
N1	$next_{1,0}$	$next_{1,1}$	$next_{1,2}$	$next_{1,null}$
N2	$next_{2,0}$	$next_{2,1}$	$next_{2,2}$	$next_{2,null}$

Cuadro 4.3: Matriz de representación interna para *next*

Un punto clave a destacar es que el análisis de SAT-Solving mantiene su condición de correctitud y completitud (acotada). El cómputo de cotas ajustadas se realiza para todos los campos pertenecientes a la especificación bajo análisis. En trabajos anteriores se presenta un algoritmo eficiente para el cómputo de cotas ajustadas [71, 74]. Nuestra propuesta incorpora cotas ajustadas en el contexto de cómputo de estados de recuperación, y analiza sus beneficios en este mismo contexto.

El cómputo de cotas ajustadas es costoso en tiempo debido a que se requiere de muchas consultas al SAT-Solver. Este costo adicional no es problemático para el procedimiento de recuperación ya que es realizado en un tiempo anterior. Las cotas son precomputadas en base al invariante de representación y a los diferentes *scopes*. Al momento de ejecutar el cómputo del estado de recuperación para un método fallido, ya se dispondrá de las cotas ajustadas apropiadas.

Para los experimentos realizados en esta tesis se computaron cotas ajustadas para todas las estructuras para las que fue evaluada la segunda técnica. Por ejemplo, para `LinkedList` fueron computadas cotas ajustadas para listas de entre 3 y 25 elementos. El cómputo de las cotas fue realizado aplicando el procedimiento Bottom-up [74] (descrito en la sección 4.3.1). Se fijó un tiempo límite de 1 hora para el cómputo de las cotas para cada caso de estudio y *scope*, y esto fue suficiente para computar cotas para todos los escenarios de recuperación considerados.

4.3.1. Algoritmo BOTTOM-UP para cómputo de cotas ajustadas

En esta sección se describe el algoritmo BOTTOM-UP (algoritmo 2 propuesto en [74]) utilizado para computar cotas ajustadas en los casos de estudio detallados en el capítulo 5. Una forma de computar cotas ajustadas consiste en visitar las instancias válidas del modelo y a partir de ellas extraer los valores para cada uno de los campos, que son los valores que pertenecen a las cotas ajustadas. En general existe una gran cantidad de instancias válidas, inclusive para *scopes* pequeños, es por ello que el enfoque BOTTOM-UP sólo considera las instancias útiles para incorporar nuevos valores en la cota ajustada. Para obtener estas instancias se realizan llamadas a un procedimiento de decisión basado en SAT-Solving.

```

1 function BOTTOM-UP( $f_1, \dots, f_n$ , scopes, I, C)
2    $S_{f_1}, \dots, S_{f_n} = \emptyset, \dots, \emptyset$ ;
3   while True do
4      $\phi = I(f_1, \dots, f_n) \wedge C(f_1, \dots, f_n) \wedge (\exists_{j \in 1, \dots, n} | f_j \notin S_{f_j})$ ;
5     if SAT( $\phi$ , scopes, instance) then
6       | extend( $S_{f_1}, \dots, S_{f_n}$ , instance);
7     end
8     else
9       | break;
10    end
11  end
12  return  $S_{f_1}, \dots, S_{f_n}$ ;
13 end

```

Algoritmo 2: Algoritmo BOTTOM-UP

El Algoritmo 2 presenta un pseudocódigo de BOTTOM-UP. El algoritmo recibe como parámetros de entrada a los campos f_i de la estructura, los *scopes* para cada uno de los tipos involucrados, el invariante de representación *I* (que caracteriza a las instancias válidas del *heap*) y los predicados de rotura de simetrías *C* utilizados para canonizar el *heap* (computados según lo explicado en la sección 4.2.1). Como salida el algoritmo retorna los conjuntos S_{f_i} en donde quedan almacenadas las cotas ajustadas para cada campo f_i .

El procedimiento BOTTOM-UP inicializa en vacío a los conjuntos S_{f_1}, \dots, S_{f_n}

(línea 2). El ciclo comprendido entre las líneas 3 y 11 requiere de un procedimiento de decisión $SAT(\phi, scopes, instance)$ para producir instancias válidas del *heap* que permitan extender los conjuntos S_{f_1}, \dots, S_{f_n} . En la línea 4 se define a ϕ como la conjunción del invariante de representación $I(f_1, \dots, f_n)$, los predicados de rotura de simetría $C(f_1, \dots, f_n)$ y la fórmula $(\exists_{j \in 1, \dots, n} | f_j \notin S_{f_j})$. Esta última exige al SAT-Solver que genere instancias para los campos f_1, \dots, f_n que no estén incluidas previamente en los conjuntos de cotas, asegurando que los conjuntos de cotas sean extendidos con al menos un nuevo valor en cada iteración. El proceso finaliza cuando no existen instancias nuevas para incorporar a los conjuntos S_{f_1}, \dots, S_{f_n} .

A partir de las instancias generadas, se incorporan en los conjuntos S_{f_1}, \dots, S_{f_n} (línea 6) los nuevos valores para cada campo f_i .

Para el caso de estudio de `LinkedList`, el cómputo de cotas ajustadas es realizado para los campos `head` y `next`. Para simplificar la presentación, no se incluye el campo `value` de listas en el cómputo de cotas. En la figura 4.25 se muestra la ejecución del procedimiento de cómputo de cotas para `LinkedList` fijando los siguientes *scopes*: 1 `LinkedList`, 3 `Node`.

Cada fila de la figura 4.25 presenta una nueva instancia de `LinkedList` obtenida de aplicar SAT (algoritmo 2 - línea 5). En las columnas S_{head} y S_{next} se muestran las cotas de los campos `head` y `next` en cada iteración. El proceso finaliza cuando el SAT-Solver no puede generar nuevos valores para incorporar a estos conjuntos.

En primer lugar se definen los conjuntos S_{head} y S_{next} inicializándolos en vacío. Luego, en la primera iteración el SAT-Solver genera una instancia válida de `LinkedList` (una lista vacía en este caso) que permite extender S_{head} con el par `(this, Null)`. En la segunda iteración el SAT-Solver genera una nueva instancia de `LinkedList` (esta vez la lista formada por los nodos `N0` y `N1`) permitiendo extender a S_{head} con el par `(this, N0)` y a S_{next} con los pares `(N0, N1)`, `(N1, Null)`. De esa forma el procedimiento continua iterando y agregando valores a los conjuntos de cotas hasta que en la iteración nro. 5 el SAT-Solver retorna UNSAT no encontrando una instancia para extender las cotas y terminando así su ejecución.

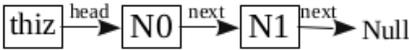
Inicialización	S_{head}	S_{next}
	$\{\}$	$\{\}$
Iteración nro.1 - SAT	S_{head}	S_{next}
	$\{(\mathbf{this}, \mathbf{Null})\}$	$\{\}$
Iteración nro.2 - SAT	S_{head}	S_{next}
	$\{(\mathbf{this}, \mathbf{Null}), (\mathbf{this}, \mathbf{N0})\}$	$\{(\mathbf{N0}, \mathbf{N1}), (\mathbf{N1}, \mathbf{Null})\}$
Iteración nro.3 - SAT	S_{head}	S_{next}
	$\{(\mathbf{this}, \mathbf{Null}), (\mathbf{this}, \mathbf{N0})\}$	$\{(\mathbf{N0}, \mathbf{N1}), (\mathbf{N1}, \mathbf{Null}), (\mathbf{N1}, \mathbf{N2}), (\mathbf{N2}, \mathbf{Null})\}$
Iteración nro.4 - SAT	S_{head}	S_{next}
	$\{(\mathbf{this}, \mathbf{Null}), (\mathbf{this}, \mathbf{N0})\}$	$\{(\mathbf{N0}, \mathbf{N1}), (\mathbf{N1}, \mathbf{Null}), (\mathbf{N1}, \mathbf{N2}), (\mathbf{N2}, \mathbf{Null}), (\mathbf{N0}, \mathbf{Null})\}$
Iteración nro.5 - UNSAT	S_{head}	S_{next}
	$\{(\mathbf{this}, \mathbf{Null}), (\mathbf{this}, \mathbf{N0})\}$	$\{(\mathbf{N0}, \mathbf{N1}), (\mathbf{N1}, \mathbf{Null}), (\mathbf{N1}, \mathbf{N2}), (\mathbf{N2}, \mathbf{Null}), (\mathbf{N0}, \mathbf{Null})\}$

Figura 4.25: Ejecución de Bottom-up para LinkedList

En la sección 5.3 de la evaluación experimental se pueden observar que los predicados de rotura de simetría y las cotas ajustadas ayudan a hacer más eficiente el cómputo de estados de recuperación, y permiten incrementar significativamente la cantidad de casos en los que se puede generar un estado de recuperación en la práctica.

Capítulo 5

Evaluación experimental

A fin de establecer una valoración de la efectividad de las técnicas de cómputo de *workarounds transitorios* se realizaron una serie de experimentos sobre implementaciones de colecciones y `TimeOfDay`¹. Esta última es una implementación estándar que opera con fechas y horas, tomada de la biblioteca ampliamente usada para la manipulación de fechas `JodaTime`. Las técnicas presentadas en esta tesis fueron evaluadas sobre los siguientes casos de estudio:

- `LinkedList`²: una implementación de listas simplemente encadenadas para la interface `java.util.List` de Java.
- `ApacheList`³: una implementación de listas doblemente encadenadas extraída de `Apache.Commons.Collections` para la interface `List`.
- `BinarySearchTree`: una implementación de la interface `java.util.Set`⁴ de Java con árboles binarios de búsqueda extraída de [87].
- `AVLTree`: una implementación de la interface `java.util.Set` basada en árboles balanceados AVL tomada de [8].
- `TreeSet`⁵: una implementación de árboles rojos y negros (balanceados) [7] que implementan la interface `java.util.Set`.

¹<https://www.joda.org/joda-time/apidocs/org/joda/time/TimeOfDay.html>

²<https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html>

³<https://commons.apache.org/proper/commons-jcs/commons-jcs-core/apidocs/org/apache/commons/jcs/struct/DoubleLinkedList.html>

⁴<https://docs.oracle.com/javase/7/docs/api/java/util/Set.html>

⁵<https://docs.oracle.com/javase/7/docs/api/java/util/TreeSet.html>

-
- `TreeMap`⁶: una implementación de la interface `java.util.Map`. Al igual que `TreeSet`, `TreeMap` está basada en árboles rojos y negros. Un `TreeMap` define una colección de pares de elementos (clave, valor), donde no se permiten claves duplicadas.
 - `TimeOfDay`⁷: Esta clase pertenece a la biblioteca `JodaTime`⁸ y permite representar un punto exacto en la línea de tiempo con una precisión de milisegundos. `JodaTime` cuenta con una serie de clases para operar sobre fechas y horas gestionando múltiples sistemas de calendarios. Al mismo tiempo proporciona una API simple con operaciones aritméticas sobre fechas.

Cada caso de estudio fue extendido con su correspondiente invariante de representación JML junto a las precondiciones y postcondiciones de cada uno de sus métodos.

Para la evaluación de cada caso de estudio se generaron escenarios de recuperación aleatoriamente mediante la herramienta *Randoop* [68]. Esta herramienta construye automáticamente tests unitarios seleccionando de forma aleatoria secuencias de métodos de una clase Java. El procedimiento de generación invoca los métodos de las clases bajo test y utiliza los resultados obtenidos de la ejecución de los métodos para crear aserciones que capturen el comportamiento actual del programa. *Randoop* puede utilizarse con dos propósitos: encontrar errores en un programa y crear tests de regresión para alertar sobre posibles errores introducidos por cambios futuros del programa. En este trabajo *Randoop* fue utilizado para generar trazas aleatorias de métodos que al ser ejecutadas generan los objetos que representan el estado de invocación de un método que se asume defectuoso. La cantidad y tamaño de los objetos generados es dependiente del tiempo de ejecución de *Randoop* y la cantidad y complejidad de los métodos contenidos en la clase.

Para la generación de estados de invocación sobre listas se utilizó la clase `LinkedList`, para conjuntos se utilizó `TreeSet` y para Maps, `TreeMap`. Para `TimeOfDay` se utilizó esta misma clase. Luego de ejecutar *Randoop* durante 1 hora para cada caso, se generaron: 141.000 tests para `LinkedList`, 136.000 tests para `TreeSet`, 138.000 tests para `TreeMap` y 48.000 tests para `TimeOfDay`. De cada conjunto de tests fue seleccionada aleatoriamente una

⁶<https://docs.oracle.com/javase/7/docs/api/java/util/TreeMap.html>

⁷<https://www.joda.org/joda-time/apidocs/org/joda/time/TimeOfDay.html>

⁸<https://www.joda.org/joda-time/>

muestra tomando 1 instancia de cada 1000, obteniendo como resultado: 141 `LinkedList`, 136 `TreeSet`, 138 `TreeMap` y 48 `TimeOfDay`. Al ejecutar cada uno de estos test se obtuvo un estado de invocación concreto que luego de ser traducido a su representación abstracta Alloy se transforma en un estado inicial para la interface correspondiente. Este procedimiento fue detallado en la sección 3.1.

Para los demás casos de estudio no fue necesario repetir el procedimiento de generación de estados de invocación, ya que a nivel abstracto coinciden con los ya generados. Por ejemplo, las representaciones abstractas de `ApacheList` y `LinkedList`, que implementan la interface `java.util.List`, son secuencias de enteros.

Del mismo modo ocurre para las diferentes implementaciones de la interface `java.util.Set`: `TreeSet`, `AVLTree` y `BinarySearchTree`, su representación abstracta es definida como un conjunto de números enteros.

Para la clase `TreeMap` (que implementa la interface `java.util.Map`) la representación abstracta fue definida como un conjunto de pares (clave,valor).

Finalmente para `TimeOfDay` la representación abstracta es definida como una tupla de enteros: (horas, minutos, segundos, milisegundos).

Para cada método m_i definido en las clases bajo análisis y para cada uno de los estados iniciales generados, se asumió que m_i no pudo satisfacer su contrato, y a partir de ese momento se intentó computar un *workaround transitorio*. De esta manera, por ejemplo, para el método `removeLast` de `LinkedList` se intentó computar un *workarounds transitorio* en 116 estados de invocación diferentes generados para la interface `java.util.List`.

Para las ejecuciones de los experimentos se estableció un tiempo límite de cómputo de workarounds de 10 minutos.

Todos los experimentos fueron ejecutados en una PC con CPU Intel 3.40 GHz Intel (R) Core (TM) i5-4460, con 8 GB de RAM. Se utilizó GNU/Linux 3.2.0 como sistema operativo. La herramienta implementada para el cómputo de *workarounds transitorios* fue desarrollada como un proyecto Eclipse Java y ejecutada utilizando OpenJDK 1.7 como la plataforma Java subyacente, junto con Python 2.7. Esta herramienta junto con las especificaciones utilizadas para los experimentos pueden ser encontradas en [1].

Cabe aclarar que la implementación actual de la herramienta de cómputo de workarounds permite solamente la inclusión métodos que reciben como parámetros a la estructura de datos corriente `this` junto a parámetros de tipos primitivos `boolean` e `int`. Es por ello que algunos métodos que por ejemplo tomen otras colecciones como parámetros no se incluyen en los ex-

perimentos. Se tiene planificado como trabajo futuro extender la herramienta a fin de incluir aquellos métodos que utilicen tipos no primitivos.

5.1. Evaluación de técnica 1: Cómputo de secuencias de métodos para *workarounds transitorios*

La primera de las técnicas, presentada en el capítulo 3, fue evaluada midiendo su capacidad para construir alternativas de ejecución utilizando la redundancia intrínseca contenida en los módulos de software. Nótese que para la evaluación de técnica 1 se especificaron los métodos definidos en la interface que implementa cada clase, y las especificaciones se escribieron en términos de las estructuras abstractas que definen dichas interfaces. Esto implica que por ejemplo para los casos `BinarySearchTree`, `AVLTree` y `TreeSet` se evalúa el cómputo *workarounds transitorios* sobre las especificaciones de los métodos definidos en la interface `java.util.Set`. De igual manera ocurre para `ApacheList` y `LinkedList`, los *workarounds transitorios* son computados sobre las especificaciones de los métodos definidos en la interface `java.util.List`. Y para `TreeMap` se hace lo propio con los métodos definidos en la interface `java.util.Map`

En esta sección se muestran los resultados experimentales de aplicar técnica 1 sobre los casos de estudio citados al comienzo del capítulo. Los resultados son presentados en tablas donde se registra: nombre del caso de estudio, cantidad de estructuras analizadas, tamaño mínimo, tamaño máximo y tamaño promedio de las estructuras analizadas en el experimento. Estos datos son incluidos en el encabezado de cada tabla con el siguiente formato (ejemplo tabla 5.1): List (núm./ mín./ máx./ prom.) : 141 / 11 / 21 / 13.68, indicando así que fueron evaluadas 141 listas, la lista de menor y mayor longitud fueron de 11 y 21 elementos respectivamente, y la longitud promedio de las listas es de 13.68 elementos.

Por cada método de la clase (**Método**) se reporta la cantidad de *workarounds transitorios* computados (**#wa**), la longitud promedio de los *workarounds transitorios* computados (**Long.prom.wa**), el tamaño máximo de la estructura recuperada (**#nodos máx.**), el tiempo promedio de cómputo (**Tiempo prom.**), el tiempo mínimo y tiempo máximo de cómputo registrado (**Tiempo mín.**, **Tiempo máx.**) y finalmente el tiempo total que demandó el experi-

mento (**Tiempo total**). Los tiempos son informados en formato hh:mm:ss (horas, minutos y segundos).

List

En el cuadro 5.1 se presentan los resultados del cómputo de *workarounds transitorios* aplicando técnica 1 para los métodos pertenecientes a la interface **List**.

List (núm./ mín./ máx./ prom.) : 141 / 11 / 21 / 13.68							
Método	#wa	Long. prom. wa	#nodos máx.	Tiempo prom.	Tiempo mín.	Tiempo máx.	Tiempo total
add	141	1	21	00:00:19	00:00:06	00:00:50	00:44:39
addFirst	141	1	21	00:00:18	00:00:06	00:00:50	00:42:18
addLast	141	1	21	00:00:19	00:00:06	00:00:55	00:44:39
add2	141	1	21	00:00:19	00:00:05	00:00:54	00:44:39
clear	0	-	-	-	-	-	23:30:00
contains	141	1	21	00:00:17	00:00:05	00:00:55	00:39:57
element	141	1	21	00:00:16	00:00:05	00:00:52	00:37:36
get	141	1	21	00:00:19	00:00:05	00:00:53	00:44:39
getFirst	141	1	21	00:00:16	00:00:05	00:00:54	00:37:36
getLast	141	1	21	00:00:16	00:00:06	00:00:51	00:37:36
indexOf	141	1	21	00:00:19	00:00:05	00:00:49	00:44:39
isEmpty	141	1	21	00:00:15	00:00:05	00:00:55	00:35:15
lastIndexOf	141	1	21	00:00:19	00:00:06	00:00:54	00:44:39
offer	141	1	21	00:00:17	00:00:05	00:00:51	00:39:57
offerFirst	141	1	21	00:00:17	00:00:05	00:00:56	00:39:57
offerLast	141	1	21	00:00:17	00:00:06	00:00:56	00:39:57
peek	141	1	21	00:00:17	00:00:06	00:00:55	00:39:57
poll	141	1	21	00:00:16	00:00:06	00:00:56	00:37:36
pollFirst	141	1	21	00:00:16	00:00:05	00:00:53	00:37:36
pollLast	141	1	21	00:00:16	00:00:05	00:00:55	00:37:36
pop	141	1	21	00:00:17	00:00:05	00:00:51	00:39:57
push	141	1	21	00:00:19	00:00:05	00:00:50	00:44:39
remove	141	1	21	00:00:19	00:00:06	00:00:49	00:44:39
removeFirst	141	1	21	00:00:18	00:00:05	00:00:48	00:42:18
removeFO	141	1	21	00:00:19	00:00:05	00:00:54	00:44:39
removeLast	141	1	21	00:00:19	00:00:06	00:00:54	00:44:39
removeIndex	141	1	21	00:00:20	00:00:07	00:00:59	00:47:00
removeElem	141	1	21	00:00:19	00:00:05	00:00:49	00:44:39
setElement	141	2	21	00:00:57	00:00:15	00:03:02	02:13:57

Cuadro 5.1: Caso de estudio: **List** - Técnica 1

Como se puede observar en el cuadro 5.1, el único método para el cual no se pudo computar un *workaround transitorio* ha sido **clear**. Si bien para este método existen *workarounds transitorios*, la imposibilidad de su cómputo está

vinculado al tamaño de las listas analizadas. Por ejemplo para vaciar una lista que contiene 11 elementos, se podría invocar al método `removeLast()` 11 veces. El cómputo de *workarounds* se realiza de manera incremental, inicialmente se busca una combinación de acciones de longitud 1, luego de longitud 2, luego de longitud 3, etc. Esto se logra incrementando la cantidad de *loop unrolls* al momento de verificar el programa DynAlloy de cómputo de *workarounds transitorios*. Este procedimiento ha sido detallado en la sección 3.4. El incremento en la cantidad de *loop unrolls* genera fórmulas de mayor tamaño, esto implica un mayor tiempo de análisis insumido por el SAT-Solver al momento de encontrar una valuación que las satisfaga. En el caso del método `clear`, para lograr computar el *workaround* formado por la secuencia de 11 invocaciones a `removeLast()` (para la lista de menor tamaño incluida en los experimentos) se van a requerir 11 *loop unrolls*. Para esta fórmula el SAT-Solver no logra encontrar una valuación antes de alcanzar el tiempo límite. Lo mismo ocurre con listas mayores a 11 elementos. Para el resto de los métodos analizados se lograron computar *workarounds transitorios* de longitudes menores o iguales a 2.

Con respecto a los tiempos de cómputo podemos observar que de los 29 métodos incluidos en el análisis, para 28 de ellos se logró computar *workarounds* para la totalidad de los estados de invocación en tiempos promedio aceptables, comprendidos entre 16 y 20 segundos, con mínimos comprendidos entre 5 y 7 segundos. Los tiempos obtenidos pueden ser mejorados sustancialmente incorporando *esquemas de workarounds* (sección 5.1.1).

Cabe señalar que la interface `List` posee un alto grado de redundancia intrínseca en sus métodos lo cual es muy beneficioso para técnica 1. Pares de métodos como por ejemplo: `offerFirst(e)` y `addFirst(e)`; `offerLast(e)` y `addLast(e)`; `getFirst()` y `element()`; `removeFirst()` y `remove()`; `getLast()` y `get(i)`; entre otros, tienen el mismo comportamiento o bien éste depende de la elección adecuada de parámetros. Esto posibilita que el procedimiento de cómputo de *workarounds* encuentre rápidamente una alternativa de ejecución.

Set

En el cuadro 5.2 se muestran los resultados experimentales para los métodos de la interface `Set`.

Set (núm./ mín./ máx./ prom.) : 136 / 11 / 22 / 13.16							
Método	#wa	Long. prom. wa	#nodos máx.	Tiempo prom.	Tiempo mín.	Tiempo máx.	Tiempo Total
add	58	2	22	00:01:58	00:00:57	00:06:59	14:54:04
ceiling	136	1	22	00:00:38	00:00:14	00:00:55	02:16:08
clear	-	-	-	-	-	-	22:40:00
contains	136	1	22	00:00:13	00:00:17	00:01:08	01:19:28
first	136	1	22	00:00:37	00:00:26	00:01:03	02:13:52
floor	136	1	22	00:00:34	00:00:14	00:00:52	02:07:04
higher	136	1	22	00:00:40	00:00:15	00:00:59	02:20:40
isEmpty	136	1	22	00:00:39	00:00:15	00:01:09	02:18:24
last	136	1	22	00:00:39	00:00:26	00:01:04	02:18:24
lower	136	1	22	00:00:34	00:00:14	00:00:59	02:07:04
pollFirst	136	1	22	00:00:39	00:00:16	00:03:25	02:18:24
pollLast	136	1	22	00:00:39	00:00:18	00:03:07	02:18:24
remove	136	1	22	00:00:39	00:00:16	00:01:04	02:18:24

Cuadro 5.2: Caso de estudio: Set - Técnica 1

De los métodos evaluados en Set, para el método clear no pudieron computarse workarounds antes de alcanzar el tiempo límite por las mismas razones que se explicaron para el método clear de List.

Por otro lado, para el método add se lograron computar 58 *workarounds transitorios*. Para el resto de los métodos se pudo computar alternativas de ejecución en el 100 % de los casos. Para el método add, de los 136 estados iniciales analizados, sólo para 58 pudieron computarse *workarounds transitorios*. El método add toma como parámetro un elemento e intenta insertarlo en el conjunto corriente. Si tiene éxito retorna verdadero, en caso contrario retorna falso. Analizando detenidamente los estados en donde se ha logrado computar workarounds para add exitosamente encontramos que son aquellos casos en donde el elemento a insertar ya se encontraba previamente en el conjunto, por lo que invocando una acción de consulta que retornara falso, como por ejemplo isEmpty, basta.

Para los casos en donde el elemento a insertar no se encontraba previamente en el conjunto corriente, no se logró computar workarounds. Ésto se debe a que dentro de los métodos incluidos en el análisis, el único que puede agregar elementos a un conjunto es add.

Con respecto a los tiempos de cómputo se puede observar que salvo para el método add, el tiempo promedio obtenido está comprendido entre 13 y 40 segundos. Como en el caso anterior, estos tiempos pueden mejorarse al incorporar *esquemas de workarounds* al procedimiento.

Map

En el cuadro 5.3 se presentan los resultados del cómputo de *workarounds* transitorios para **Map**.

Map (núm / mín / máx / prom) : 138 / 11 / 22 / 13.68							
Método	#wa	Long. prom. wa	#nodos máx.	Tiempo prom.	Tiempo mín.	Tiempo máx.	Tiempo Total
ceilingKey	138	1	21	00:00:12	00:00:08	00:00:29	00:27:30
clear	0	-	21	-	-	-	23:00:00
containsValue	138	1	21	00:00:12	00:00:08	00:00:30	00:27:36
firstEntry	138	1	21	00:00:12	00:00:08	00:00:28	00:27:36
get	0	-	21	00:00:00	00:00:00	00:00:00	23:00:00
higherEntry	138	1	21	00:00:12	00:00:08	00:00:29	00:27:36
isEmpty	138	1	21	00:00:12	00:00:08	00:00:26	00:27:36
lastKey	138	1	21	00:00:11	00:00:08	00:00:24	00:25:18
lowerEntry	138	1	21	00:00:12	00:00:08	00:00:25	00:27:36
pollLastEntry	138	2	21	00:02:28	00:01:38	00:04:10	05:15:44
pollFirstEntry	138	2	21	00:02:22	00:01:33	00:04:02	05:26:36
put	0	-	21	-	-	-	23:00:00
remove	0	-	21	-	-	-	23:00:00
ceilingEntry	138	1	21	00:00:12	00:00:08	00:00:29	00:27:36
containsKey	138	1	21	00:00:12	00:00:08	00:00:29	00:27:36
firstKey	138	1	21	00:00:12	00:00:08	00:00:28	00:27:36
floorKey	138	1	21	00:00:12	00:00:08	00:00:28	00:27:36
higherKey	138	1	21	00:00:11	00:00:08	00:00:28	00:25:18
floorEntry	138	1	21	00:00:12	00:00:08	00:00:28	00:27:36
lowerKey	138	1	21	00:00:11	00:00:08	00:00:26	00:25:18
lastEntry	138	1	21	00:00:12	00:00:08	00:00:27	00:27:36

Cuadro 5.3: Caso de estudio: **Map** - Técnica nro.1

De los 21 métodos analizados en **Map** no se encontraron *workarounds* para **clear**, **get**, **put** y **remove**. Esto se debe a que la redundancia intrínseca en **Map** no es tan alta como en los casos anteriores. Analizando el método **remove**, por ejemplo, vemos que podría ser remplazado por un *workaround* que invoque al método **get** y luego **pollFirstEntry** o **pollLastEntry** pero únicamente en aquellos casos en donde el par (**clave**, **valor**) a eliminar contenga en su campo **clave** el menor o mayor de las claves almacenadas en el **Map**, lo cual no ha sucedido en los estados iniciales evaluados.

De los 17 métodos restantes incluidos en el experimento, para 15 de ellos lograron computarse *workarounds* en tiempos promedio comprendidos entre 11 y 12 segundos (tiempos aceptables teniendo en cuenta el tamaño y la complejidad de las estructuras de datos). Sólo para los métodos **pollLastEntry** y **pollFirstEntry** el tiempo promedio de cómputo fue de 00:02:28 y 00:02:22

respectivamente. Este incremento se debe principalmente a que la longitud promedio de workarounds para estos métodos fue computada mediante 2 *loops unrolls*.

TimeOfDay

La clase `TimeOfDay` representa un instante de tiempo determinado, compuesto de horas, minutos, segundos y milisegundos. Sobre un objeto `TimeOfDay` es posible realizar operaciones de suma y resta, entre otras, de distintos instantes de tiempo. `TimeOfDay` se define como una 4-tupla (`horas`, `minutos`, `segundos`, `milisegundos`) donde las componentes representan respectivamente la hora, los minutos, los segundos y los milisegundos de un día. Debido a la aritmética utilizada por los métodos de la clase, fue de interés analizar los resultados experimentales variando el ancho de bits usado para la representación de los números enteros. Los experimentos fueron realizados para diferentes anchos de bits: 16, 24 y 32 bits. En los cuadros 5.4, 5.5 y 5.6 se muestran los resultados, respectivamente.

5.1. EVALUACIÓN DE TÉCNICA 1: CÓMPUTO DE SECUENCIAS DE MÉTODOS PARA WORKAROUNDS TRANSITORIOS

TimeOfDay: 48 estados						
Método	#wa	Long. prom. wa.	Tiempo prom.	Tiempo mín.	Tiempo máx.	Tiempo total
minusHours	48	1	00:00:10	00:00:04	00:00:15	00:08:23
minusMillis	1	1	00:00:10	00:00:03	00:00:12	07:50:10
minusMinutes	9	1	00:00:10	00:00:04	00:00:14	06:31:31
minusPeriodHours	48	1	00:00:11	00:00:04	00:00:14	00:08:42
minusPeriodMillis	1	1	00:00:10	00:00:03	00:00:16	07:50:10
minusPeriodMinutes	9	1	00:00:10	00:00:04	00:00:14	06:31:33
minusPeriodSeconds	9	1	00:00:10	00:00:03	00:00:14	06:31:30
minusSeconds	9	1	00:00:10	00:00:04	00:00:14	06:31:32
plusHours	48	1	00:00:11	00:00:04	00:00:16	00:08:58
plusMillis	1	1	00:00:13	00:00:05	00:00:16	07:50:13
plusMinutes	29	1	00:00:10	00:00:03	00:00:16	03:14:50
plusPeriodHours	48	1	00:00:11	00:00:04	00:00:15	00:08:45
plusPeriodMillis	1	1	00:00:12	00:00:05	00:00:15	07:50:12
plusPeriodMinutes	29	1	00:00:10	00:00:03	00:00:14	03:14:50
plusPeriodSeconds	29	1	00:00:14	00:00:05	00:00:16	03:16:45
plusSeconds	29	1	00:00:12	00:00:04	00:00:15	03:15:34
withHourOfDay	48	1	00:00:12	00:00:05	00:00:15	00:09:22
withMillisOfSecond	48	1	00:00:10	00:00:04	00:00:14	00:08:16
withMinuteOfHour	48	1	00:00:11	00:00:05	00:00:15	00:08:48
withSecondOfMinute	48	1	00:00:10	00:00:04	00:00:14	00:08:21
getHourOfDay	0	-	-	-	-	08:00:00
getMillisOfSecond	0	-	-	-	-	08:00:00
getMinuteOfHour	0	-	-	-	-	08:00:00
getSecondOfMinute	0	-	-	-	-	08:00:00

Cuadro 5.4: Caso de estudio: TimeOfDay - 16 bits - Técnica 1

5.1. EVALUACIÓN DE TÉCNICA 1: CÓMPUTO DE SECUENCIAS DE MÉTODOS PARA WORKAROUNDS TRANSITORIOS

TimeOfDay: 48 estados						
Método	#wa	Long. prom. wa.	Tiempo prom.	Tiempo mín.	Tiempo máx.	Tiempo total
minusHours	48	1	00:00:08	00:00:06	00:00:12	00:06:24
minusMillis	29	1	00:01:24	00:00:55	00:01:37	03:50:36
minusMinutes	35	1	00:00:20	00:00:16	00:00:22	02:21:40
minusPeriodHours	48	1	00:00:08	00:00:06	00:00:13	00:06:24
minusPeriodMillis	29	1	00:01:25	00:00:54	00:01:38	03:51:05
minusPeriodMinutes	35	1	00:00:20	00:00:14	00:00:23	02:21:40
minusPeriodSeconds	35	1	00:00:23	00:00:18	00:00:24	02:23:25
minusSeconds	35	1	00:00:22	00:00:17	00:00:24	02:22:50
plusHours	48	1	00:00:09	00:00:07	00:00:11	00:07:12
plusMillis	30	1	00:01:24	00:00:55	00:01:36	03:42:00
plusMinutes	38	1	00:00:21	00:00:12	00:00:23	01:53:18
plusPeriodHours	48	1	00:00:09	00:00:06	00:00:11	00:07:12
plusPeriodMillis	30	1	00:01:24	00:00:54	00:01:38	03:42:00
plusPeriodMinutes	35	1	00:00:21	00:00:12	00:00:23	02:22:15
plusPeriodSeconds	35	1	00:00:23	00:00:18	00:00:25	02:23:25
plusSeconds	31	1	00:00:25	00:00:16	00:00:29	03:02:55
withHourOfDay	48	1	00:00:13	00:00:09	00:00:15	00:10:24
withMillisOfSecond	48	1	00:00:12	00:00:08	00:00:14	00:09:36
withMinuteOfHour	48	1	00:00:13	00:00:08	00:00:15	00:10:24
withSecondOfMinute	48	1	00:00:11	00:00:08	00:00:13	00:08:48
getHourOfDay	0	-	-	-	-	08:00:00
getMillisOfSecond	0	-	-	-	-	08:00:00
getMinuteOfHour	0	-	-	-	-	08:00:00
getSecondOfMinute	0	-	-	-	-	08:00:00

Cuadro 5.5: Caso de estudio: TimeOfDay - 24 bits - Técnica 1

TimeOfDay: 48 estados						
Método	#wa	Long. prom. wa.	Tiempo prom.	Tiempo mín.	Tiempo máx.	Tiempo total
minusHours	48	1	00:00:12	00:00:08	00:00:13	00:09:36
minusMillis	48	1	00:01:32	00:01:20	00:03:53	01:13:36
minusMinutes	46	1	00:00:30	00:00:24	00:02:39	00:43:00
minusPeriodHours	48	1	00:00:12	00:00:09	00:00:14	00:09:36
minusPeriodMillis	45	1	00:01:31	00:00:54	00:01:38	01:38:15
minusPeriodMinutes	46	1	00:00:32	00:00:23	00:02:35	00:44:32
minusPeriodSeconds	47	1	00:00:45	00:00:23	00:03:22	00:45:15
minusSeconds	47	1	00:00:46	00:00:24	00:03:23	00:46:02
plusHours	48	1	00:00:15	00:00:09	00:00:17	00:12:00
plusMillis	48	1	00:01:31	00:00:35	00:02:06	01:12:48
plusMinutes	38	1	00:00:31	00:00:16	00:02:33	01:59:38
plusPeriodHours	48	1	00:00:14	00:00:10	00:00:16	00:11:12
plusPeriodMillis	48	1	00:01:32	00:00:22	00:03:22	01:13:36
plusPeriodMinutes	48	1	00:00:31	00:00:19	00:01:39	00:24:48
plusPeriodSeconds	48	1	00:00:42	00:00:22	00:04:05	00:33:36
plusSeconds	48	1	00:00:44	00:00:24	00:04:10	00:35:12
withHourOfDay	48	1	00:00:15	00:00:10	00:00:17	00:12:00
withMillisOfSecond	48	1	00:00:15	00:00:10	00:00:18	00:12:00
withMinuteOfHour	48	1	00:00:16	00:00:11	00:00:17	00:12:48
withSecondOfMinute	48	1	00:00:14	00:00:12	00:00:15	00:11:12
getHourOfDay	0	-	-	-	-	08:00:00
getMillisOfSecond	0	-	-	-	-	08:00:00
getMinuteOfHour	0	-	-	-	-	08:00:00
getSecondOfMinute	0	-	-	-	-	08:00:00

Cuadro 5.6: Caso de estudio: TimeOfDay - 32 bits - Técnica 1

En las tablas se puede observar que el tiempo promedio de cómputo de workarounds se incrementa a medida que se extiende la cantidad de bits de representación de los números enteros involucrados. El promedio general de cómputo de workarounds para los métodos de TimeOfDay con un ancho de bits de 16 es de 17.3 segundos, para 24 bits es de 29.7 segundos y para 32 bits es de 39 segundos. En contrapartida, a medida que aumenta la cantidad de bits de representación, la cantidad de workarounds también crece. La cantidad promedio de workarounds computados para un ancho de 16 bits es 31.9, para 24 bits es 39, y 47.5 para 32 bits. Esto se debe básicamente a que tanto las operaciones aritméticas requeridas como así también los parámetros enteros generados deben ser representables en el ancho de bits fijado.

El incremento del ancho de bits para la representación de los enteros es, en este caso de estudio, un factor importante para poder elevar la cantidad de workarounds computados. Si bien es cierto que comparando un ancho de 24 con 16 bits se puede apreciar un incremento en la cantidad de workarounds computados en tiempos promedio levemente superiores, es con un ancho de

32 bits donde se alcanza a computar workarounds para casi la totalidad de los estados iniciales para todos los métodos. Estos tiempos pueden ser mejorados incorporando esquemas de workarounds al procedimiento de cómputo, como ya fue mencionado anteriormente. Si se cuenta con el hardware apropiado, es posible ejecutar técnica 1 en paralelo para los anchos de bits considerados. Cada ejecución buscará computar workarounds con un ancho de bits diferente, retornando el workaround del primero que se encuentre.

Notar que en este trabajo se utilizó una implementación propia y eficiente de enteros, con una representación precisa a nivel de bits de hasta 32 bits. Esto habilita el análisis de especificaciones con operaciones numéricas complejas como la de `TimeOfDay`. Por el contrario, para un ancho de bits dado, los enteros por defecto de Alloy definen un átomo por cada uno de los enteros posibles dentro del ancho de bits. En la práctica, la escalabilidad de esta solución es pobre a medida que aumenta el ancho de bits (y no permite soportar el análisis de `TimeOfDay`).

5.1.1. Evaluación de *esquemas de workarounds*

Para mejorar los tiempos de cómputo de workarounds aplicando técnica 1 se propuso (sección 3.6.1) incorporar *esquemas de workarounds*. Para evaluar los beneficios de incorporar esquemas se analizaron los *workarounds transitorios* obtenidos para los casos `List`, `Set` y `Map` (presentados en las tablas 5.1, 5.2 y 5.3 respectivamente). En cada caso fueron seleccionados los *workarounds transitorios* obtenidos en los diferentes estados de invocación de cada método (salvo instanciación de parámetros). Luego éstos fueron generalizados a *esquemas de workarounds* (de acuerdo a lo detallado en la sección 3.6) para luego ser almacenados en el repositorio `WA Schema Database`. Recordemos que el repositorio `WA Schema Database` almacena para cada método un conjunto de esquemas de workaround (programas `DynAlloy`) cuya instanciación por el SAT-Solver posibilitará acelerar el cómputo de workarounds en algunos casos.

En las tablas 5.7, 5.8 y 5.9 se presenta la comparativa entre los tiempos promedio de cómputo de workarounds obtenidos para los métodos seleccionados de `List`, `Set` y `Map` (presentados en la sección 5.1), y los tiempos promedio de cómputo obtenidos de aplicar el mismo procedimiento incorporando *esquemas de workaround* (con la base de datos de esquemas inicializada como se indicó anteriormente).

Las tablas con los resultados cuentan con el nombre del método (`Método`),

5.1. EVALUACIÓN DE TÉCNICA 1: CÓMPUTO DE SECUENCIAS DE MÉTODOS PARA WORKAROUNDS TRANSITORIOS

Método	Tiempo prom.	Tiempo prom.+ esquemas	Aceleración
add	0:00:19	0:00:04	4.7
addFirst	0:00:18	0:00:04	4.7
addLast	0:00:19	0:00:04	4.7
add2	0:00:19	0:00:04	4.75
contains	0:00:17	0:00:02	8.5
element	0:00:16	0:00:03	5.3
get	0:00:19	0:00:03	6.3
getFirst	0:00:16	0:00:03	5.3
getLast	0:00:16	0:00:03	5.3
indexOf	0:00:19	0:00:04	4.75
isEmpty	0:00:15	0:00:02	7.5
lastIndexOf	0:00:19	0:00:04	4.75
offer	0:00:18	0:00:04	4.7
offerFirst	0:00:18	0:00:04	4.7
offerLast	0:00:18	0:00:04	4.7
peek	0:00:17	0:00:03	5.6
pool	0:00:16	0:00:03	5.3
pollFirst	0:00:16	0:00:03	5.3
pollLast	0:00:16	0:00:03	5.3
pop	0:00:17	0:00:03	5.6
push	0:00:19	0:00:04	4.7
remove	0:00:19	0:00:03	6.3
removeFirst	0:00:16	0:00:03	5.3
removeFO	0:00:19	0:00:04	4.75
removeLast	0:00:19	0:00:03	6.3
removeIndex	0:00:19	0:00:04	4.75
removeElem	0:00:19	0:00:04	4.75
setElement	0:00:57	0:00:04	14.25

Cuadro 5.7: Aceleración aplicando *esquemas de workarounds* para List

tiempo promedio de cómputo sin esquemas (Tiempo prom.), tiempo promedio de cómputo aplicando esquemas (Tiempo prom.+ esquemas) y cuánto más rápido es el cómputo de workarounds usando esquemas (Aceleración).

Método	Tiempo prom.	Tiempo prom.+ esquemas	Aceleración
add	0:01:58	0:00:22	5.3
ceiling	0:00:38	0:00:09	4.22
contains	0:00:13	0:00:04	3.25
first	0:00:37	0:00:05	7.4
floor	0:00:34	0:00:04	8.5
higher	0:00:40	0:00:09	4.4
isEmpty	0:00:39	0:00:05	7.8
last	0:00:39	0:00:03	13
lower	0:00:34	0:00:04	8.5
pollFirst	0:00:16	0:00:03	5.3
pollLast	0:00:18	0:00:03	6
remove	0:00:39	0:00:04	9.75

Cuadro 5.8: Aceleración aplicando *esquemas de workarounds* para Set

Método	Tiempo prom.	Tiempo prom.+ esquemas	Aceleración
ceilingKey	12	3	4
containsValue	12	1	12
firstEntry	12	3	4
higherEntry	12	3	4
isEmpty	12	1	12
lastKey	11	3	3.6
lowerEntry	12	3	4
pollLastEntry	148	10	14.8
pollFirstEntry	142	11	12.9
ceilingEntry	12	3	4
containsKey	12	1	12
firstKey	12	3	4
floorKey	12	3	4
higherKey	11	3	3.6
floorEntry	12	3	4
lowerKey	11	3	3.6
lastEntry	12	3	4

Cuadro 5.9: Aceleración aplicando *esquemas de workarounds* para Map

Para el caso de listas se evaluaron esquemas para 28 métodos de un total de 29. El método excluido fue `clear` para el cual no se había podido computar *workarounds transitorios*.

El tiempo de cómputo promedio general de *workarounds* fue de 19.42 segundos sin aplicar esquemas, y de 3.3 segundos aplicando esquemas. La aceleración promedio general fue de 5.6 veces más rápida incorporando esquemas, con un máximo de 14.25 veces para el método `setElement`.

Para el caso de `Set` se evaluaron esquemas para 12 métodos de un total de 13. El método excluido es también fue `clear` para el cual no se lograron computar *workarounds transitorios*. El promedio de cómputo general sin aplicar esquemas fue de 38.75 segundos y de 6.25 segundos aplicando esquemas. La aceleración promedio general lograda en el cómputo fue de 6.36 veces más rápida aplicando esquemas, alcanzando un máximo de 9.75 para el método `remove`.

Finalmente para `Map` se evaluaron esquemas para 17 métodos de un total de 21. Para los métodos `get`, `put`, `clear` y `remove` no fueron posible definir esquemas (debido a que no se lograron inicialmente computar *workarounds transitorios* para estos métodos). El promedio de cómputo general obtenido sin aplicar esquemas fue de 27.47 segundos y de 3.52 segundos aplicando esquemas. La aceleración promedio general lograda en el cómputo fue de 6.5 veces más rápida aplicando esquemas, alcanzando un máximo de 14.8 para el método `pollLastEntry`.

La significativa baja en los tiempos de cómputo, como ya fue explicado en la sección 3.6, se debe a la simplificación de la tarea del SAT-Solver. Una vez fijada la secuencia de acciones el SAT-Solver se encargará de computar valores para los parámetros involucrados en el esquema.

En vista de estos resultados, si se cuenta con el hardware apropiado para almacenar *esquemas de workarounds*, luego ejecutar el cómputo de workarounds con y sin esquemas en paralelo, y retornar el workaround computado por la primera de las ejecuciones en terminar (como se describió en la sección 3.6), puede traer beneficios significativos.

5.2. Workarounds transitorios versus permanentes

En esta sección se realiza un análisis comparativo entre la técnica 1 propuesta en esta tesis (para computar *workarounds transitorios*) y la técnica SBES Search-Based Equivalent Synthesis [45] (para computar secuencias de métodos equivalentes o *workarounds permanentes*). El objetivo de este análisis es evaluar la contribución de nuestras técnicas de cómputo de *workarounds transitorios* teniendo en cuenta la existencia de SBES, que computa *workarounds permanentes*.

Para comparar estas dos técnicas se tomaron clases evaluadas con SBES en [45]. Se definieron los contratos JML para sus métodos y se asumieron los supuestos planteados en la sección 5.1 para técnica 1, es decir, se ha ejecutado un método que ha ocasionado una falla, luego de aplicar una acción de *rollback* a su estado de invocación se procede a ejecutar técnica 1 para alcanzar un estado de recuperación que verifique la postcondición del método defectuoso. El tiempo límite de búsqueda para la búsqueda de *workarounds transitorios* fue fijado en 10 minutos. Los estados de invocación fueron generados utilizando *Randoop* al igual que para los casos evaluados con técnica 1.

Las clases evaluadas fueron las siguientes [45]:

- **Stack**⁹: La clase Stack modela una pila de objetos LIFO (last-in-first-out). **Stack** extiende de la clase **Vector** con cinco operaciones que permiten tratar un vector como una pila.

⁹<https://docs.oracle.com/javase/7/docs/api/java/util/Stack.html>

- De la librería **Graphstream**¹⁰ (biblioteca de Java para el modelado y análisis de grafos dinámicos) fueron evaluadas las siguientes clases:
 - **Vector2**¹¹: provee una implementación de pares ordenados de números.
 - **Vector3**¹²: provee una implementación de tuplas de tres valores numéricos.
 - **Edge**: permite definir y manipular aristas entre nodos de un grafo.
 - **Path**: permite almacenar listas ordenadas de nodos y aristas adyacentes. Un **Path** define una estructura dinámica donde es posible agregar aristas para construir caminos en grafos.

Los resultados de la ejecución de los experimentos son presentados en tablas en donde se incluye: la cantidad de estructuras analizadas junto con su tamaño mínimo, máximo y promedio. Para las clases **Vector2**, **Vector3** y **Edge** sólo se incluyen la cantidad de estructuras analizadas ya que sus tamaños son constantes. **Vector2** modela un vector que contiene 2 valores numéricos, **Vector3** modela un vector con una terna de valores numéricos y **Edge** define una arista entre un par de nodos más una serie de atributos asociados, como por ejemplo, su identificador y si la arista es o no dirigida.

Las tablas incluyen el método analizado (**Método**), la cantidad de *workarounds transitorios* computados aplicando técnica 1 (**#Wa.**) para las estructuras consideradas, y la columna **Wa. permanente (Si/No)** donde se registra si la técnica SBES ha logrado computar un *workaround permanente* para dicho método (según lo reportado en [45]).

Stack

En el cuadro 5.10 se muestran los resultados para el caso de estudio **Stack**. Para este caso fueron generadas con *Randoop* 141.000 **Stack** de las cuales se seleccionaron 141 (1 cada 1000).

¹⁰<https://graphstream-project.org/>

¹¹<https://data.graphstream-project.org/api/gs-core/1.3/index.html?org/graphstream/ui/geom/Vector2.html>

¹²<https://data.graphstream-project.org/api/gs-core/1.3/index.html?org/graphstream/ui/geom/Vector3.html>

Stack (núm./ mín./ máx./ prom.) : 141 / 5 / 19 / 12.19		
Método	#Wa.	Wa. permanente (Si/No)
stackContains	141	No
stackIsEmpty	141	No
stackFirstElement	141	Si
stackPeek	141	Si
stackPop	141	Si
stackPush	141	Si
vectorAdd	141	Si
vectorAddElement	141	Si
vectorAddIndexItem	141	Si
vectorClear	141	Si
vectorElementAt	141	Si
vectorGet	141	Si
vectorInsertElementAt	141	No
vectorLastElement	141	Si
vectorRemoveAllElements	141	No
vectorRemoveElement	117	Si
vectorRemoveElementAt	141	Si
vectorRemoveIndex	141	Si
vectorSet	141	Si

Cuadro 5.10: Cómputo de *workarounds* para Stack

Debido a que en la evaluación experimental de **Stack** [45] se incluyen métodos heredados de la clase `java.util.Vector`¹³, los mismos han sido incluidos con el prefijo `vector` en nuestra evaluación.

Para los 19 métodos evaluados de **Stack** pudieron computarse *workarounds transitorios* para la totalidad de los estados de invocación evaluados (141), a excepción del método `vectorRemoveElement` para el cual se computaron *workarounds transitorios* para 117 estados.

Por otro lado se puede observar que si bien para la mayoría de los métodos existen secuencias de métodos permanentes de recuperación, se observan algunos métodos para los cuales sólo existen alternativas transitorias tales como `stackContains`, `stackIsEmpty`, `vectorRemoveAllElements` y `vectorInsertElementAt`.

Vector2 y Vector3

Para la evaluación de la clase **Vector2** y **Vector3** fueron generados 200.000 y 180.000 estados de invocación respectivamente. De los cuales fueron selec-

¹³<https://docs.oracle.com/javase/7/docs/api/java/util/Vector.html>

cionados 200 para la clase `Vector2` y 180 para la clase `Vector3`. Los resultados son presentados en las tablas 5.11 y 5.12.

Vector2 (núm) : 200		
Método	#Wa.	Wa. permanente (Si/No)
add	200	No
at	200	No
copy	200	Si
equals	200	No
fill	200	Si
x	200	Si
y	200	Si
isZero	200	No
scalarAdd	200	No
scalarSub	200	No
set2	200	No
set	200	Si
sub	200	No
validComponent	200	No

Cuadro 5.11: Cómputo de *workarounds* para `Vector2`

Vector3 (núm) : 180		
Método	#Wa.	Wa. permanente (Si/No)
add	180	No
at	180	No
copy	180	Si
equals	180	No
fill	180	Si
x	180	Si
y	180	Si
z	180	Si
isZero	180	No
scalarAdd	180	No
scalarSub	180	No
set2	180	No
set	180	Si
sub	180	No
validComponent	180	No

Cuadro 5.12: Cómputo de *workarounds* para `Vector3`

La clase `Vector2` fue evaluada para 14 métodos (ver tabla 5.11), para todos ellos se lograron computar *workarounds transitorios* en la totalidad (200) de los estados evaluados. De estos 14 métodos, SBES [45] reporta que

logró computar secuencias equivalentes para 5 métodos (`copy`, `x`, `y`, `fill` y `set`). Los 9 métodos restantes tendrán la posibilidad de ser recuperados sólo mediante un *workaround transitorio*.

Los resultados experimentales para la clase `Vector3` (tabla 5.12) son similares a los obtenidos para la clase `Vector2`. En este caso la cantidad de métodos evaluados fueron 15. Para todos los métodos analizados se logró computar alternativas transitorias en la totalidad de los estados evaluados (180). Para este caso SBES reporta el cómputo de secuencias equivalentes para 6 métodos (`copy`, `x`, `y`, `z`, `fill` y `set`).

Edge y Path

Para las clases `Edge` y `Path` fueron generados con *Randoop* 190.000 y 60.000 estados de invocación respectivamente. Para `Edge` fueron seleccionados 190 y para `Path` 60. Dentro de los estados de invocación para la evaluación de los métodos de `Path`, *Randoop* generó caminos de longitudes comprendidas entre 1 y 15 nodos.

Edge (núm) : 190		
Método	#Wa.	Wa. permanente (Si/No)
<code>getNode0</code>	190	Si
<code>getNode1</code>	190	Si
<code>getOpposite</code>	190	No
<code>getSource</code>	190	Si
<code>getTarget</code>	190	Si
<code>isDirected</code>	190	No
<code>isLoop</code>	190	No
<code>setAttribute</code>	190	Si
<code>addAttribute</code>	190	Si
<code>addAttributes</code>	190	No
<code>changeAttribute</code>	190	Si
<code>removeAttribute</code>	11	No
<code>clearAttributes</code>	53	No
<code>getAttribute</code>	190	Si
<code>getFirstAttribute</code>	190	Si

Cuadro 5.13: Cómputo de *workarounds* para `Edge`

Para la clase `Edge` se evaluaron 15 métodos (ver tabla 5.13). Técnica 1 logró computar *workarounds transitorios* para 13 métodos en la totalidad de los estados, el método `removeAttribute` pudo ser recuperado en 11 estados y `clearAttributes` en 53 estados.

Por su parte SBES logró computar *workarounds permanentes* para 9 de los 15 métodos. Para los métodos `getOpposite`, `isDirected`, `isLoop`, `addAttributes`, `removeAttributes` y `clearAttributes` SBES no pudo computar alternativas permanentes. Para estos métodos técnica 1 brinda una posibilidad de recuperación.

Path (núm./ mín./ máx./ prom.) : 60 / 1 / 15 / 7.9		
Método	#Wa.	Wa. permanente (Si/No)
<code>addEdge</code>	60	No
<code>addNodeEdge</code>	60	No
<code>clearPath</code>	19	No
<code>containsEdge</code>	60	No
<code>containsNode</code>	60	No
<code>empty</code>	60	No
<code>getEdgeCount</code>	60	Si
<code>getNodeCount</code>	60	Si
<code>getRoot</code>	60	No
<code>peekEdge</code>	55	No
<code>peekNode</code>	19	No
<code>popEdge</code>	21	No
<code>popNode</code>	60	No
<code>pushEdge</code>	60	No
<code>pushNodeEdge</code>	60	No
<code>setRoot</code>	16	No
<code>sizePath</code>	60	No

Cuadro 5.14: Cómputo de *workarounds* para Path

La clase `Path` fue evaluada para 17 de sus métodos (ver tabla 5.14). Sólo para los métodos `getNodeCount` y `getEdgeCount` SBES logró computar *workarounds permanentes*. Técnica 1 pudo computar *workarounds transitorios* para los 17 métodos evaluados, para 10 de ellos se computaron alternativas de ejecución transitorias en la totalidad de los casos evaluados, el resto de los métodos pudieron ser recuperados en diferentes cantidades de estados `clearPath` (19), `peekEdge` (55), `peekNode` (19), `peekEdge` (21) y `setRoot` (16).

De los casos evaluados en esta sección es posible observar que existe una cantidad significativa de métodos que sólo pueden ser recuperados aplicando técnica 1. Las técnicas no son excluyentes, sino que más bien pueden complementarse. En aquellos casos en donde no existen *workarounds permanentes* podrá aplicarse técnica 1 a fin de computar un *workarounds transitorio*.

5.3. Evaluación de técnica 2: Generación de estados de recuperación

La segunda técnica fue evaluada observando su eficiencia en la producción de estados de recuperación.

Del mismo modo que para la evaluación de la técnica anterior, para técnica 2 se asumió la ocurrencia de una falla al ejecutar un método en un estado inicial generado aleatoriamente. A partir de este estado inicial se ejecuta el procedimiento de cómputo de un estado de recuperación que verifique la postcondición del método defectuoso.

La evaluación experimental para técnica 1 fue realizada sobre interfaces debido a que los *workarounds transitorios* son computados mediante una combinación de métodos, especificados en términos de las interfaces, que al ejecutarse generan el estado de recuperación del método defectuoso en cuestión. En cambio técnica 2 fue evaluada sobre implementaciones de las interfaces, debido a que los estados de recuperación concretos para el programa Java deben ser computados directamente por el SAT-solver. Los resultados son presentados en tablas divididas en dos secciones: la primera denominada **SAT** muestra la ejecución de los experimentos sin incluir predicados de rotura de simetrías y cotas ajustadas siguiendo el enfoque clásico de reparación de estados aplicando SAT [55,92], mientras que la segunda sección **SAT+Rotura Simetrías+Cotas ajustadas** refiere a la técnica 2 propuesta en esta tesis la cual incluye predicados de rotura de simetrías (sección 4.2) y cotas ajustadas (sección 4.3). De esta forma se puede observar el beneficio obtenido en cada método en cuanto a cantidad de *workarounds transitorios* computados, disminución en los tiempos promedio de recuperación e incremento del tamaño máximo de las estructuras recuperadas.

LinkedList y ApacheList

Los cuadros 5.15 y 5.16 muestran los resultados de aplicar la técnica sobre `LinkedList` y `ApacheList`.

LinkedList (núm / mín / máx / prom) : 141 / 11 / 21 / 13.68								
Método	SAT				SAT+Rotura Simetrías+Cotas ajustadas			
	#wa	#nodos máx.	Tiempo prom.	Tiempo total	#wa	#nodos máx.	Tiempo prom.	Tiempo total
push	141	21	00:00:02	00:04:42	141	21	00:00:03	00:02:21
addFirst	141	21	00:00:02	00:04:42	141	21	00:00:03	00:02:21
add	141	21	00:00:02	00:04:42	141	21	00:00:02	00:02:21
addLast	141	21	00:00:02	00:04:42	141	21	00:00:02	00:02:21
clear	141	21	00:00:01	00:02:21	141	21	00:00:02	00:02:21
contains	141	21	00:00:01	00:02:21	141	21	00:00:02	00:02:21
element	141	21	00:00:01	00:02:21	141	21	00:00:02	00:02:21
getFirst	141	21	00:00:01	00:02:21	141	21	00:00:02	00:02:21
get	141	21	00:00:01	00:02:21	141	21	00:00:02	00:02:21
getLast	141	21	00:00:02	00:04:42	141	21	00:00:03	00:04:42
indexOf	141	21	00:00:02	00:04:42	141	21	00:00:03	00:02:21
isEmpty	141	21	00:00:01	00:02:21	141	21	00:00:02	00:02:21
lastIndexOf	141	21	00:00:02	00:04:42	141	21	00:00:03	00:02:21
offer	141	21	00:00:02	00:04:42	141	21	00:00:02	00:02:21
offerFirst	141	21	00:00:02	00:04:42	141	21	00:00:02	00:02:21
offerLast	141	21	00:00:02	00:04:42	141	21	00:00:02	00:02:21
peek	141	21	00:00:02	00:04:42	141	21	00:00:02	00:02:21
pop	141	21	00:00:02	00:04:42	141	21	00:00:03	00:02:21
remove	141	21	00:00:02	00:04:42	141	21	00:00:03	00:04:42
removeIndex	141	21	00:00:14	00:32:54	141	21	00:00:14	00:07:03
removeFirst	141	21	00:00:02	00:04:42	141	21	00:00:03	00:02:21
removeLast	141	21	00:00:02	00:04:42	141	21	00:00:02	00:02:21
setElement()	141	21	00:00:01	00:02:21	141	21	00:00:02	00:02:21

Cuadro 5.15: Caso de estudio: LinkedList - Técnica 2

En el caso de `LinkedList`, de los 23 métodos analizados, en 22 de ellos el promedio de cómputo de reparación de estados siguiendo el enfoque clásico fue de 1 o 2 segundos. Para el método `removeIndex` el promedio fue de 14 segundos. Los resultados aplicando técnica 2 presentan una leve disminución en los tiempos promedio.

Con respecto a `ApacheList`, la segunda implementación de listas evaluada, los resultados son similares a los obtenidos para `LinkedList`. El tiempo promedio de cómputo para los métodos de `ApacheList` se incrementan levemente en 1 o 2 segundos en casi la totalidad de los métodos comparando con los mismos métodos evaluados en `LinkedList`.

Con respecto a la cantidad de estados recuperados es posible observar que ambas técnicas han computado estados de recuperación en el 100% de los estados iniciales evaluados tanto para `LinkedList` como para `ApacheList`. Del mismo modo se han logrado computar estados de recuperación, en ambos casos, para listas del tamaño máximo evaluado (21 elementos).

La poca ganancia de técnica 2 en implementaciones de listas se debe a

ApacheList (núm./ mín./ máx./ prom.) : 141 / 11 / 21 / 13.68								
Método	SAT				SAT+Rotura Simetrías+Cotas ajustadas			
	#wa	#nodos máx.	Tiempo prom.	Tiempo total	#wa	#nodos máx.	Tiempo prom.	Tiempo total
push	141	21	00:00:04	00:09:24	141	21	00:00:02	00:04:42
addFirst	141	21	00:00:03	00:07:03	141	21	00:00:02	00:04:42
add	141	21	00:00:04	00:09:24	141	21	00:00:02	00:04:42
addLast	141	21	00:00:02	00:04:42	141	21	00:00:02	00:04:42
clear	141	21	00:00:03	00:07:03	141	21	00:00:02	00:04:42
contains	141	21	00:00:02	00:04:42	141	21	00:00:02	00:04:42
element	141	21	00:00:04	00:09:24	141	21	00:00:01	00:01:53
getFirst	141	21	00:00:02	00:04:42	141	21	00:00:01	00:02:07
get	141	21	00:00:04	00:09:24	141	21	00:00:02	00:04:42
getLast	141	21	00:00:03	00:07:03	141	21	00:00:01	00:03:17
indexOf	141	21	00:00:03	00:07:03	141	21	00:00:01	00:02:07
isEmpty	141	21	00:00:02	00:04:42	141	21	00:00:01	00:01:53
lastIndexOf	141	21	00:00:03	00:07:03	141	21	00:00:01	00:01:53
offer	141	21	00:00:04	00:09:24	141	21	00:00:01	00:01:53
offerFirst	141	21	00:00:03	00:07:03	141	21	00:00:01	00:02:07
offerLast	141	21	00:00:02	00:04:42	141	21	00:00:01	00:01:53
peek	141	21	00:00:02	00:04:42	141	21	00:00:01	00:01:39
pop	141	21	00:00:04	00:09:24	141	21	00:00:01	00:01:53
remove	141	21	00:00:04	00:09:24	141	21	00:00:02	00:04:42
removeIndex	141	21	00:00:19	00:44:39	141	21	00:00:05	00:11:45
removeFirst	141	21	00:00:03	00:07:03	141	21	00:00:01	00:02:21
removeLast	141	21	00:00:02	00:04:42	141	21	00:00:01	00:01:53
setElement	141	21	00:00:02	00:04:42	141	21	00:00:01	00:01:53

Cuadro 5.16: Caso de estudio: ApacheList - Técnica 2

la baja complejidad de estas estructuras (son las estructuras más simples en toda nuestra evaluación experimental). Es posible que para estructuras de mayor tamaño que las generadas, los beneficios de usar técnica 2 empiecen a ser visibles (teniendo en cuenta los resultados de las estructuras más complejas que se presentan a continuación).

El caso particular del método `clear` que no había podido ser recuperado aplicando técnica 1 (con un timeout de 10 minutos), sí se pudo recuperar aplicando técnica 2. Esto se debe a que para satisfacer la postcondición del método `clear` el SAT-Solver debe simplemente generar una lista vacía.

TreeSet

Se presentan en esta sección los resultados de aplicar técnica 2 para **TreeSet**. En primer lugar se observa en el cuadro 5.17 una diferencia significativa en lo que se refiere a cantidad de *workarounds transitorios* computados sin incluir predicados de rotura de simetría y cotas ajustadas e incluyéndolos. Recordemos que los casos no recuperados son aquellos que superaron un *timeout* de 10 minutos. Se puede ver también que al incorporar predicados de rotura de simetría y cotas ajustadas se lograron computar estados que involucran **TreeSets** de tamaño mayor, permitiendo incrementar el tamaño máximo de las estructuras analizadas. De un total de 136 estados evaluados para cada método, la cantidad promedio de *workarounds* computados sin las optimizaciones fue de 42, mientras que al incorporar predicados de rotura de simetrías y cotas ajustadas este valor se eleva a 109. Por otro lado, el tamaño máximo promedio de **TreeSet** para el cual se logró computar un *workaround* crece de 14 a 19 nodos.

Con respecto a los tiempos de cómputo, la disminución de los tiempos es significativa. De un promedio de cómputo general de 00:02:47 se reduce a 24 segundos al aplicar técnica 2. Cabe señalar que el promedio de cómputo general al cual se hace referencia en el párrafo anterior registra el promedio de tiempos computado sobre los estados recuperados. Por ejemplo en el caso del método **remove** de **TreeSet**, el tiempo promedio de cómputo registrado es de 00:02:01 (computado sobre 21 estados), mientras que para técnica 2 es de 00:00:32 (computado para 109 estados). Esta observación es importante tenerla en cuenta al momento de evaluar cuán significativa ha sido la mejora lograda por técnica 2 en lo que se refiere a tiempos cómputo.

La aplicación de técnica 2 en este caso de estudio incrementa notablemente la cantidad de estados recuperados, el tamaño máximo de la estructura recuperada y disminuye los tiempos promedio de cómputo, lo que favorece a su aplicación práctica.

BSTree y AVLTree

Los resultados experimentales para árboles binarios de búsqueda y árboles AVL se presentan en las tablas 5.18 y 5.19 respectivamente.

Para el caso de estudio **BSTree** la cantidad promedio general de *workarounds* computados sin predicados de rotura de simetría y cotas ajustadas fue de 133.5 mientras que aplicando técnica 2 fue de 135. El tiempo promedio

TreeSet (núm./ mín./ máx./ prom.) : 136 / 11 / 22 / 13.16								
Método	SAT				SAT+Rotura Simetrías+Cotas ajustadas			
	#wa	#nodos máx.	Tiempo prom.	Tiempo total	#wa	#nodos máx.	Tiempo prom.	Tiempo total
add	25	14	00:02:55	19:42:55	95	17	00:00:40	07:53:20
ceiling	136	22	00:00:09	00:20:24	136	22	00:00:06	00:13:36
clear	136	22	00:00:05	00:11:20	135	22	00:00:05	00:21:15
contains	23	13	00:04:29	20:33:07	136	22	00:00:05	00:11:20
first	23	12	00:04:02	20:22:46	87	19	00:00:31	08:54:57
last	33	13	00:04:11	19:28:03	90	17	00:00:37	08:35:30
floor	28	13	00:02:21	19:05:48	99	18	00:00:33	07:04:27
higher	27	12	00:02:38	19:21:06	99	18	00:00:35	06:47:45
isEmpty	28	13	00:03:33	19:59:24	104	19	00:00:26	06:05:04
lower	20	12	00:04:00	20:40:00	104	18	00:00:30	06:12:00
pollFirst	32	13	00:02:17	18:33:04	95	17	00:00:30	07:37:30
pollLast	36	13	00:03:31	18:46:36	128	22	00:00:06	01:32:48
remove	21	14	00:02:01	19:52:21	109	17	00:00:32	05:28:08

Cuadro 5.17: Caso de estudio: TreeSet - Técnica 2

BSTree (núm./ mín./ máx./ prom.) : 136 / 11 / 22 / 13.16								
Método	SAT				SAT+Rotura Simetrías+Cotas ajustadas			
	#wa	#nodos máx.	Tiempo prom.	Tiempo total	#wa	#nodos máx.	Tiempo prom.	Tiempo total
add	130	17	00:00:34	02:13:40	135	19	00:00:14	00:41:30
ceiling	136	22	00:00:09	00:20:24	136	22	00:00:06	00:13:36
clear	136	22	00:00:05	00:11:20	136	22	00:00:04	00:09:04
contains	135	22	00:00:25	01:06:15	136	22	00:00:14	00:31:44
first	136	19	00:00:32	01:12:32	134	19	00:00:10	00:42:20
last	132	22	00:00:29	01:43:48	136	22	00:00:09	00:20:24
floor	131	19	00:00:29	01:53:19	135	22	00:00:11	00:34:45
higher	129	18	00:00:22	01:57:18	132	18	00:00:13	01:08:36
isEmpty	133	19	00:00:16	01:05:28	136	22	00:00:12	00:27:12
lower	135	19	00:00:19	00:52:45	135	19	00:00:15	00:43:45
pollFirst	135	22	00:00:20	00:55:00	134	19	00:00:09	00:40:06
pollLast	133	18	00:00:24	01:23:12	135	19	00:00:11	00:34:45
remove	135	19	00:02:01	04:42:15	136	22	00:00:12	00:27:12

Cuadro 5.18: Caso de estudio: BSTree - Técnica 2

general de cómputo según el enfoque clásico fue de 29.6 segundos, mientras que con técnica 2 fue de 10.7 segundos. Podemos observar aquí una reducción significativa en estos tiempos. Finalmente, el tamaño promedio máximo de BSTree para el cual se pudieron computar workarounds aumentó de 19.5 a 20.5 nodos.

En el caso de AVLTree, la cantidad promedio general de workarounds computados sin predicados de rotura de simetría y cotas ajustadas fue de

AVLTree (núm./ mín./ máx./ prom.) : 136 / 11 / 22 / 13.16								
Método	SAT				SAT+Rotura Simetrías+Cotas ajustadas			
	#wa	#nodos máx.	Tiempo prom.	Tiempo total	#wa	#nodos máx.	Tiempo prom.	Tiempo total
add	130	17	00:02:55	07:19:10	135	19	00:00:28	01:13:00
ceiling	136	22	00:00:07	00:15:52	136	22	00:00:08	00:18:08
clear	136	22	00:00:05	00:11:20	136	22	00:00:05	00:11:20
contains	132	17	00:01:19	03:33:48	135	19	00:00:20	00:55:00
first	135	19	00:01:05	02:36:15	136	19	00:00:20	00:45:20
last	134	19	00:01:03	02:40:42	135	19	00:00:21	00:57:15
floor	134	19	00:01:13	03:03:02	135	19	00:00:21	00:57:15
higher	130	17	00:01:01	03:12:10	136	19	00:00:22	00:49:52
isEmpty	135	19	00:00:59	02:22:45	135	19	00:00:18	00:50:30
lower	133	19	00:00:57	02:36:21	135	19	00:00:19	00:52:45
pollFirst	134	18	00:00:48	02:07:12	136	22	00:00:21	00:47:36
pollLast	134	19	00:00:53	02:18:22	135	19	00:00:19	00:52:45
remove	132	17	00:02:01	05:22:10	136	22	00:00:22	00:49:52

Cuadro 5.19: Caso de estudio: AVLTree - Técnica 2

133.3 mientras que con técnica 2 fue de 135.4. El tiempo promedio general de cómputo de los estados de recuperación sin las optimizaciones fue de 66.53 segundos, mientras que con técnica 2 fue de 18.7. Nuevamente se observa una reducción importante en el tiempo. Finalmente, la cantidad promedio máxima de AVLTree para el cual se pudieron computar workarounds crece de 18.7 a 20 nodos. Particularmente, el método `remove` incrementó su estructura máxima recuperada de 17 a 22 nodos.

En ambos casos se observa, principalmente, una disminución significativa en los tiempos promedio de cómputo. En lo que se refiere a la cantidad de workarounds computados y a las estructuras máximas recuperadas, se mejoran levemente los valores.

TreeMap

En esta sección se presentan los resultados experimentales de aplicar técnica 2 para el caso de estudio `TreeMap` (cuadro 5.20).

Analizando los valores presentados es posible observar que el promedio general de la cantidad de workarounds computados al aplicar las optimizaciones crece significativamente de 34,5 a 127, de un total de 136 estados a recuperar. Particularmente para el método `containsKey` se logró computar estados de recuperación en sólo 7 estados de invocación siguiendo el enfoque clásico, mientras que al aplicar técnica 2 esta cantidad ascendió a 127.

TreeMap (núm./ mín./ máx./ prom.) : 138 / 11 / 21 / 13.68								
Método	SAT				SAT+Rotura Simetrías+Cotas ajustadas			
	#wa	#nodos máx.	Tiempo prom.	Tiempo total	#wa	#nodos máx.	Tiempo prom.	Tiempo total
ceilingEntry	138	21	00:00:08	00:19:19	138	21	00:00:07	00:16:06
clear	138	21	00:00:09	00:19:33	137	20	00:00:06	00:23:42
containsKey	7	12	00:02:56	22:10:32	127	19	00:01:52	05:47:04
containsValue	5	12	00:04:06	22:30:30	123	18	00:02:41	08:00:03
firstEntry	6	12	00:03:39	22:21:54	123	20	00:02:20	07:17:00
firstKey	3	21	00:03:16	22:39:48	124	18	00:02:19	07:07:16
floorKey	11	13	00:03:32	21:48:52	128	21	00:01:42	05:17:36
floorEntry	11	12	00:03:34	21:49:14	130	18	00:01:57	05:33:30
get	9	12	00:02:58	21:56:42	123	18	00:02:18	07:12:54
higherKey	14	12	00:01:49	21:05:26	129	18	00:01:59	05:45:51
higherEntry	8	12	00:01:01	21:48:08	129	18	00:02:08	06:05:12
isEmpty	8	13	00:03:11	22:05:28	138	18	00:01:47	04:06:06
lastKey	5	21	00:03:16	22:39:48	122	17	00:02:37	07:59:14
lastEntry	54	15	00:01:50	15:39:00	127	19	00:01:57	05:57:39
lowerKey	10	12	00:02:10	21:41:40	126	18	00:01:26	05:00:36
lowerEntry	9	12	00:01:01	21:48:08	128	18	00:02:05	06:06:40
put	85	14	00:01:37	11:07:25	107	18	00:01:28	07:46:56
remove	55	15	00:01:43	15:24:25	126	18	00:01:29	04:36:54
pollLast	58	15	00:01:43	15:24:25	129	18	00:01:29	05:11:21
pollEntry	56	15	00:01:43	15:24:25	126	18	00:01:25	25:58:30

Cuadro 5.20: Caso de estudio: TreeMap - Técnica 2

Se puede apreciar también una mejora significativa en el tiempo promedio de cómputo general, disminuyendo éstos de 00:02:26 a 00:01:46. Finalmente, el tamaño promedio máximo para el cual se logró computar un workaround creció de 14,6 a 18,55 nodos. Un ejemplo es el método `floorKey` que incrementó su estructura máxima recuperada de 13 a 21 elementos.

Ténganse en cuenta que, como se mencionaba para el caso de `TreeSet`, el tiempo promedio general de cómputo se obtiene de los estados de invocación a partir de los cuales se pudieron computar estados de recuperación. De incluir todos los estados el tiempo promedio general es más favorable aún para técnica 2. Por ejemplo, el tiempo promedio de cómputo para el método `containsKey` de `TreeMap` registrado siguiendo el enfoque clásico es de 00:02:56 (computado sobre 7 estados), de incluir los *timeouts* obtenidos para 131 estados restantes (de un total de 138) el tiempo promedio hubiese sido de 00:09:38, mientras que el promedio de cómputo general incluyendo los estados no recuperados con técnica 2 hubiese sido de 00:02:31.

Para `TreeMap` técnica 2 logra incrementar notablemente la cantidad de estados recuperados reduciendo los tiempos de cómputo, e incrementando en

algunos casos significativamente los tamaños de la estructuras analizadas.

TimeOfDay

Para el caso de estudio de `TimeOfDay` la estructura de datos definida posee cuatro campos enteros: (`horas`, `minutos`, `segundos`, `milisegundos`). Computar predicados de rotura de simetrías y cotas ajustadas, en este caso particular, no agrega ningún beneficio debido a que la misma no posee campos de tipo referencia. En los casos de estudio evaluados anteriormente el beneficio de aplicar predicados de rotura de simetrías y cotas ajustadas es significativo ya que son estructuras dinámicas alojadas en el *heap*, y poseen restricciones complejas sobre las posibles formas de asignar valores a sus campos (de tipo referencia).

Interesa en este caso evaluar la generación de estados de recuperación tomando diferentes anchos de bits para la representación de números enteros. Para ello se hace uso de una especificación Alloy propia para enteros de 16, 24 y 32 bits. La complejidad en este caso de estudio está dada por las operaciones aritméticas que se realizan en sus métodos. Interesa determinar las relaciones entre tamaño de bits utilizado, cantidad de estados de recuperación computados y tiempos obtenidos.

En los cuadros 5.21, 5.22 y 5.23 se presentan los resultados experimentales de la ejecución de técnica 2 incrementando el ancho de bits de representación de los enteros.

Método	TimeOfDay: 48		
	SAT		
	#wa	Tiempo prom.	Tiempo total
minusHours	48	00:00:02	00:01:36
minusMillis	14	00:00:02	05:40:28
minusMinutes	30	00:00:02	03:01:00
minusPeriodHours	48	00:00:02	00:01:36
minusPeriodMillis	14	00:00:02	05:40:28
minusPeriodMinutes	30	00:00:02	03:01:00
minusPeriodSeconds	28	00:00:02	03:20:56
minusSeconds	28	00:00:02	03:20:56
plusHours	48	00:00:02	00:01:36
plusMillis	10	00:00:02	06:20:20
plusMinutes	31	00:00:02	02:51:02
plusPeriodHours	48	00:00:02	00:01:36
plusPeriodMillis	14	00:00:02	05:40:28
plusPeriodMinutes	31	00:00:02	02:51:02
plusPeriodSeconds	25	00:00:02	03:50:50
plusSeconds	28	00:00:02	03:20:56
withHourOfDay	48	00:00:01	00:00:48
withMillisOfSecond	48	00:00:01	00:00:48
withMinuteOfHour	48	00:00:01	00:00:48
withSecondOfMinute	48	00:00:01	00:00:48
getHourOfDay	48	00:00:01	00:00:48
getMillisOfSecond	48	00:00:01	00:00:48
getMinuteOfHour	48	00:00:01	00:00:48
getSecondOfMinute	48	00:00:01	00:00:48

Cuadro 5.21: Caso de estudio: TimeOfDay - 16 bits

Método	TimeOfDay: 48		
	SAT		
	#wa	Tiempo prom.	Tiempo total
minusHours	48	00:00:02	00:01:36
minusMillis	31	00:00:04	02:52:04
minusMinutes	37	00:00:03	01:51:51
minusPeriodHours	48	00:00:02	00:01:36
minusPeriodMillis	31	00:00:04	02:52:04
minusPeriodMinutes	37	00:00:03	01:51:51
minusPeriodSeconds	37	00:00:03	01:51:51
minusSeconds	36	00:00:03	02:01:48
plusHours	48	00:00:02	00:01:36
plusMillis	31	00:00:03	02:51:33
plusMinutes	39	00:00:03	01:31:57
plusPeriodHours	48	00:00:02	00:01:36
plusPeriodMillis	30	00:00:03	03:01:30
plusPeriodMinutes	35	00:00:03	02:11:45
plusPeriodSeconds	35	00:00:03	02:11:45
plusSeconds	31	00:00:03	02:51:33
withHourOfDay	48	00:00:02	00:01:36
withMillisOfSecond	48	00:00:02	00:01:36
withMinuteOfHour	48	00:00:02	00:01:36
withSecondOfMinute	48	00:00:02	00:01:36
getHourOfDay	48	00:00:02	00:01:36
getMillisOfSecond	48	00:00:02	00:01:36
getMinuteOfHour	48	00:00:02	00:01:36
getSecondOfMinute	48	00:00:02	00:01:36

Cuadro 5.22: Caso de estudio: TimeOfDay - 24 bits

Método	TimeOfDay: 48		
	SAT		
	#wa	Tiempo prom.	Tiempo total
minusHours	48	00:00:03	00:02:24
minusMillis	48	00:00:07	00:05:36
minusMinutes	48	00:00:07	00:05:36
minusPeriodHours	48	00:00:03	00:02:24
minusPeriodMillis	48	00:00:07	00:05:36
minusPeriodMinutes	48	00:00:06	00:04:48
minusPeriodSeconds	48	00:00:05	00:04:00
minusSeconds	48	00:00:04	00:03:12
plusHours	48	00:00:03	00:02:24
plusMillis	48	00:00:06	00:04:48
plusMinutes	48	00:00:04	00:03:12
plusPeriodHours	48	00:00:03	00:02:24
plusPeriodMillis	48	00:00:06	00:04:48
plusPeriodMinutes	48	00:00:04	00:03:12
plusPeriodSeconds	48	00:00:04	00:03:12
plusSeconds	48	00:00:04	00:03:12
withHourOfDay	48	00:00:03	00:02:24
withMillisOfSecond	48	00:00:03	00:02:24
withMinuteOfHour	48	00:00:03	00:02:24
withSecondOfMinute	48	00:00:03	00:02:24
getHourOfDay	48	00:00:03	00:02:24
getMillisOfSecond	48	00:00:03	00:02:24
getMinuteOfHour	48	00:00:03	00:02:24
getSecondOfMinute	48	00:00:03	00:02:24

Cuadro 5.23: Caso de estudio: TimeOfDay - 32 bits

Del mismo modo observado al evaluar técnica 1, a medida que se incrementa el ancho de bits de representación se obtiene mayor cantidad de *workarounds transitorios* para cada método incrementándose los tiempos promedio de cómputo. La cantidad de estados promedio computados para 16 bits es de 30.5, para 24 es de 41, y de 48 para 32 bits. Con respecto a los tiempos de cómputo promedio, se observa un incremento al aumentar la cantidad de bits de representación. El tiempo promedio general para 16 bits fue de 1.6 segundos, de 2,58 segundos para 24 bits y de 4,1 segundos para 32 bits.

Un tamaño de 32 bits para la representación de los enteros es lo más conveniente para este caso permitiendo computar estados de recuperación para la totalidad de los estados evaluados.

Capítulo 6

Trabajos relacionados

En este capítulo se describe brevemente una serie de trabajos relevantes a las técnicas propuestas en esta tesis.

6.1. Redundancia en el software

El concepto de redundancia en el software está estrechamente vinculado con el de *workaround*. La redundancia en el software ha sido central en sistemas tolerantes a fallas [58, 77] y de computación autónoma [69]. En sus comienzos el concepto de redundancia en sistemas de información fue asociado con la duplicación de dispositivos físicos a fin de implementar mecanismos ágiles para dar solución a eventuales errores en el hardware. Del mismo modo comenzó a aplicarse en el software, en este caso la idea subyacente fue la de incorporar código adicional a fin de evitar fallas.

Se describen a continuación algunas de las primeras técnicas que utilizan de base el concepto de redundancia para la recuperación de sistemas ante la ocurrencia de fallas.

N-Versiones [22] es una de las primeras técnicas utilizadas en sistemas tolerantes a fallas. La misma consiste en producir una cantidad N de versiones de un módulo de software desarrollados de manera independiente. Esto brindará la posibilidad de que ante una eventual falla pueda utilizarse otra implementación del módulo para restablecer el sistema. Las diferentes implementaciones son ejecutadas de forma concurrente, alternando entre ellas cuando algún problema es detectado.

Micro-reboots [19, 85] es una técnica de recuperación que consiste en re-

iniciar un componente en caso de que se produzca una falla en el mismo. Para esto, se usan los procedimientos de inicialización definidos para el componente. El reinicio de componentes busca obtener los beneficios de no tener que ejecutar el sistema completo nuevamente, lo que reduce el tiempo de recuperación.

Recovery-blocks [86] utiliza múltiples versiones de un módulo de software. A diferencia de la técnica de N-versiones en este caso las diferentes implementaciones son ejecutadas de manera secuencial. Si una falla es detectada, otra implementación del mismo módulo es seleccionada para ejecutar la operación.

Self-checking programming [90] combina la técnica de N-Versiones y Recovery-blocks. Cada componente posee al menos dos implementaciones independientes que serán ejecutadas en paralelo. Si uno de los componentes falla, otro es seleccionado como activo para continuar la ejecución.

6.2. *Workarounds* para aplicaciones Web

A diferencia de las técnicas anteriores los workarounds no requieren la provisión de código adicional, sino que su objetivo es aprovechar la redundancia intrínseca del software para recuperar sistemas en tiempo de ejecución.

Las aplicaciones web es un dominio en donde se han utilizado exitosamente *workarounds automáticos* [16]. Por su naturaleza estas aplicaciones poseen mecanismos de implementación y comunicación muy favorables para su recuperación a través de *workarounds*. Las aplicaciones web son construidas bajo una arquitectura de capas. Esto define por lo general un diseño que consta de una capa de presentación (con la interfaz de usuario), una capa con la lógica de la aplicación (programas) y, finalmente, una capa para la administración de los datos (gestor de base de datos). La comunicación entre estas dos últimas se realiza a través de transacciones de forma tal que las operaciones son confirmadas (*commit*) o canceladas (*rollback*), lo que permite mantener consistente el estado del sistema en caso de producirse una falla. Por otro lado, esta arquitectura posibilita la supervisión de la comunicación entre las capas, permitiendo la intersección de una falla y la ejecución posterior de un *workaround* en su lugar.

Las fallas en este contexto podrán ser detectadas por el servidor y por el usuario de la aplicación web quien podrá reportar un error a través de un botón o enlace insertado en la interfaz gráfica. Por ejemplo, si el usuario ha

intentado consultar el saldo de su cuenta bancaria y no ha obtenido respuesta, podrá solicitar un *workaround*.

Para modelar *workarounds* en aplicaciones web se han presentado dos formalismos. En primer lugar se plantea un enfoque que consiste en computar los *workarounds* sobre un modelo abstracto de la aplicación a través de una máquina de estados finitos [82]. La máquina de estados es provista por el usuario y el cómputo consiste en encontrar caminos alternativos para ir de un estado inicial al estado final de la acción a recuperar. Esta propuesta fue evaluada experimentalmente en dos aplicaciones web de uso masivo. Flickr ¹ una plataforma que permite almacenar y compartir fotografías y videos, y Google Maps ² un servidor mundial de mapas. Dadas las máquinas de estado con el comportamiento de las aplicaciones mencionadas, se logró mantener funcionando las aplicaciones siendo éstas sometidas a fallas ya conocidas.

El segundo formalismo presentado en [16] que aplica *workarounds* sobre aplicaciones web realiza su cómputo a través de un conjunto de reglas de reescritura de términos. Cada regla de reescritura define una equivalencia entre dos secuencias de rutinas de una aplicación (respecto de sus especificaciones). De esta manera, la aplicación de una regla de reescritura permitirá sustituir en el código fuente, la invocación de la rutina por una secuencia de rutinas equivalente. La búsqueda de *workarounds* se reduce a la aplicación de reglas de reescritura hasta lograr (o no) la recuperación de una falla en tiempo de ejecución. Esta propuesta fue implementada en un prototipo denominado *RAW (Runtime Automatic Workarounds)* [17] y evaluada experimentalmente para las APIs de Google Maps y Youtube³. Las fallas fueron extraídas de repositorios de bugs y foros. Del total de las fallas recopiladas para cada API lograron encontrarse *workarounds* en un 10 % de los casos para Google Maps y en un 42 % de los casos para Youtube.

A diferencia de las especificaciones requeridas por las técnicas aplicables a sistemas web descritas anteriormente (máquinas de estado, sistemas de reescritura), las técnicas propuestas en esta tesis requieren de especificaciones de contratos en JML. Si bien definir contratos requiere un trabajo adicional por parte de los programadores, en general estos están más familiarizados con este tipo de especificaciones. Además, la definición de contratos tiene muchos otros beneficios adicionales en el desarrollo de software. Por ejemplo,

¹<https://www.flickr.com/>

²<https://www.google.com.ar/maps>

³<https://www.youtube.com/>

permiten detectar fallas en tiempo de ejecución, ayudan a detectar más fallas durante el testing, habilitan el uso de herramientas automáticas de testing [33] y de verificación [38], etc.

Además, en nuestros enfoques el mecanismo de búsqueda se realiza a través de técnicas basadas en los contratos y SAT solving, mientras que las otras usan algoritmos de búsqueda de caminos en grafos o bien hacen uso de sistemas de reescritura de términos.

La aplicación de workarounds en Java difiere significativamente de la aplicación de workarounds en aplicaciones web. Por un lado, como se explicó anteriormente en las aplicaciones web el usuario cumple la función de oráculo para la detección de fallas, mientras que en las aplicaciones Java se usan excepciones y/o los contratos de los métodos, si están disponibles (como se explicó en la sección 2.7). Por otro lado, la aplicación de workarounds en sistemas web consiste simplemente en invocar una serie de servicios diferentes provistos por la API del sistema. En cambio, para la técnica 1 propuesta en este trabajo la aplicación de un workaround involucra hacer un rollback a un estado inicial y ejecutar el workaround computado para producir el estado esperado (también explicado en la sección 2.7. Para la técnica 2, aplicar un workaround consiste en cambiar el estado de la Máquina Virtual de Java (JVM) por el estado computado por la técnica.

6.3. Secuencias equivalentes

Vinculado con la construcción de alternativas de ejecución a un método dado, otros autores presentaron una técnica de cómputo automático de secuencias equivalentes [45]. Esta propuesta hace uso de algoritmos genéticos. Este enfoque toma como entrada una rutina r y una suite de tests (conjunto de escenarios iniciales). En una primera etapa, el enfoque busca computar una secuencia de métodos candidata que se comporte igual que r en dichos escenarios (tests). Si se logra encontrar una secuencia candidata, en una segunda etapa se busca generar un escenario (test) donde la ejecución de esta secuencia y la rutina r se comporten de manera diferente. En tal caso la secuencia candidata es descartada y el escenario es incorporado al conjunto de escenarios disponibles. Luego, la búsqueda se reinicia.

Nótese que tanto para la búsqueda como para la eliminación de los candidatos se ejecuta el código de las rutinas. Es decir, las secuencias retornadas son equivalentes a r con respecto a la implementación (y a escenarios gene-

rados) y no respecto a su especificación. Esto trae como consecuencia que si existe un defecto en r , las secuencias equivalentes computadas por el enfoque imitarán el comportamiento erróneo, por lo que no serían de utilidad como *workarounds* para recuperar fallas. Otra posibilidad es que debido a un defecto en r , el enfoque no pueda encontrar secuencias de métodos que sí son equivalentes a r según las especificaciones. Si bien es cierto que esta técnica comparte con las propuestas de esta tesis el aprovechamiento de redundancia intrínseca en el software, por los motivos mencionados, creemos que este enfoque es más apropiado para otros tipos de aplicaciones como por ejemplo, para el descubrimiento de aserciones para testing [45].

En la sección 5.2 se realizó un análisis comparativo entre la técnica 1 de cómputo de *workarounds* propuesta en esa tesis y el trabajo original de cómputo de secuencias equivalentes descrito en [45]. Como se mencionó anteriormente en la evaluación experimental una gran cantidad de métodos no posee alternativas de recuperación permanentes y si transitorias.

6.4. Reparación de estructuras de datos

Otros trabajos [55,91,92] relevantes a esta tesis plantean la reparación de estructuras de datos complejas (árboles binarios, listas simplemente encadenadas, listas doblemente encadenadas, etc). Estos trabajos coinciden con las técnicas propuestas en esta tesis con la necesidad de especificaciones formales: precondiciones, postcondiciones e invariantes de representación, y con el objetivo de generar estructuras de datos que permitan restablecer la ejecución fallida de un método.

En [55] se propone un mecanismo de reparación de estructuras de datos que requiere de un invariante de clase, definido a través de un predicado imperativo **repOK**. Este predicado establece las restricciones de integridad que deben cumplir las estructuras. Por ejemplo la propiedad de aciclicidad para listas simplemente encadenadas. El procedimiento de reparación se aplica al detectar una instancia que no cumple con el **repOK**, luego usa técnicas de búsqueda (exhaustiva acotada para ser más precisos, como en la herramienta de generación [11]) hasta encontrar una estructura que satisfaga el **repOK**.

En [91, 92] los autores proponen una alternativa de reparación de estructuras de datos donde se utilizan especificaciones Alloy (precondiciones, postcondiciones e invariantes de representación) y SAT solving para computar nuevas instancias que verifiquen la especificación del método fallido. Este

procedimiento de reparación hace uso del invariante de representación de la clase y de las postcondiciones de los métodos para detectar fallas. Luego se utilizan las especificaciones de la estructura junto a la traza de ejecución del método fallido para intentar reparar la estructura. Para obtener la traza de ejecución, previamente el código es instrumentado para registrar aquellos campos que son leídos o escritos en la traza de ejecución del método fallido. El proceso de reparación aplica SAT solving en varias etapas. En un principio se intentará construir una estructura que satisfaga la postcondición y el invariante de representación permitiendo al SAT solver modificar únicamente los campos de la estructura que han sido leídos en la traza. Si no se logra generar una estructura que cumpla con las condiciones requeridas, se aplicará SAT solving nuevamente, esta vez permitiendo modificar los campos que hayan sido leídos y escritos en la traza. Finalmente, si no se puede generar la estructura de reparación buscada se permitirá al SAT solver modificar cualquier campo de la estructura. Este método de reparación en etapas reduce los tiempos de solving y acelera la reparación de estructuras, ya que durante las primeras dos etapas una parte de la estructura se encuentra fija y el SAT solver sólo tiene que considerar las posibilidades para los campos accedidos (leídos en la primera etapa, leídos y escritos en la segunda etapa).

La utilización de SAT solving como mecanismo de cómputo trae consigo una limitación vinculada al tamaño de las estructuras, que en general son relativamente pequeñas debido a las dificultades de escalabilidad de esta técnica. En [91, 92] se busca mejorar la escalabilidad fijando campos de la estructura y reduciendo así el espacio de búsqueda del SAT solver. Sin embargo, en estos trabajos los beneficios obtenidos dependen fuertemente de la estructura bajo análisis y de cómo cambia la misma al ejecutar sus métodos. Por ejemplo, en estructuras tales como árboles rojo y negros donde las operaciones de inserción y eliminación pueden requerir re-balancear la estructura, todos los campos de la estructura resultan modificados por el método. En estos casos, las primeras etapas no ayudan a resolver el problema de reparación más eficientemente (ya que hay que modificar todos los campos de la estructura desde el inicio).

En cambio, en la técnica 2 propuesta en esta tesis, la contribución principal es la incorporación de predicados de rotura de simetrías y cotas ajustas para reducir el espacio de búsqueda de SAT. Estas optimizaciones son más generales, ya que sus beneficios se obtienen independientemente de los accesos a campos realizados por el método fallido. Por otra parte, estas técnicas pueden ser complementarias, es decir, es posible desarrollar un enfoque que

combine las técnicas de [91, 92] con las optimizaciones de técnica 2 para obtener los beneficios de ambas técnicas (reducción del espacio de búsqueda por campos accedidos, mejora de eficiencia de solving por cotas ajustadas y rotura de simetrías).

6.5. Análisis de código basado de SAT solving

La aplicación eficiente del análisis basado en SAT solving es fundamental para el éxito de las técnicas propuestas en esta tesis. La incorporación de predicados de rotura de simetría y cotas ajustadas realizan un gran aporte a la eficiencia de la propuesta. Se describen brevemente a continuación algunos de los trabajos vinculados más relevantes.

JForge [27], es una herramienta de verificación acotada basada en SAT para chequear automáticamente programas definidos en Java respecto de especificaciones JML. JForge traduce el programa y su especificación a Alloy, que luego es usado como procedimiento de decisión. En [48] y [89] se presentan F-Soft y Saturn, ambas herramientas realizan análisis basados en SAT a fin de detectar fallas en programas.

Al igual que JForge, TACO (Translation of Annotated COde) [38, 40] realiza verificación acotada de código Java anotado con especificaciones JML y basada en SAT. TACO traduce en primer lugar, los contratos JML incluidos en el programa a predicados Alloy, por otro lado se traducen los métodos de la clase a programas DynAlloy. Luego ambas especificaciones son integradas en un único modelo DynAlloy. El procedimiento continúa traduciendo el modelo DynAlloy a Alloy, y posteriormente al lenguaje intermedio KodKod [84] sobre el que se aplica SAT solving. TACO incorpora por primera vez predicados de rotura de simetría y cotas ajustadas a la verificación exhaustiva acotada de programas basada en SAT, y muestra que estas optimizaciones resultan en una mejora significativa de los tiempos de ejecución y la escalabilidad de la verificación [38, 40]. En este trabajo, las optimizaciones mencionadas se aplicaron a un problema diferente, el cómputo de workarounds transitorios. Además, TACO presenta el primer algoritmo eficiente de cómputo de cotas ajustadas [40].

Las técnicas propuestas en esta tesis han adoptado de TACO el procedimiento de traducción de las especificaciones JML a fórmulas Alloy. Tanto JForge como las demás herramientas mencionadas no hacen uso de cotas ajustadas en su proceso de verificación, es por ello que su rendimiento y

escalabilidad son inferiores los de TACO [38].

En [74] se presentan dos enfoques para el cómputo eficientemente de cotas ajustadas en entornos de ejecución secuenciales. El primero de ellos, Bottom-up [72] fue usado para el cómputo de cotas ajustadas para la técnica 2 propuesta, y fue explicado en la sección 4.3. El segundo enfoque SLBD [73] se basa en especificaciones de invariantes de representación de estructuras en términos de predicados inductivos de la lógica de separación [31].

En [70] se presenta un análisis basado en *dataflow* para propagar cotas ajustadas del estado inicial al resto de los estados del programa. Como resultado de esta propagación se obtiene una mejora significativa en la eficiencia de la verificación acotada de código.

En [2] se presenta una técnica que combina análisis incremental de SAT y cotas ajustadas para la generación automática de casos de test. La técnica es implementada por la herramienta FAJITA (una adaptación de TACO).

Por último, es importante mencionar que en la literatura pueden encontrarse numerosos ejemplos de análisis automáticos que utilizan rotura de simetrías para beneficiarse de una cantidad reducida de instancias isomorfas durante el análisis. Por ejemplo en [47, 66] se canoniza el *heap* en el contexto de *model checking*. Por otro lado, otros enfoques rompen simetrías para acelerar la generación automática de casos de test, tanto de caja negra [11] como de caja blanca [43, 88].

Capítulo 7

Conclusiones y trabajos futuros

La complejidad intrínseca del software, la constante adaptación/extensión que sufre el software junto a otros factores, hacen muy difícil producir sistemas que mantengan niveles elevados de calidad durante toda su vida útil. Esta situación combinada con presiones cada vez más fuertes para proveer una disponibilidad constante en el software, hace que las técnicas que ayudan a los sistemas a tolerar fallas relacionadas con errores sean muy relevantes.

En esta tesis se han propuesto dos técnicas que contribuyen a tolerar fallas relacionadas a errores que ocurren en tiempo de ejecución. Estas técnicas proponen el uso de análisis basado en SAT para computar automáticamente *workarounds transitorios*, a partir de especificaciones formales, más precisamente, invariantes de representación de estructuras y pre y postcondiciones para métodos, en el lenguaje declarativo JML. Estas especificaciones son la forma más común de describir comportamiento del código fuente, a diferencia de las representaciones utilizadas por trabajos relacionados, que requiere proveer especificaciones de más alto nivel (máquinas de estado, reglas de reescritura). Es decir, nuestras técnicas aprovechan de manera automática las especificaciones del programa a nivel de detalle de código fuente, a diferencia de las alternativas que requieren que el ingeniero genere manualmente abstracciones de programas de máquinas de estados de alto nivel.

Por otro lado, y a diferencia de los trabajos existentes, los *workarounds* computados por nuestras técnicas son específicos para un estado, denominados transitorios, en el sentido de que aprovechan el estado en que se ejecutó la rutina fallida durante la búsqueda.

Las dos técnicas utilizan análisis basado en SAT para el cómputo de *workarounds transitorios*. Para mejorar el rendimiento de éstas técnicas y a

sabiendas de las limitaciones vinculadas al análisis de SAT, se propusieron optimizaciones al método de cómputo. Para la primera propuesta se computaron esquemas de *workarounds* obteniendo una importante aceleración en los tiempos de recuperación. Para la segunda técnica se incorporaron predicados de rotura de simetrías y cotas ajustadas. Estos dos conceptos ya han sido utilizados previamente para la verificación de software aunque nunca habían sido aplicados para computar *workarounds transitorios*.

Las técnicas propuestas pueden integrarse con la arquitectura de recuperación propuesta en ARMOR [15] proporcionando un mecanismo de cómputo de *workarounds transitorios* aplicable en tiempo de ejecución.

Nuestras técnicas fueron sometidas a una evaluación experimental que involucró varias implementaciones equipadas con contratos (incluidas las de aritmética intensiva). Esta evaluación demostró que las técnicas pueden computar *workarounds transitorios* a partir de contratos, que son útiles para recuperar la invocación de un método fallido en tiempo de ejecución, en tiempos de ejecución aceptables para esta tarea, ya sea mediante la generación de una secuencia de métodos alternativa (técnica 1) o la generación de un estado de recuperación (técnica 2). Los experimentos también muestran que nuestras técnicas pueden computar *workarounds transitorios* en muchas situaciones en las que no pueden computarse soluciones alternativas basadas en *workarounds permanentes*, lo que resalta la importancia de las técnicas propuestas.

El trabajo realizado en esta tesis conduce a varias líneas de trabajo posterior. Los *workarounds transitorios* permiten recuperar la invocación fallida de un método en una situación específica. Sin embargo, muchos de los *workarounds transitorios* son de hecho instancias de *workarounds permanentes*, es decir, pueden generalizarse para producir soluciones independientes del estado inicial. La noción de esquema de *workarounds* es un paso en la dirección de producir automáticamente generalizaciones a partir de soluciones transitorias, pero como mencionamos anteriormente, no constituyen soluciones permanentes per se (la instanciación de variables de esquema aún exige llamadas al SAT). Como trabajo futuro se plantea el estudio de mecanismos de promoción de esquemas de *workaround* a *workarounds permanentes*.

Las técnicas propuestas en esta tesis requieren de contratos para definir el comportamiento de los métodos involucrados. En este trabajo, se definieron contratos que determinan el comportamiento preciso de los métodos involucrados. Un estudio interesante es evaluar la efectividad de nuestras técnicas aplicadas en casos de estudio donde los contratos definan parcialmente los

comportamientos de los métodos. Por ejemplo, que luego de insertar en una lista, el tamaño se incrementa en uno, sin mencionar como quedan los elementos de la lista luego de la inserción. Una hipótesis en este sentido es que, dado qué técnica 1 computa secuencias de métodos equivalentes, y luego estas secuencias se aplican en tiempo de ejecución, la mayoría de los workarounds computados por esta técnica serían útiles para recuperar al software en tiempo de ejecución. Como trabajo futuro se planea evaluar experimentalmente esta hipótesis.

A nivel de implementación de las técnicas, se prevee su integración a la arquitectura de ARMOR para desarrollar una herramienta usable de recuperación de fallas en tiempo de ejecución mediante workarounds. Esto permitiría además, realizar una evaluación de las técnicas en un conjunto más grande de casos de estudio.

Por otro lado, es de nuestro interés realizar el estudio y análisis de factibilidad del uso de EPAs (*Enabledness Preserving Abstractions*) [25] para el cómputo de *workarounds*. Las EPAs son máquinas de estados finitos que definen un protocolo ejecución para los métodos de una clase. Las EPAs son generadas de forma automática y especifican cuando un método está o no habilitado en un estado (abstracto) determinado. Resulta de interés estudiar si es posible definir un mecanismo de cómputo de workarounds a partir de las especificaciones provistas por las EPAs.

Bibliografía

- [1] *Replication package for Automated Workarounds from Java Program Specifications based on SAT solving*, <http://dc.exa.unrc.edu.ar/staff/naguirre/sat-workarounds/>. 5
- [2] Pablo Abad, Nazareno Aguirre, Valeria Bengolea, Daniel Ciolek, Marcelo F. Frias, Juan Galeotti, Tom Maibaum, Mariano Moscato, Nicolas Rosner, and Ignacio Vissani, *Improving Test Generation under Rich Contracts by Tight Bounds and Incremental SAT solving*, 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation (Los Alamitos, CA, USA), ICST'13, IEEE Computer Society, 2013, pp. 21–30. 1, 2.3, 3.2.1, 4, 6.5
- [3] Raul Alborodo, Nicolás Ricci, Galeotti J.P., and Aguirre N., *Análisis modular y recuperación de contraejemplos en TACO*, XVII Congreso Argentino de Ciencias de la Computación (2011). 3.2, 3.4, 3.5
- [4] Carzaniga Antonio, Gorla Alessandra, and Pezzè Mauro, *Self-healing by means of automatic workarounds*, 2008 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2008, Leipzig, Germany, May 12-13, 2008, 2008, pp. 17–24. 1.1.1, 1.2, 2.6, 26
- [5] Isabelle Attali and Thomas P. Jensen (eds.), *Java on Smart Cards: Programming and Security, First International Workshop, JavaCard 2000, Cannes, France, September 14, 2000, Revised Papers*, Lecture Notes in Computer Science, vol. 2041, Springer, 2001. 2.4
- [6] Gilles Barthe, Lilian Burdy, Julien Charles, Benjamin Grégoire, Marieke Huisman, Jean-Louis Lanet, Mariela Pavlova, and Antoine Requet, *JACK - A tool for Validation of Security and Behaviour of Java Applications*, Formal Methods for Components and Objects, 5th International

-
- Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures (Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, eds.), Lecture Notes in Computer Science, vol. 4709, Springer, 2006, pp. 152–174. 2.4
- [7] Rudolf Bayer, *Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms*, Acta Informatica **1** (1972), 290–306. 5
- [8] Jason Belt, Robby, and Xianghua Deng, *Sireum/Topi LDP: a lightweight semi-decision procedure for optimizing symbolic execution-based analyses*, Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (New York, NY, USA), ESEC/FSE '09, ACM, 2009, pp. 355–364. 1, 5
- [9] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu, *Bounded model checking*, Advances in Computers **58** (2003), 117–148. 1
- [10] Jan Boerman, Marieke Huisman, and Sebastiaan Jozef Christiaan Joosten, *Reasoning About JML: Differences Between KeY and OpenJML*, IFM 2018, LNCS (Carlo A. Furia and Kirsten Winter, eds.), vol. 11023, 9 2018, pp. 1–17 (English). 2.4
- [11] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov, *Korat: Automated Testing Based on Java Predicates*, ISSTA '02, Association for Computing Machinery, 2002, pp. 123–133. 1, 6.4, 6.5
- [12] British Broadcasting Corporation News (1BBC News), *Facebook, Instagram and Whatsapp suffer outages.*, <https://www.bbc.com/news/technology-47927714>, Published: 14.04.2019, Accessed: 01.03.2020. 1
- [13] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll, *An overview of JML tools and applications*, Int. J. Softw. Tools Technol. Transf. **7** (2005), no. 3, 212–232. 2.4
- [14] Antonio Carzaniga, Alberto Goffi, Alessandra Gorla, Andrea Mattavelli, Nicolò Perino, Mauro Pezzè, and Paolo Tonella, *Intrinsic software*

- redundancy for self-healing software systems, automated oracle generation*, Software Engineering & Management 2015, Multikonferenz der GI-Fachbereiche Softwaretechnik (SWT) und Wirtschaftsinformatik (WI), FA WI-MAW, 17. März - 20. März 2015, Dresden, Germany (Uwe Abmann, Birgit Demuth, Thorsten Spitta, Georg Püschel, and Ronny Kaiser, eds.), LNI, vol. P-239, GI, 2015, pp. 129–130. 1.2, 2.6
- [15] Antonio Carzaniga, Alessandra Gorla, Andrea Mattavelli, Nicolò Perino, and Mauro Pezzè, *Automatic recovery from runtime failures*, 35th International Conference on Software Engineering, ICSE 2013, San Francisco, CA, USA, May 18-26, 2013 (David Notkin, Betty H. C. Cheng, and Klaus Pohl, eds.), IEEE Computer Society, 2013, pp. 782–791. 1, 1.1.1, 1.2, 2.7, 3.4, 4, 7
- [16] Antonio Carzaniga, Alessandra Gorla, Nicolò Perino, and Mauro Pezzè, *Automatic workarounds for web applications*, Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010, ACM, 2010, pp. 237–246. 6.2
- [17] Antonio Carzaniga, Alessandra Gorla, Nicolò Perino, and Mauro Pezzè, *RAW: runtime automatic workarounds*, Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE 2010, Cape Town, South Africa, 1-8 May 2010 (Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel, eds.), ACM, 2010, pp. 321–322. 1.1.1, 26, 6.2
- [18] Antonio Carzaniga, Alessandra Gorla, Nicolò Perino, and Mauro Pezzè, *Automatic Workarounds: Exploiting the Intrinsic Redundancy of Web Applications*, ACM Trans. Softw. Eng. Methodol. **24** (2015), no. 3, 16:1–16:42. 1.1.1, 2.6
- [19] Antonio Carzaniga, Alessandra Gorla, and Mauro Pezzè, *Handling Software Faults with Redundancy*, Architecting Dependable Systems VI (Rogério de Lemos, Jean-Charles Fabre, Cristina Gacek, Fabio Gadducci, and Maurice H. ter Beek, eds.), Lecture Notes in Computer Science, vol. 5835, Springer, 2008, pp. 148–171. 6.1
- [20] 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. 1.1.2, 2.4
- [21] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll, *Beyond assertions: advanced specification and verification with JML and ESC/Java2*, Proceedings of the 4th international conference on Formal Methods for Components and Objects (Berlin, Heidelberg), FMCO'05, Springer-Verlag, 2006, pp. 342–363. 1
- [22] Liming Chen and Algirdas Avizienis, *N-version programming: A fault-tolerance approach to reliability of software operation*, Proc. 8th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-8), vol. 1, 1978, pp. 3–9. 6.1
- [23] Yoonsik Cheon and Gary T. Leavens, *A runtime assertion checker for the Java Modeling Language (JML)*, PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING RESEARCH AND PRACTICE (SERP '02), LAS VEGAS, CSREA Press, 2002, pp. 322–328. 2.4
- [24] David R. Cok and Joseph Kiniry, *ESC/Java2: Uniting ESC/Java and JML*, Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, International Workshop, CASSIS 2004, Marseille, France, March 10-14, 2004, Revised Selected Papers (Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, eds.), Lecture Notes in Computer Science, vol. 3362, Springer, 2004, pp. 108–128. 2.4
- [25] Guido de Caso, Víctor A. Braberman, Diego Garbervetsky, and Sebastián Uchitel, *Enabledness-based program abstractions for behavior validation*, ACM Trans. Softw. Eng. Methodol. **22** (2013), no. 3, 25:1–25:46. 7
- [26] Leonardo De Moura and Nikolaj Bjørner, *Z3: An Efficient SMT Solver*, Proceedings of the Theory and Practice of Software, 14th International

-
- Conference on Tools and Algorithms for the Construction and Analysis of Systems (Berlin, Heidelberg), TACAS'08/ETAPS'08, Springer-Verlag, 2008, pp. 337–340. 1
- [27] Greg Dennis, Felix Sheng-Ho Chang, and Daniel Jackson, *Modular verification of code with SAT*, Proceedings of the 2006 international symposium on Software testing and analysis (New York, NY, USA), ISSTA '06, ACM, 2006, pp. 109–120. 1, 2.1, 6.5
- [28] Greg Dennis, Kuat Yessenov, and Daniel Jackson, *Bounded Verification of Voting Software*, Verified Software: Theories, Tools, Experiments (Berlin, Heidelberg) (Natarajan Shankar and Jim Woodcock, eds.), Springer Berlin Heidelberg, 2008, pp. 130–145. 2.5
- [29] Jo Devriendt, Bart Bogaerts, Maurice Bruynooghe, and Marc Denecker, *Improved Static Symmetry Breaking for SAT*, Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings (Nadia Creignou and Daniel Le Berre, eds.), Lecture Notes in Computer Science, vol. 9710, Springer, 2016, pp. 104–122. 3.1
- [30] Edsger W. Dijkstra and Carel S. Scholten, *Predicate calculus and program semantics*, Springer-Verlag New York, Inc., New York, NY, USA, 1990. 1
- [31] Dino Distefano, Peter W. O'Hearn, and Hongseok Yang, *A Local Shape Analysis Based on Separation Logic*, Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (Berlin, Heidelberg), TACAS'06, Springer-Verlag, 2006, pp. 287–302. 1, 6.5
- [32] Mark Dowson, *The Ariane 5 Software Failure*, SIGSOFT Software Engineering Notes **22** (1997), no. 2, 84. 1
- [33] Bouquet Fabrice, *Automated Boundary Test Generation from JML Specifications*, 2006, pp. 428–443. 6.2
- [34] Ira R. Forman, Nate Forman, Dr. John Vlissides Ibm, Ira R. Forman, and Nate Forman, *Java Reflection in Action*, 2004. 3.1, 3.1

-
- [35] Gordon Fraser and Andrea Arcuri, *Evosuite: Automatic test suite generation for object-oriented software*, 09 2011, pp. 416–419. 1.1.1
- [36] Marcelo F. Frias, Juan P. Galeotti, Carlos G. López Pombo, and Nazareno M. Aguirre, *Dynalloy: upgrading alloy with actions*, Proceedings of the 27th international conference on Software engineering (New York, NY, USA), ICSE '05, ACM, 2005, pp. 442–451. 1.1.3, 2.3
- [37] Marcelo F. Frias, Carlos G. Lopez Pombo, Juan P. Galeotti, and Nazareno M. Aguirre, *Efficient Analysis of Dynalloy Specifications*, ACM Trans. Softw. Eng. Methodol. **17** (2007), no. 1, 4:1–4:34. 1.1.3, 2.3, 2.3
- [38] Juan P. Galeotti, Nicolas Rosner, Carlos G. Lopez Pombo, and Marcelo F. Frias, *TACO: Efficient SAT-Based Bounded Verification Using Symmetry Breaking and Tight bounds*, IEEE Transactions on Software Engineering **99** (2013), no. PrePrints, 1. 1.1.3, 1.2, 2.3, 3.1, 4, 6.2, 6.5
- [39] Juan Pablo Galeotti, *Verificación de software usando alloy*, Ph.D. thesis, Universidad de Buenos Aires. Argentina, 2010. 3.2, 3.4, 4.2, 4.2.1, 4.2.1, 4.2.1
- [40] Juan Pablo Galeotti, Nicolás Rosner, Carlos Gustavo López Pombo, and Marcelo Fabian Frias, *Analysis of invariants for efficient bounded verification*, Proceedings of the 19th international symposium on Software testing and analysis (New York, NY, USA), ISSTA '10, ACM, 2010, pp. 25–36. 1.2, 2.3, 4.2, 6.5
- [41] David Garlan and Bradley Schmerl, *Model-Based Adaptation for Self-Healing Systems*, Proceedings of the First Workshop on Self-Healing Systems (New York, NY, USA), WOSS '02, Association for Computing Machinery, 2002, p. 27–32. 1.1
- [42] Carlo Ghezzi, *Of software and change*, Journal of Software: Evolution and Process **29** (2017), no. 9. 1
- [43] 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 (New York, NY, USA), ICSE '10, ACM, 2010, pp. 225–234. 6.5

-
- [44] Alberto Goffi, Alessandra Gorla, Andrea Mattavelli, and Mauro Pezzè, *Intrinsic Redundancy for Reliability and Beyond*, Present and Ulterior Software Engineering (Manuel Mazzara and Bertrand Meyer, eds.), Springer, 2017, pp. 153–171. 1.2, 2.6
- [45] Alberto Goffi, Alessandra Gorla, Andrea Mattavelli, Mauro Pezzè, and Paolo Tonella, *Search-based synthesis of equivalent method sequences*, Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014 (Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey, eds.), ACM, 2014, pp. 366–376. 1.1.1, 26, 5.2, 5.2, 5.2, 6.3
- [46] Marieke Huisman, *Reasoning about Java programs in higher order logic using PVS and Isabelle*, IPA dissertation series, vol. 03, IPA, 2001. 2.4
- [47] Radu Iosif, *Symmetry Reduction Criteria for Software Model Checking*, Proceedings of the 9th International SPIN Workshop on Model Checking of Software (London, UK, UK), Springer-Verlag, 2002, pp. 22–41. 6.5
- [48] Franco Ivančić, Zijiang Yang, Malay K. Ganai, Aarti Gupta, Ilya Shlyakhter, and Pranav Ashar, *F-SOFT: software verification platform*, Proceedings of the 17th international conference on Computer Aided Verification (Berlin, Heidelberg), CAV'05, Springer-Verlag, 2005, pp. 301–306. 6.5
- [49] Daniel Jackson, *Software Abstractions - Logic, Language, and Analysis*, MIT Press, 2006. 1, 1.1.3, 2.2
- [50] Daniel Jackson, *Software Abstractions: logic, language, and analysis*, MIT press, 2012. 4.2
- [51] Bart Jacobs, *Weakest Precondition Reasoning for Java Programs with JML Annotations*, Journal of Logic and Algebraic Programming **58** (2002), 61–88. 2.4
- [52] Bart Jacobs and Erik Poll, *A Logic for the Java Modeling Language JML*, Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering (Berlin, Heidelberg), FASE '01, Springer-Verlag, 2001, p. 284–299. 2.4

-
- [53] Bart Jacobs, Joachim van den Berg, Marieke Huisman, Martijn van Berkum, U. Hensel, and H. Tews, *Reasoning about Java Classes: Preliminary Report*, SIGPLAN Not. **33** (1998), no. 10, 329–340. 2.4
- [54] Angelos D Keromytis, *Characterizing Software Self-healing Systems*, International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security, Springer, 2007, pp. 22–33. 2.6
- [55] Sarfraz Khurshid, Iván García, and Yuk Lai Suen, *Repairing Structurally Complex Data*, Model Checking Software, 12th International SPIN Workshop, San Francisco, CA, USA, August 22-24, 2005, Proceedings, pages = 123–138, year = 2005. 1.1.1, 4, 5.3, 6.4
- [56] J. W. Klop, *Chapter 1: Term Rewriting Systems*, Handbook of Logic in Computer Science (S. Abramsky, D. Gabbay, and T. Maibaurn, eds.), Oxford University Press, 1992, pp. 1–116. 1.1.1
- [57] Burdy L., Cheon Y., Cok D. and Ernst M., and Kiniry J. and Leavens G., *An overview of JML tools and applications*, Int. J. Softw. Tools Technol. Transf. **7** (2005), no. 3, 212–232. 1.1.2, 2.4
- [58] J.-C. Laprie, J. Arlat, C. Beounes, and K. Kanoun, *Definition and analysis of hardware- and software-fault-tolerant architectures*, Computer **23** (1990), no. 7, 39–51. 6.1
- [59] Gary Leavens, Albert Baker, and Clyde Ruby, *JML: a Java Modeling Language*, (1998). 1.1.2
- [60] Gary T. Leavens, Joseph R. Kiniry, and Erik Poll, *A JML Tutorial: Modular Specification and Verification of Functional behavior for java*, Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings (Werner Damm and Holger Hermanns, eds.), Lecture Notes in Computer Science, vol. 4590, Springer, 2007, p. 37. 2.4
- [61] Nancy Leveson and Clark Turner, *An Investigation of the Therac-25 Accidents*, Computer **26** (1993), no. 7, 18–41. 1
- [62] Yue Li, Tian Tan, and Jingling Xue, *Understanding and Analyzing Java Reflection*, ACM Trans. Softw. Eng. Methodol. **28** (2019), no. 2. 3.1

-
- [63] Barbara Liskov and John V. Guttag, *Program Development in Java - Abstraction, Specification, and Object-Oriented design*, Addison-Wesley, 2001. 2.4
- [64] Bertrand Meyer, *Eiffel: The Language*, Prentice-Hall, 1991. 1.1.2
- [65] Richard Mitchell, Jim McKim, and Bertrand Meyer, *Design by contract, by example*, Addison Wesley Longman Publishing Co., Inc., USA, 2001. 1.1.2, 2.4
- [66] Madanlal Musuvathi and David L. Dill, *An incremental heap canonicalization algorithm*, Proceedings of the 12th international conference on Model Checking Software (Berlin, Heidelberg), SPIN'05, Springer-Verlag, 2005, pp. 28–42. 6.5
- [67] Galeotti Juan P. and Frias Marcelo F., *Dynalloy as a Formal Method for the Analysis of Java Programs*, Software Engineering Techniques: Design for Quality, SET 2006, October 17-20, 2006, Warsaw, Poland (Krzysztof Sacha, ed.), IFIP, vol. 227, Springer, 2006, pp. 249–260. 2.5
- [68] Carlos Pacheco and Michael D. Ernst, *Randoop: Feedback-directed Random Testing for Java*, OOPSLA 2007 Companion, Montreal, Canada, ACM, October 2007. 1.2, 5
- [69] Manish Parashar and Salim Hariri, *Autonomic computing: An overview*, International workshop on unconventional programming paradigms, Springer, 2004, pp. 257–269. 6.1
- [70] Bruno Cuervo Parrino, Juan Pablo Galeotti, Diego Garbervetsky, and Marcelo F. Frias, *A dataflow analysis to improve SAT-based bounded program verification*, Proceedings of the 9th international conference on Software engineering and formal methods (Berlin, Heidelberg), SEFM'11, Springer-Verlag, 2011, pp. 138–154. 6.5
- [71] Godio Rosner Nicolás Arroyo Marcelo Aguirre Nazareno Frias Marcelo F. Ponzio, Pablo Daniel, *Efficient Bounded Model Checking of Heap-Manipulating Programs using Tight Field Bounds*, Fundamental Approaches to Software Engineering” (Cham) (Esther Guerra and Mariëlle Stoeilinga, eds.), Springer International Publishing, 2021, pp. 218–239. 4.2, 4.3, 4.3

-
- [72] 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. 1.2, 6.5
- [73] Pablo Ponzio, Nicolás Rosner, Nazareno Aguirre, and Marcelo Frias, *Efficient Tight Field Bounds Computation Based on Shape Predicates*, Proceedings of the 19th International Symposium on FM 2014: Formal Methods - Volume 8442 (Berlin, Heidelberg), Springer-Verlag, 2014, pp. 531–546. 6.5
- [74] Pablo Daniel Ponzio, *Cómputo secuencial eficiente de cotas ajustadas, y su impacto en la performance de los análisis de programas basados en SAT*, Ph.D. thesis, Universidad de Buenos Aires. Argentina, 2014. 1.2, 2.1, 4, 4.2, 4.3, 4.3, 4.3.1, 6.5
- [75] Ben Potter, David Till, and Jane Sinclair, *An Introduction to Formal Specification and Z*, 2nd ed., Prentice Hall PTR, USA, 1996. 1
- [76] Roger S. Pressman, *Development Strategies and Project Management*, The Computer Science and Engineering Handbook (Allen B. Tucker, ed.), CRC Press, 1997, pp. 2399–2418. 1
- [77] Brian Randell, *System structure for software fault tolerance*, Ieee transactions on software engineering (1975), no. 2, 220–232. 2.6, 6.1
- [78] Nicolás Rosner, Valeria S. Bengolea, Pablo Ponzio, Shadi Abdul Khalek, Nazareno Aguirre, Marcelo F. Frias, and Sarfraz Khurshid, *Bounded exhaustive test input generation from hybrid invariants*, Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014, 2014, pp. 655–674. 1.2
- [79] Nicolás Rosner, Jaco Geldenhuys, Nazareno M. Aguirre, Willem Visser, and Marcelo F. Frias, *BLISS: Bounded Lazy Initialization with SAT Support*. 2.1, 4
- [80] Karem A Sakallah, *Symmetry and Satisfiability.*, Handbook of Satisfiability **185** (2009), 289–338. 4

-
- [81] Hesam Samimi, Ei Darli Aung, and Todd D. Millstein, *Falling Back on Executable Specifications*, ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings, 2010, pp. 552–576. 1.2
- [82] Fred B. Schneider, *The State Machine Approach: A tutorial*, Tech. report, USA, 1986. 6.2
- [83] Ian Sommerville, *IEEE Software and Professional Development*, IEEE Software **33** (2016), no. 2, 90–92. 1
- [84] Emina Torlak and Daniel Jackson, *Kodkod: a relational model finder*, Proceedings of the 13th international conference on Tools and algorithms for the construction and analysis of systems (Berlin, Heidelberg), TACAS'07, Springer-Verlag, 2007, pp. 632–647. 4.3, 6.5
- [85] Michael Treaster, *A survey of fault-tolerance and fault-recovery techniques in parallel systems*, arXiv preprint cs/0501002 (2005). 6.1
- [86] Andrew M Tyrrell, *Recovery blocks and algorithm-based fault tolerance*, Proceedings of EUROMICRO 96. 22nd Euromicro Conference. Beyond 2000: Hardware and Software Design Strategies, IEEE, 1996, pp. 292–299. 6.1
- [87] Willem Visser, Corina S. Pasareanu, and Radek Pelánek, *Test input generation for java containers using state matching*, Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17-20, 2006, 2006, pp. 37–48. 5
- [88] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid, *Test input generation with Java Pathfinder*, Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis (New York, NY, USA), ISSTA '04, ACM, 2004, pp. 97–107. 6.5
- [89] Yichen Xie and Alex Aiken, *Saturn: A scalable framework for error detection using Boolean satisfiability*, ACM Trans. Program. Lang. Syst. **29** (2007), no. 3. 6.5
- [90] Stephen S Yau and Ray C Cheung, *Design of self-checking software*, ACM SIGPLAN Notices **10** (1975), no. 6, 450–455. 6.1

- [91] 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, 2012, pp. 2–17. 1.1.1, 1.2, 6.4
- [92] Razieh Nokhbeh Zaeem and Sarfraz Khurshid, *Contract-Based Data Structure Repair Using Alloy*, ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings, 2010, pp. 577–598. 1.2, 4, 5.3, 6.4
- [93] Ilya Zakirzyanov, Antonio Morgado, Alexey Ignatiev, Vladimir Ulyantsev, and Joao Marques-Silva, *Efficient Symmetry Breaking for SAT-Based minimum DFA inference*, Language and Automata Theory and Applications (Carlos Martín-Vide, Alexander Okhotin, and Dana Shapira, eds.), Lecture Notes in Computer Science, Springer, 2019, International Conference on Language and Automata Theory and Applications 2019, LATA 2019 ; Conference date: 26-03-2019 Through 29-03-2019, pp. 159–173. 4