Optimización de Técnicas de Abstracción Para Especificaciones DynAlloy

Trabajo Final (Cód. 1970)
Carrera: Licenciatura en Ciencias de la Computación

Ariño, Rodrigo
DNI 32.861.642

Degiovanni, Renzo G.
DNI 32.705.627

Fervari, Raúl A.
DNI 32.705.462

—

Departamento de Computación
Facultad de Cs. Exactas Fco-Qcas y Naturales
Universidad Nacional de Río Cuarto

—

8 de julio de 2010
Director: Dr. Nazareno Aguirre
Co-Director: Lic. Pablo Ponzio

Índice general

1. Introducción **3**
- 1.1. Los Métodos Formales . . . 3
  - 1.1.1. Desde la Verificación Manual Hacia la Automática . . . 4

2. Alloy **7**
- 2.1. Lógica . . . 8
  - 2.1.1. Constantes y Operadores . . . 9
- 2.2. Lenguaje . . . 11
  - 2.2.1. Construcción y análisis de un pequeño módulo Alloy . . . 11
  - 2.2.2. Signaturas y atributos . . . 15
  - 2.2.3. Tipos y chequeo de tipos . . . 16
  - 2.2.4. Hechos . . . 17
  - 2.2.5. Aserciones . . . 17
- 2.3. Análisis . . . 17
  - 2.3.1. Búsqueda de instancias . . . 17
  - 2.3.2. Alcance (scope) - La hipótesis de la cota pequeña . . . 18

3. DynAlloy **19**
- 3.1. Mejorar Alloy con Acciones . . . 19
- 3.2. Sintaxis y Semántica de DynAlloy . . . 20
- 3.3. Análisis de especificaciones DynAlloy . . . 22
- 3.4. La herramienta: DynAlloy Translator . . . 23
  - 3.4.1. Ventajas . . . 24

4. Abstracción por Predicados **26**
- 4.1. Introducción . . . 26
- 4.2. Descripciones Concretas y Abstractas . . . 26
  - 4.2.1. Contraejemplos Concretizables y Espúreos . . . 27
- 4.3. Abstracción Para Especificaciones DynAlloy . . . 28
  - 4.3.1. Verificando Propiedades Sobre Modelos Concretos y Abstractos . . . 29
- 4.4. Refinamiento de Abstracciones Basado en Descubrimiento de Predicados . . . 33
  - 4.4.1. Descubrimiento de Predicados . . . 33

5. Implementación de Abstracción por Predicados **35**
- 5.1. Algoritmos de Abstracción . . . 36
  - 5.1.1. Ejecución Bajo Demanda . . . 36
  - 5.1.2. Ejecución Bajo Demanda + Refinamiento de Abstracciones . . . 37
  - 5.1.3. Detección de ciclos . . . 38
  - 5.1.4. Ejecución Bajo Demanda + Refinamiento de Abstracciones + Detección de Ciclos . . . 39
- 5.2. Almacenamiento de Cálculos Previos . . . 41
- 5.3. Refinamiento de Abstracciones . . . 41

5.4. Estructura y Diseño 42
5.5. Uso de la Herramienta 44
5.5.1. Carga de Parámetros 44
5.5.2. Ejecución 46
5.5.3. Interpretación de resultados 46
5.5.4. Algunas restricciones a la hora de utilizar la herramienta 47
5.5.5. Desarrollo y Avance de la Herramienta 48

6. Casos de estudio **50**
6.1. Abstracción sin necesidad de refinamiento 50
6.1.1. La eliminación preserva aciclicidad de listas 51
6.1.2. Asignación de valores a una lista según un flag 52
6.1.3. División de una lista en dos 53
6.2. Abstracción utilizando refinamiento 54
6.2.1. La eliminación preserva aciclicidad de listas + Refinamiento 55

7. Conclusiones y trabajo futuro **57**
7.1. Conclusiones 57
7.2. Trabajo futuro 58

A. Especificaciones completas de casos de estudio **59**
A.1. La eliminación preserva aciclicidad de listas 59
A.2. Asignación de valores a una lista según un flag 62
A.3. División de una lista en dos 65

Capítulo 1

Introducción

Desde tiempos previos a la llamada *crisis del software*, se ha reconocido que la complejidad y el tama no de los sistemas de software, cuyo campo de aplicación ha crecido significativamente respecto de sus aplicaciones en cálculo numérico de los inicios de la computación, demanda metodologías sistemáticas de desarrollo. Por supuesto, el objetivo de éstas es permitir crear, dise nar y mantener (exitosamente) software de calidad y de gran escala (en los tiempos estipulados).

Uno de los problemas más importantes asociados con la construcción de software es la corrección del mismo, es decir, en qué medida el software construido satisface los requisitos (funcionales o no) establecidos durante las etapas tempranas del desarrollo. Ésto es un factor crítico que hace a la calidad del software. Si bien existen varias metodologías que apuntan a garantizar la calidad del software, éstas se enfocan mayoritariamente en el proceso de desarrollo y no en el producto en sí (CMM [16]). Además, aquellas de las más usadas que se concentran en la calidad del producto desde el punto de vista funcional, son en general informales y ofrecen garantías de corrección muy limitadas. Éste es el caso del testing [6] y de la simulación [5].

Debido a varias razones, muchas de las cuales están ligadas a limitaciones intrínsecas de la computación, resulta que en la actualidad no exista un método efectivo que nos permita garantizar la corrección del software que desarrollamos, es decir, que funcione de manera esperada en todos los posibles casos en que será utilizado. De hecho, los sistemas de software desarrollados utilizando las metodologías de desarrollo más ampliamente difundidas, contienen en general un gran número de errores, que requieren de un gran esfuerzo de mantenimiento luego de finalizar el proceso de construcción. Si bien el problema de desarrollar software libre de fallas es muy difícil, existen varias metodologías que apuntan a mejorar la calidad del mismo. Entre éstas metodologías se destaca una familia particular, conocida como *Métodos Formales* [7]. Los métodos formales brindan garantías de corrección notablemente más fuertes que las brindadas por técnicas informales, gracias a las sólidas bases matemáticas sobre las cuales están fundados. Sin embargo, es sabido que su aplicación práctica es sumamente difícil, puesto que requieren de conocimientos y experiencias en la manipulación de formalismos matemáticos.

1.1. Los Métodos Formales

En busca de proveer garantías del correcto funcionamiento del software, surgieron una variedad de técnicas y metodologías de desarrollo con sólidas bases matemáticas y lógicas conocidas como *Métodos Formales*. Éstos se basan en el uso de notaciones matemáticas con semántica formal, que permiten razonar sobre modelos de las especificaciones de los sistemas. Dichos modelos deben reflejar las características del programa que se quiere construir. Si ésto sucede, pueden verificarse propiedades sobre los modelos, aún antes de haber construido el software, lo que posibilita detectar errores en etapas tempranas del desarrollo. Además, la utilización de métodos formales ayuda a la comprensión del problema a resolver por parte de los desarrolladores, lo que contribuye a mejorar la calidad del programa.

Tradicionalmente, los métodos formales han estado asociados a mecanismos deductivos para razonar sobre programas ([8], [9]). Ésto ha hecho que la aplicación de métodos formales en la práctica requiera de desarrolladores con una sólida formación matemática-lógica, y que su aplicación demande importantes costos de tiempo y dinero. Por ésta razón, la aplicación de métodos formales en la industria se ha visto confinada, durante muchos años, a sistemas críticos de software/hardware cuyo mal funcionamiento o falla puede causar da nos de magnitud, como la pérdida de vidas humanas. Por éste motivo, las metodologías de desarrollo más utilizadas en la actualidad son informales, en el sentido de que las notaciones y procesos utilizados no cuentan con semántica formal o precisamente descripta en algún formalismo matemático (por ejemplo, la amplia mayoría de las notaciones abarcadas por UML [Booch et al. 1998]). Más aún, el *testing*, claramente una técnica informal, sigue siendo en la actualidad la técnica para la garantía de corrección (funcional) del software más ampliamente utilizada en la práctica.

Sin embargo, es ampliamente reconocido que los beneficios que ofrecen los métodos formales, en comparación con las técnicas informales, en lo que hace a la corrección del software, son demasiado importantes como para decidir abandonar su utilización debido a su costo. Por ésta razón, y en búsqueda de hacer a los métodos formales más fácilmente utilizables, se han desarrollado lenguajes para describir modelos matemáticos cada vez más sencillos, con herramientas de software que soporten tanto el proceso de contrucción de modelos, como el análisis de los mismos.

La evolución en los métodos formales puede observarse claramente comparando lenguajes muy populares hace unos a nos, tales como Z [Spivey 1988], VDM [Jones 1986], y Larch [Guttag and Horning 1993], con otros más modernos como B [Abrial], ASM y Alloy [Jackson]. También puede observarse la evolución en otros lenguajes ligados a descripciones más operacionales: CSP [Hoare 1985] y CCS [Milner 1980], si bien siguen siendo importantes referentes, no son tan fácilmente utilizables como Statecharts [Harel 1987] y FSP [Magee y Kramer 1999].

Los métodos formales se usan en general en dos actividades: la especificación y la verificación. El proceso de especificación es el acto de describir las cosas de manera precisa. El principal beneficio de hacer esto es conseguir una comprensión más profunda del sistema que está siendo especificado. A través de este proceso de especificación, los desarrolladores descubren errores de dise no, inconsistencias, ambigüedades y partes incompletas. Un subproducto tangible de este proceso es un resultado que puede ser analizado formalmente, por ejemplo, chequeado para comprobar que sea consistente internamente o usado para derivar otras propiedades del sistema especificado. La especificación es un mecanismo de comunicación útil entre cliente y dise nador, entre dise nador e implementador, y entre implementador y verificador. Además, sirve como documentación del sistema.

A partir de una correcta y completa especificación de un sistema se pueden probar propiedades y/o características del mismo. La verificación está fuertemente relacionada con el descubrimiento de errores, a partir de una descripción formal del sistema. Permite comprobar que el sistema y sus componentes cumplen con la especificación del problema.
La verificación puede describirse básicamente, de la siguiente manera: dado un programa `M` y una especificación `h` determinar si el comportamiento de `M` se corresponde con la especificación `h`.

1.1.1. Desde la Verificación Manual Hacia la Automática

Como se mencionó anteriormente, en los primeros métodos formales, la verificación axiomática era el paradigma predominante de verificación. La orientación de este paradigma era la demostración manual para la corrección de programas secuenciales (determinísticos), que normalmente comenzaban con una entrada y terminaban con una salida. Más adelante aparecieron nuevos trabajos que dieron lugar a nuevos avances, como el trabajo de Floyd [Fl67] que estableció principios básicos para demostrar correcciones parciales y totales, y el de Hoare [Ho69] que propuso un sistema lógico, el cual proporciona una serie de reglas de inferencia para razonar sobre la corrección de programas imperativos. La principal característica de esta lógica, es la terna **"{Q} S {R}"**, donde Q y R son predicados lógicos que deben satisfacerse para que el programa `S` funcione. Una importante ventaja del método de Hoare es que era *composicional*, tal es así que la demostración de un programa se obtenía desde la demostración de sus sub-programas.

Los trabajos de Floyd-Hoare tuvieron un gran éxito intelectual, a pesar de su poco uso en la práctica. Éstos dieron a conocer importantes conocimientos científicos y de gran avance para la verificación. La demostración de sistemas era propuesta para nuevos lenguajes de programación, y para programas peque nos se comenzaban a realizar demostraciones de corrección. Sin embargo, éstos trabajos eran limitados en la práctica. No resultaban ser para nada accesibles para programas de gran tama no. Los problemas empiezan con el enfoque manual de la construcción de una demostración. Éstas demostraciones formales involucran el uso de fórmulas lógicas extremadamente largas, cuya manipulación es difícil y tediosa.

Se puede intentar un proceso de construcción de prueba parcialmente automático, usando un demostrador de teoremas interactivo. Este puede aliviar mucho la carga. Sin embargo, la ingeniosidad del humano es todavía requerida para invariantes (por ejemplo dise nar aserciones adecuadas para invariantes de ciclos) y diversos lemas. El demostrador de teoremas puede también requerir de un operador experto para que sea usado efectivamente.

Alrededor de 1980, el principal paradigma para verificación era la demostración de teoremas manualmente, razonando por medio de axiomas formales y reglas de inferencia orientados hacia programas secuenciales. La necesidad de abarcar programas concurrentes, y el deseo de evitar las dificultades de las pruebas deductivas manuales, motivaron el desarrollo de una técnica enteramente algorítmica: el *model checking*. El model ckecking es una técnica que consiste en la construcción de un modelo finito del sistema y comprobar que las propiedades deseadas se cumplen en el modelo. La comprobación se realiza como una búsqueda exhaustiva en el espacio de estados, la cual es seguro que finalice debido a que el modelo es finito. El reto técnico en model checking es el de inventar algoritmos y estructuras de datos que permita manejar espacios de búsqueda grandes y lenguajes adecuados para describir las propiedades a verificar. Model checking se usó en un principio en la verificación de hardware y de protocolos. Actualmente se aplica al análisis de especificaciones de sistemas de software.

La primera implementación de model checking, usaba grafos de transición de estados para la representación, y técnicas eficientes para la exploración de estados. Sin embargo, el *problema de explosión de estados*, en donde el número de estados del sistema aumentaba de tama no exponencialmente con respecto al número de componentes del sistema, limitaba tales técnicas a sistemas con menos de un millón de estados.

Alrededor de 1990, se incluyen técnicas que usan exploración de espacio de estado simbólicos (y por supuesto representaciones simbólicas de estados). Éstas técnicas, se basaban en el uso de BDDs (Binary Decision Diagrams). Los BDDs poseen características de funciones de conjuntos de estados, y permiten el cómputo de transiciones entre conjuntos de estados, que antes se computaban entre estados individuales. A pesar de que brindaba la posibilidad de manejar modelos de mayor tama no, también tenía sus desventajas. Con el fin de lidiar con los problemas del uso de BDDs, se presentó una nueva técnica que combinaba model checking con el análisis de satisfacibilidad (SAT-solving). Esta técnica es conocida como *bounded model checking*[18], la cual brinda una eficiente búsqueda en el espacio de estados, y para ciertos tipos de problemas obtienen un mayor rendimiento con respectos a otros métodos. Así se obtuvieron mejores resultados para el chequeo de propiedades, debido a que la búsqueda de contraejemplos se realiza eficientemente en *bounded model checking*, además requiere menor manipulación por parte del usuario, y el chequeo de satisfacibilidad alcanza muy pocas veces la explosión exponencial de estados. Esto último se debe al uso de SAT-Solving, que es una técnica que se basa en el chequeo de satisfacibilidad, es decir, determinar si las variables de cierta fórmula booleana pueden tener asignaciones en las cuales tal fórmula se evalúe a verdadero. Igualmente importante, es determinar cuando tales asignaciones no existen. En éste último caso, se dice que la fórmula en insatisfacible, mientras que en otro caso la fórmula es satisfacible. Si bien el problema de SAT es NP-Completo, existen herramientas que automatizan el chequeo y utilizan distintas heurísticas para obtener una verificación eficiente. Una herramienta que se basa en SAT-Solving, es *Alloy Analyzer*. Alloy Analyzer da soporte al análisis de modelos escritos en el lenguaje *Alloy*, uno de los más populares lenguajes formales derivados de *Z*, que se utiliza para describir programas utilizando una lógica relacional. Alloy Analyzer fue desarrollado para proveer análisis completamente automático, a diferencia de las demostraciones de teoremas interactivas comúnmente utilizadas en lenguajes similares a Alloy. El desarrollo del Alloy Analyzer estuvo originalmente inspirado en el análisis automático provisto por el Model Checking. Sin embargo, el Model Checking es poco adecuado para el tipo de modelos especificados en Alloy, por lo cual el Alloy Analyzer incorpora SAT-Solving. Básicamente, se construye una fórmula booleana correspondiente a partir del modelo

relacional descripto en Alloy, y se invoca a un SAT-Solver sobre dicha fórmula booleana. Una vez que se encuentran soluciones, se realiza la asociación de constantes a variables en el modelo lógico relacional.

El Problema de la Explosión Combinatoria de Estados

El problema más serio de model checking es la explosión de estados. El tama no del grafo de estados puede ser exponencial al tama no del programa. Un programa concurrente con k procesos puede tener un grafo de estados de tama no `exp(K)`. Para una instancia de una red con 100 cajeros automáticos, cada uno controlado por una máquina de estados finitos con 10 estados, se puede tener 10^{100} estados globales.
Hoy, model checking es apto para verificar protocolos con millones de estados y circuitos de hardware con más o menos 10^{50} estados. Incluso algunos sistemas con un número infinito de estados pueden ser tratados con model checking, si se obtiene una apropiada representación finita del conjunto infinito de estados, en términos de restricciones simbólicas. Para evitarlo, diversos investigadores han desarrollado técnicas basadas en algoritmos simbólicos, **abstracción**, reducción de orden parcial y model checking on the fly.

En el presente trabajo, se presenta la implementación de una técnica de abstracción, la cual permite lidiar con éste problema.

Capítulo 2

Alloy

En las primeras etapas de la construcción del software , las ideas alrededor del diseño suelen tomar forma de *abstracciones*. Una abstracción en el sentido de [1] es "una estructura pura y simple –una idea reducida a su forma esencial". Un problema que desde hace ya muchos años ha sido observado es la distancia entre las abstracciones iniciales y el código que las implementa. Esto hace que, generalmente, muchos de los problemas de diseño de las abstracciones no puedan ser observados hasta llegada la etapa de implementación.

Este problema tiene que ver con que muchas ideas, en abstracto, parecen correctas, pero al llevarlas a cabo (implementarlas) surgen errores, inconsistencias e incoherencias.

Una de las tareas que es ampliamente aceptada para intentar resolver, al menos en parte, este problema es la *especificación* de las abstracciones. Esto incluye en muchos casos tanto de la especificación de requisitos como la especificación del diseño de la solución (y las abstracciones asociadas a éste).

Existen varias alternativas para describir estas especificaciones, la mayor parte de las cuales utilizan *lenguages informales* (es decir, lenguajes cuyas expresiones cuentan con significados intuitivos pero no formalizados adecuadamente de manera tal de eliminar cualquier tipo de ambigüedades en su interpretación). Alloy, en cambio, es un alternativa *formal*, dado que las especificaciones que uno describe en el lenguaje gozan de semántica formal, libre de ambigüedades. Este lenguaje ofrece más que un formalismo libre de ambigüedad: brinda una poderosa y madura herramienta de análisis de especificaciones, **Alloy Analyzer**. Esta herramienta permite analizar especificaciones, en busca de ambigüedades, inconsistencias, comprensiones erróneas, etc, de manera completamente automática.

Esto se logra mediante la reducción del análisis a un problema de satisfacibilidad de una formula proposicional, y el empleo de poderosos y sofisticados *SAT Solvers*.

El mecanismo de análisis tiene, sin embargo, algunas limitaciones. La principal tiene que ver con la *incompletitud* del análisis. Más específicamente, cuando utilizamos la herramienta para comprobar una propiedad, pueden darse dos situaciones:

- que la herramienta encuentre un contraejemplo, garantizando que la propiedad es inválida,
- que la herramienta no encuentre contraejemplos dentro de las cotas establecidas por el usuario. En este caso, al no encontrar contraejemplos, ganamos confianza en la propiedad, aunque **no** tenemos garantías de su validez, ya que podrían en principio existir contraejemplos fuera de las cotas establecidas.

Luego, el análisis asociado a Alloy puede entenderse como una forma de testing. Sin embargo, como se describe en [1, Jackson 2006], el alcance de la técnica es superior a la usualmente asociada al testing, ya que la cantidad de casos examinados suele ser muy grande (del orden de miles de millones) y éstos no deben ser provistos manualmente, sino que la herramienta los genera sin intervención del usuario.

Como se describe en [1, Jackson 2006], se pueden detectar algunos elementos claves en Alloy: una *lógica*, un *lenguaje* y un *análisis*:

[0]Este capítulo está basado fuertemente en [Jackson 2006]. Una parte importante del texto de este capítulo ha sido tomado del mismo, traducido al castellano.

- La ***lógica*** permite la construcción de los bloques del lenguaje. Todas las estructuras son representadas como *relaciones*, y todas las propiedades estructurales son expresadas mediante simples (pero poderosos) operadores relacionales. Los estados y ejecuciones son expresadas mediante fórmulas y expresiones booleanas, conocidas como *constraints*, permitiendo refinarlos mediante la introducción de nuevos constraints.
- El ***lenguaje*** agrega pequeños detalles a la sintaxis de la lógica para estructurar la descripción. Alloy permite sub-tipos, unión de tipos y brinda un sistema simple de módulos que permite que declaraciones genéricas y constraints sean reusados en diferentes contextos.
- El ***análisis*** es una forma de *constraint solving*. Se lo puede considerar en dos partes: *Simulación*, trata la búsqueda de instancias de estados o ejecuciones que satisfacen una propiedad y *Verificación*, trata la búsqueda de contraejemplos, una instancia que viola la propiedad dada.

2.1. Lógica

En el núcleo de todo lenguaje de modelado existe una *lógica* que provee los conceptos fundamentales. Ésta debe ser pequeña, simple y expresiva. Alloy utiliza una lógica relacional que combina los cuantificadores de la lógica de primer orden con operadores del cálculo relacional. Esta lógica es fácil de aprender y sorpresivamente poderosa. Una de las características clave, que la distingue de las lógicas tradicionales, es la generalización de la noción del *join* (composición) relacional. Como en una base de datos relacional, una relación es una conjunto de tuplas. Los conjuntos son representados como relaciones con una sóla columna, y los escalares como singletons. Consecuentemente, el mismo operador join puede ser aplicado a escalares, conjuntos y relaciones.

La lógica de Alloy soporta tres diferentes estilos, que pueden mezclarse y variarse a gusto propio:

- En el estilo de ***cálculo de predicados*** hay sólo dos tipos de expresiones: nombres de relación, que son utilizados como predicados, y tuplas formadas a partir de variables cuantificadas.
- En el estilo de ***navegación de expresiones***, las expresiones denotan conjuntos, las cuáles son formadas *navegando* desde las variables cuantificadas a través de relaciones.
- En el estilo de ***cálculo relacional***, las expresiones denotan relaciones, en las cuales no hay cuantificadores.

El estilo de cálculo de predicados es usualmente muy detallado; el estilo de cálculo relacional suele ser complicado de entender, aunque permite expresar mucho con expresiones cortas. Sin embargo, el estilo más empleado es el de navegación, con usos ocasionales de los otros estilos.

> *Los estilos mencionados no poseen el mismo poder expresivo. El estilo de navegación es el más expresivo. El cálculo de predicados carece de clausura transitiva, por lo tanto hay propiedades accesibles que no puede expresar. El cálculo relacional no tiene cuantificadores, y no todos los cuantificadores del cálculo de predicados pueden ser expresados relacionalmente.*

Átomos y Relaciones. Alloy no permite relaciones de alto orden, es decir, no permite que una relación contenga relaciones. Esta restricción hace la lógica más manejable para el análisis, pero hace que la lógica pierda un poco de poder expresivo, aunque casi siempre se puede expresar una relación de alto orden en una o más relaciónes planas.

En Alloy, toda expresión denota una relación. Como las relaciones pueden ser de cualquier aridad, esto nos permite en particular denotar conjuntos, relaciones binarias con propiedades particulares (funciones, por ejemplo), etc.

Para denotar elementos, se utilizan singletons, es decir, conjuntos de un único elemento. Luego, por ejemplo e será manipulado en Alloy a través del conjunto $\{e\}$.

Las intepretaciones de modelos Alloy se realizan sobre dominios de valores, denominados átomos. Tendremos tantos dominios como declaraciones de dominios en la especificación (que en Alloy se realizan mediante signaturas).

2.1.1. Constantes y Operadores

El lenguaje de la aritmética consiste en constantes (tales como 0,1,2...) y operadores (tales como +,*,-...). De la misma manera el lenguaje de las relaciones tiene sus constantes y operadores.

Constantes Hay tres constantes:

none relación unaria vacía

univ relación unaria universal

iden relación binaria identidad

Nótese que *none* y *univ*, representando el conjunto que no contiene átomos y el conjunto que contiene todos los átomos respectivamente, son unarios. Para representar una relación binaria vacía, se puede escribir $none -> none$. La relación identidad es binaria, y contiene tuplas que relacionan un átomo con si mismo.
Ejemplo Consideremos los siguientes dominios y los átomos que los componen:

```
Name = {(N0),(N1),(N2)}
Addr = {(D0),(D1)}
```

Luego las relaciones constantes mencionadas tienen los siguientes valores

```
none = {}
univ = {(N0),(N1),(N2),(D0),(D1)}
iden = {(N0,N0),(N1,N1),(N2,N2),(D0,D0),(D1,D1)}
```

Estas relaciones constantes son muy importantes, en particular en el estilo del cálculo relacional. Por ejemplo, la expresión $no\ \hat{}r\ \&\ iden$ indica que la relación binaria r es acíclica.

Operadores de Relaciones Básicas y Fórmulas

+ union : una tupla a pertenece a $p + q$ si pertenece a p *ó* a q (ó a ambos).

& intersección : una tupla a pertenece a $p\&q$ si pertenece a p *y* a q.

- diferencia : una tupla a pertenece a $p - q$ si pertenece a p *pero no* a q.

in subconjunto : p *in* q es true cuando toda tupla de p es también tupla de q.

= igualdad : $p = q$ es true cuando p y q tienen las mismas tuplas.

Para poder aplicar estos operadores, las relaciones involucradas deben tener la misma aridad. Consideremos el siguiente ejemplo:

```
Name = {(G0), (A0), (A1)}
Alias = {(A0), (A1)}
Group = (G0)
RecentlyUsed = {(G0), (A1)}

Alias + Group = {(G0), (A0), (A1)}

Alias & RecentlyUsed = {(A1)} Conjunto de los alias recientemente usados.
```

```
Name - RecentlyUsed = {(A0)} Nombres no usados recientemente.

RecentlyUsed in Alias   "Todos los recientemente usados están en Alias."
                      Es falsa porque G0 fue usado recientemente pero no está en Alias.

RecentlyUsed in Name    "Todos los recientemente usados estan en Name". Es verdadera.

Name = Group + Alias    "Cada nombre es un grupo o un alias". Es verdadera.
```

Otros Operadores Relacionales

`->` producto : `p->q` todas las tuplas que se pueden combinar concatenando las tuplas de p a las tuplas de q

· dot join : Para componer dos tuplas (`s_1->s_2->...->s_m` , `t_1->t_2->...->t_n`) primero hay que chequear que el átomo `s_m` matchee con el átomo `t_1`. Si esto ocurre, el resultado es una tupla que no contiene al átomo que hizo matching:`s_1->...->s_(m-1)->t_2->...->t_m`.En caso contrario, el resultado es una relación vacía .Este operador no es asociativo.

[ ] box join : Es exactamente identico al dot join, pero toma sus argumentos en diferente orden y precedencia. La expresión `e_1[e_2]`tiene el mismo significado que `e_2.e_1` .

~ transpuesta : forma una nueva relación invirtiendo el orden de los átomos de cada tupla.

^ clausura transitiva: Dada una relación binaria r, `^r=r+r.r+r.r.r+...`

* clausura reflexo-transitiva : Sea r una relación binaria, `*r=^r+iden`.

<: restricción del dominio

:> restricción del rango

++ override : Sean p y q dos relaciones, `p++q=p-(domain(q)<:p)+q`.

Operadores Lógicos

not (`!`) negación

and (`&&`) conjunción

or (`||`) disyunción

implies (`=>`) implicación

else (,) alternativa

`<=>` bi-implicación

Cuantificación Una fórmula cuantificada tiene la forma:

$$Qx : e|F$$

donde F es una restricción que contiene la variable x, e es una expresión ligando a x, y Q es un cuantificador.
Las formas de cuantificación en Alloy son:

all $x : e|F$: F vale para todo x en e

some $x : e|F$: F vale para algún x en e

no $x : e|F : F$ no vale para ningún x en e

lone $x : e|F : F$ vale para a lo sumo un x en e

one $x : e|F : F$ vale para exactamente un x en e

Algunos de estos cuantificadores también pueden ser aplicados a expresiones:

some $e : e$ tiene alguna tupla.

no $e : e$ no tiene tuplas.

lone $e : e$ tiene a lo sumo una tupla.

one $e : e$ tiene exactamente una tupla.

Cuando en una expresión cuantificada alguna de las variables ligadas por un cuantificador no es un escalar, tenemos una cuantificación de alto orden. Éstas son expresadas en Alloy, pero no siempre son analizables. Ejemplo: La siguiente expresion tienen cuantificación de alto orden:

- **all** s, t: **set univ** | s + t = t + s
  La unión sobre conjuntos es conmutativa.

Multiplicidades de relaciones

- `r:A->one B` *una función cuyo dominio es A*
- `r:A one-> B` *una relación inyectiva cuya imágen es B*
- `r:A->lone B` *una función parcial sobre el dominio A*
- `r:A one->one B` *una función biyectiva con dominio A e imágen B*
- `r:A some->some B` *una relación con dominio A e imágen B*

2.2. Lenguaje

Un *lenguaje* para describir abstracciones de software es más que sólo una lógica. Se necesitan formas de organizar el modelo, para poder construir otros más largos a partir de pequeños, y factorizarlo para poder ser reutilizado más de una vez.

Alloy es un pequeño lenguaje. Algunos de sus rasgos son únicos, como las *signaturas* y el *scope*. El resto (módulos, polimorfismo, funciones parametrizadas, ...) son rasgos comunes de la mayoría de lenguajes de modelado.

2.2.1. Construcción y análisis de un pequeño módulo Alloy

La forma de organizar un módulo Alloy es:

- Módulo (*module*): expresa el nombre del módulo. Los módulos Alloy son nombrados como los módulos Java; debe coincidir con el nombre del archivo. Éstos tienen la extensión "*.als*"
- Signaturas (*sig*): Representan un conjunto de átomos, que pueden tener algún atributo, representando alguna relación.
- Funciones y Predicados (*fun, pred*): construcciones parametrizadas que permiten expresar diferente expresiones y pueden ser utilizadas en diferentes contextos. Los predicados son muy utilizados para describir operaciones.

- Restricciones (*fact*): permiten expresar restricciones e invariantes de clase.
- Aserciones (*assert*): predicados introducidos por el usuario con el motivo de chequear su validez.
- Comandos (*run, check*): instrucciones que el analizador ejecuta.

Para la comprensión de la estructura de un módulo Alloy, se considera como ejemplo un problema conocido como *el problema de las bolitas (marbles problem).*

El juego consiste en tomar de la bolsa, aleatoriamente, dos bolitas, y avanzar en el juego mediante las siguientes reglas:

1. Si ambas son verdes, quitarlas de la bolsa y poner 2 azules dentro.
2. Si al menos una es roja, quitarlas de la bolsa y poner 3 verdes.
3. Si ambas son azules, quitar una de la bolsa y devolver la otra.

Por supuesto, el juego termina si en algún momento llegamos a tener menos de dos bolitas en la bolsa. Se puede especificar, en Alloy, dicho juego de la manera que muestra la figura 2.1

```
-- marbles.als
module marbles
sig Marble { }
--Bag's state
sig Bag {
    red: set Marble,
    green: set Marble,
    blue: set Marble
}
-- rules
-- Rule 1:take two marbles, if both are green,
    throw them out and put two blue marbles in the bag
pred preRule1[s:Bag]{
    (#s.green)>=2
}
pred postRule1[s:Bag,s':Bag]{
    (s'.red = s.red) && ((#s'.green) = ((#s.green)-2)) && ((#s'.blue) = ((#s.blue)+2))
}
pred rule1[s: Bag,s': Bag]{
  preRule1[s] and postRule1[s,s']
}
-- Rule 2:take two marbles, if at least one of them is red,
    throw them out and put three new green marbles in the bag
pred preRule2[s:Bag]{
    (((#s.green) + (#s.blue) + (#s.red)) >=2 ) && ((#s.red) >=1 )
}
pred postRule2[s:Bag,s':Bag]{
  some aux: Bag |
    (((#aux.green) + (#aux.blue) + (#aux.red)) = ((#s.green) + (#s.blue) + (#s.red) -2))
    && ((#aux.red) < (#s.red))
    && (s'.red = aux.red)
    && (s'.blue = s.blue)
    && (#s'.green = (#aux.green+3))
}
```

```
pred rule2[s: Bag,s': Bag]{
  preRule2[s] and postRule2[s,s']
}
-- Rule 3:take two marbles, if both are blue, then throw only one out,
    and put the other one back in the bag
pred preRule3[s:Bag]{
    #s.blue>=2
}
pred postRule3[s:Bag,s':Bag]{
    (s'.red = s.red) && ((#s'.green) = (#s.green)) && ((#s'.blue) = ((#s.blue)-1))
}
pred rule3[s: Bag,s': Bag]{
  preRule3[s] and postRule3[s,s']
}
-- end actions
```

Figura 2.1: Módulo de *marbles.als*.

Al analizar el módulo Alloy, mostrado en la figura 2.1, se pueden notar dos cosas:

- Dos **signaturas**: `Marble, Bag`. La signatura `Marble` representa a las bolitas, sin importar su color. La signatura `Bag` representa a una bolsa, la cual puede contener un conjunto de bolitas rojas, otro de bolitas verdes y otro de bolitas azules.
- La definición de las 3 **operaciones** o reglas que el juego otorga. Básicamente las reglas relacionan dos bolsas, la anterior a aplicar la regla, y la bolsa resultante después de aplicarla. Las especificación de las operaciones están divididas en 2 predicados, la pre-condición y la post-condición, sólo para agilizar la lectura al lector. La pre-condición hace referencia a las condiciones que debe cumplir la bolsa para poder aplicar la regla, y la post-condición describe el estado de la bolsa que resulta de aplicarla.

Este modelo no contiene comandos, por lo tanto no hay análisis que podamos realizar. Luego, el primer análisis que se puede hacer es pedir al Alloy Analyzer que encuentre una instancia posible para ese modelo. Ésto se logra agregando un *predicado* y un *comando* más a nuestro modelo:

```
pred show [] {}
run show for 3 but 1 Bag
```

El predicado tiene el cuerpo vacío, por lo tanto no introducimos ninguna restricción. El comando especifica un *scope* que limita la búsqueda de instancias; en éste caso, hay a lo sumo tres elementos de cada signatura, excepto para la signatura Bag, la cual fue limitada sólo a un elemento. Una de las posibles instancias que muestra el AlloyAnalyzer se puede ver en la figura 2.2:

Un problema que tiene esta especificación es que permite instancias en las cuales una bolita puede ser de varios colores. La siguiente figura ilustra dicho problema:

El problema mencionado puede ser resuelto introduciendo al modelo la idea de que una bolita sólo puede ser roja, verde o azul. Ésto se puede lograr de dos maneras diferentes:

1. Una de las maneras es introducir un *echo* al modelo de la manera en que muestra la figura 2.4
2. Otra manera de hacer esto es definiendo a la signatura `Marble` como abstracta, y luego definir tres nuevas signaturas –`RED, GREEN, BLUE` –que extienden a `Marble` mediante la palabra reservada **extends** en Alloy. Luego definimos las relaciones, pertenecientes a `Bag`, *red*, *green* y *blue* como conjunto de `RED`, `GREEN` y `BLUE` respectivamente. Esta idea se ve plasmada en la figura 2.5

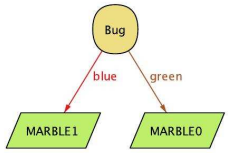

Figura 2.2: Instancia mostrada por el AlloyAnalyzer.

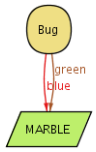

Figura 2.3: Otra instancia mostrada por el Alloy Analyzer.

Como se puede observar Alloy permite especificar propiedades de los elementos descriptos mediante hechos (*facts*). Esto permite en particular, especificar invariantes de clases *supuestos*. Alloy permite además la **extensión de signaturas**. Un uso particular de la extensión de signaturas es el uso de signaturas abstractas y signaturas unitarias que la extienden para definir conjuntos.

A diferencia de muchos lenguajes formales de especificaciones, Alloy admite el análisis automático de propiedades mediante la herramienta Alloy Analyzer. Para realizar chequeos de propiedades, es útil definir aserciones (fórmulas a chequear).

Una aserción en Alloy se forma de la siguiente manera:

assert *nombre-aserción*{
`propiedad-a-chequear`
}

donde `propiedad-a-chequear` es una fórmula lógica en la que puede haber invocación a predicados y funciones. Para comprender mejor lo que es una aserción, considere el siguiente ejemplo basado en el modelo marbles presentado anteriomente.

Nótese que el predicado *atLeastTwoMarbles* es válido si la bolsa que toma como parámetro contiene *al menos* dos bolitas.

La fórmula incluida en la aserción expresa la idea de un programa en el que inicialmente se tiene una bolsa (en el ejemplo, *b_0*) que contiene al menos dos bolitas. Luego al aplicar la regla 1 (*rule1*) se obtiene un bolsa en un estado diferente a la anterior (en el ejemplo, *b_1*), para luego chequear si la bolsa obtenida aún contiene al menos dos bolitas.

En la notación de triplas de Hoare, tal aserción se expresaría de la siguiente manera:

```
{atLeastTwoMarbles[b 0]} rule1[b 0,b 1] {atLeastTwoMarbles[b 1]}
```

Para poder chequear la validez de la aserción debemos introducir un comando de la siguiente manera

```
check program for 3 but 1 Bag
```

El comando ***check*** introducido es una orden para que el Alloy Analyzer chequee la validez de la aserción. Es decir, siguiendo el método explicado anteriormente, la herramienta trata de encontrar una instancia que viole la propiedad a chequear, o en otras palabras, busca un contra-ejemplo para la aserción. Téngase en cuenta que la búsqueda de instancias se realiza dentro del *scope* definido en el comando. Por lo tanto, si no encuentra un contra-ejemplo, no se puede *asegurar* que la propiedad vale para todos los casos posibles, ya que sólo se chequea para un número finito acotado, de interpretaciones posibles.

```
fact {
  all b:Bag |   (all r : b.red | ! r in (b.green + b.blue ) )and
                (all r : b.blue | ! r in (b.green + b.red ) )and
                (all r : b.green | ! r in (b.red + b.blue ))
}
```

Figura 2.4: Invariante para el modelo de marbles.

```
-- marbles.als
module marbles
abstract sig Marble { }
sig RED, BLUE, GREEN extends Marble {}

--states
sig Bag {
    red: set RED,
    green: set GREEN,
    blue: set BLUE
}
```

Figura 2.5: Herencia de signaturas en Alloy.

2.2.2. Signaturas y atributos

Una *signatura* introduce un conjunto de átomos. La declaración

```
sig A { }
```

introduce un conjunto llamado A. Una signatura también puede incluir declaraciones de relaciones, y puede introducir un nuevo tipo implícitamente.

Como se vio en ejemplo de la figura 2.5, un conjunto puede ser introducido como subconjunto de otro; esto es

```
sig A1 extends A {}
```

introduce un conjunto llamado $A1$ que es un subconjunto de A. Se dice que $A1$ es una *extensión* o una *subsignatura* de A.

Supongamos que se tiene el siguiente caso

```
sig A{}
sig A1 extends A {}
sig A2 extends A {}
```

Podemos inferir que $A1$ y $A2$ son disjuntos pero no que $A = A1 + A2$. Para lograr este efecto se debe definir a A como una signatura abstracta tal como ocurre en la figura 2.5. Con este tipo de declaraciones se logra un efecto de jerarquía de clasificación.

Otra de las cosas que permite Alloy es que una signatura puede ser declarada como un subconjunto de la unión de dos conjuntos: **sig** C *extends* $A + B$ $\{\}$.

Las relaciones pueden se declaradas como atributos de una signatura. Escribiendo

$$\textbf{sig}\ A\ \{f : e\}$$

```
assert program{
  all b_0:Bag,b_1:Bag|
           (atLeastTwoMarbles[b_0] and rule1[b_0,b_1]) implies atLeastTwoMarbles[b_1]
}
```

Figura 2.6: aserción en Alloy.

se introduce una nueva relación f cuyo dominio es A y cuyo rango está dado por la expresión e. Por lo tanto cuando definimos una variable $s : A$ y queremos hacer referencia al atributo f de s, basta con componer s con f de la siguiente manera : $s{\cdot}f$.

2.2.3. Tipos y chequeo de tipos

El sistema de tipos de Alloy cumple dos funciones. Primero, permite que el Alloy Analyzer detecte algunos errores antes de comenzar el análisis. Segundo, el sistema de tipos es usado para evitar sobrecarga. Cuando diferentes signaturas tienen atributos con el mismo nombre, el tipo de una expresión es usado para determinar a que atributo, con ese nombre, se hace referencia.

Tipos Básicos

Los tipos son asociados implícitamente con signaturas. Un *tipo básico* es introducido por cada signatura de nivel superior o cada signatura de extensión. Cuando una signatura $A1$ extiende a otra A, el tipo asociado a $A1$ es un *subtipo* del tipo asociado con A.

Dos tipos básicos se superponen si uno es subtipo de otro.

Tipo Relacional

Toda expresión tiene un *tipo relacional*, consistente de una unión de productos:
A1 -> B1->...
+ A2 -> B2->...
+ ...

donde cada uno de los A_i, B_i,..., son tipos básicos. Cada producto debe de tener la misma aridad para poder definir la unión de éstos.

Los tipos son deducidos automáticamente de modo que el valor del tipo siempre contenga el valor de la expresión; es decir esto es una sobreaproximación.

Los tipos son determinados como sigue:

1. La jerarquía de tipos básicos es obtenida a partir de la definición de las signaturas.
2. A cada atributo se le da el tipo que representa la expresión en la parte derecha de su definición.
3. Las restricciones del modelo son examinados, y el tipo es inferido para cada expresión, usando el tipo de signaturas y atributos, y los tipos de la variables cuantificadas.

Tipo Error

Hay dos clases de tipos de errores. Primero como la lógica asume que todas las relaciones tienen aridad fija, es ilegal formar expresiones que den como resultado una relación con aridad mixta. Por ejemplo, la unión de dos relaciones con diferente aridad es ilegal. Segundo, una expresión es ilegal si puede ser vista como redundante o contiene una sub-expresión redundante. Un caso común y simple de cuando una expresión es redundante, es cuando la relación es vacía.

En este sistema de tipos, la jerarquía de subtipos es utilizada principalmente para determinar que dos tipos son disjuntos.

2.2.4. Hechos

Las restricciones que se se asumen siempre como válidas se llaman *echos*(facts). Un modelo puede tener cualquier número de echos y no importa en el orden en el que fueron introducidos. A cada fact se le puede dar un nombre único.

Echos de Signatura Una restricción declarada inmediatamente después de una signatura es cuantificada implícitamente sobre sus elementos.
$sig\ A\ \{\ldots\}\ \{F\}$
es equivalente a escribir
$sig\ A\ \{\ldots\}$
$fact\{all\ this : A|F'\}$
donde F' es como F, pero cada mención de un atributo g que aparece en A o en uno de sus supertipos es reemplazado por $this.g$. *Los echos de signatura cumplen el mismo rol que los invariantes de clases.*

2.2.5. Aserciones

Las aserciones son utilizadas para chequear la validez de propiedades sobre nuestro modelo. Por lo general en las aserciones figuran llamadas a funciones o predicados. Algunos ejemplos son los siguientes:

```
assert testRule1{
  all s_0: Bag,s_1 Bag|
    (atLeastTwoMarbles[s_0] and rule1[s_0,s_1]) implies atLeastTwoBlueMarbles[s_1]
}

assert testRule2{
  all s_0: Bag,s_1: Bag|
    (atLeastTwoMarbles[s_0] and rule2[s_0,s_1]))  implies atLeastThreeGreenMarbles[s_1]
}
```

Las aserciones pueden utilizarse para la detección de errores en nuestros modelos. Por ejemplo, si creemos que el modelo especificado debe cumplir cierta propiedad, introducimos una aserción que especifique dicha propiedad, y un comando para invocar al Alloy Analyzer para chequearla. Si el analizador no logró encontrar un contra-ejemplo, tenemos la certeza de que nuestro modelo satisface la propiedad dentro de los límites impuestos en las cotas especificadas en el comando.

2.3. Análisis

2.3.1. Búsqueda de instancias

Alloy Analyzer soporta el análisis de especificaciones mediante dos comandos, run y check. A continuación describimos brevemente su significado:

- **run P** intenta encontrar una asignación de valores para las variables, de tal modo que hagan verdadero a **P**.
- **check Q** busca una asignación de valores para las variables, de manera tal que **Q** resulte falso. Esa instancia de valores para las variables, en caso de encontrarse, se conoce como *contra-ejemplo.*

Chequear una aserción y correr un predicado se puede reducir al mismo problema: la *búsqueda de instancia.* Es decir, en el caso del comando run, se trata de encontrar una instancia de las variables tal que haga verdadero el predicado y en el caso del chequeo de aserciones, se busca hacer válido a $\neg Q$, para así encontrar un contraejemplo para la propiedad (Q).

La lógica relacional subyacente a Alloy es indecidible. Esto quiere decir que es imposible construir una herramienta automática que diga cuando una aserción es válida. Es por eso que utiliza una noción de cota (scope), para poder acotar el dominio, y de esta manera buscar dentro del dominio finito instancias que violen la propiedad para luego reportarlas como contra-ejemplos.

2.3.2. Alcance (scope) - La hipótesis de la cota pequeña

Para hacer posible la búsqueda de instancias se define un scopeo alcance para limitar el dominio de las instancias consideradas. Si el usuario lo desea, se puede explicitar para cada signatura un valor por defecto.

Una vez que se tiene un dominio finito acotado (por el scope), se puede recorrer efectivamente todas las instancias buscando algún contra-ejemplo. De esta manera, si existe algún contra-ejemplo, cuyo tamaño este dentro del dominio, con seguridad va ser reportado como tal.

La búsqueda instancia suele ser confundida con el testing. De cierto modo, sólo se chequea la validez de la aserción para un número finito de casos, pero este número sería inalcanzable por medio del testing. Es por eso que mientras más grande es el scope, más garantías tenemos de que la aserción sea válida.

En la práctica se dá el caso de que si una aserción es inválida, no se necesita un scope muy grande para encontrar un contra-ejemplo. Esto dió origen a lo que se conoce como **la hipótesis de la cota pequeña**. Esta indica que para aseciones inválidas probablemente es suficiente con analizar instancias pequeñas para encontrar un contra-ejemplo. Nótese que esto es sólo una hipótesis, que puede interpretarse como un consejo práctico, de la siguiente manera: se puede comenzar a chequear una aserción a partir de un scope pequeño y luego ir incrementandolo a medida que no se encuentra contra-ejemplos.

Capítulo 3

DynAlloy

DynAlloy es una extensión del lenguaje Alloy, el cual agrega nuevas características que en cierto modo lo restringían. Unas de la deficiencias más importante de Alloy, es que carece de una forma sencilla para expresar propiedades dinámicas de sistemas. DynAlloy, inspirado en la lógica dinámica, consigue especificar propiedades de trazas de manera más apropiada. También se caracteriza por la sencillez y rapidez de poder traducir modelos DynAlloy a Alloy sin requerir intervención por parte del usuario, para luego realizar validaciones utilizando el Alloy Analyzer. A continuación se brinda una breve descripción del lenguaje a partir de [12].

3.1. Mejorar Alloy con Acciones

La representación de sistemas en Alloy, como se mencionó en el capítulo 2, está basado en modelos abstractos. Éstos modelos son definidos esencialmente en término de dominios, y operaciones entre ellos. En particular, se usan los dominios para especificar el estado de un sistema o un componente, y se utilizan las operaciones para la especificación de cambios de estado.
Como se observó, Alloy posee un poder expresivo suficiente como para poder expresar propiedades interesantes sobre los modelos abstractos, permitiendo simularlos y validar dichas propiedades con Alloy Analyzer. Sin embargo, Alloy no es apropiado para la validación de propiedades observando ejecuciones de trazas del sistema, en cierta forma, está limitado en este sentido. Si bien existe un mecanismo, propuesto en [10], para poder analizar las propiedades de ejecución, pero posee ciertas dificultades. El mismo consiste en la complementación de la especificación de un sistema con una especificación explícita de la traza, haciendolo posible mediante la inclusión de una nueva signatura para la traza de ejecución, y especificando restricciones, que indican cómo éstas trazas son construidas desde las operaciones. Básicamente, las trazas se definen como la composición de todos los estados intermedios visitados.
Si uno quisiera especificar la siguiente situación en Alloy:
Dada dos operaciones `Oper1 y Oper2` , un estado inicial α, un estado final β, y se necesita especificar que toda ejecución arbitraria de estas dos operaciones, que satisfagan el estado inicial α, terminen en el estado final β.
De acuedo a [10], es necesario dar una especificación explícita de la ejecución de trazas, como la siguiente:

1. Especificar el estado inicial como un estado que satisfaga α.
2. Especificar que sólo se puede pasar de un estado al siguiente a través de una de las operaciones `Oper1` o `Oper2`.
3. Especificar que el estado final satiface β.

La principal desventaja de este mecanismo es que la definición de trazas de ejecución depende fuertemente de la propiedad de trazas que uno desearía validar. Además, las especificaciones poseen un grado de dificultad

mas elevado, dado a que en ellas se mezclan dos aspectos del sistema evidentemente diferentes, la definición **estática** del dominio y las operaciones que constituye el sistema, y la especificacíon **dinámica** de las trazas de las ejecuciones de las operaciones.
Con este propósito Dynalloy introduce **acciones** [12], con una semántica de entrada/salida bien definida, permitiendo representar de manera adecuada cambios de estados y caracterizar propiedades en cuanto a las trazas de ejecución de un modo conveniente. Además le provee a Alloy una importante mejora en expresividad y analizabilidad.
Las acciones se expresan en DynAlloy en términos de sus pre y post-condiciones. De este modo, las acciones elementales pasan a tener la siguiente forma:

$$\{\alpha\}\ A\ \{\beta\}$$

Esta especificación nos asegura que, si ejecutamos la acción A en un estado que satisface la pre-condición α, obtenemos un estado en el que vale β.
Usando acciones, las trazas de ejecución son sólo usadas implícitamente. La específicación anterior puede ser escrita de una forma simple:

$$\begin{gathered}\{\alpha\}\\ (Oper1 + Oper2)^*\\ \{\beta\}\end{gathered}$$

Esta notación corresponde a la tradicional y conocida notación para **aserciones de corrección parcial**. Como puede observarse, en esta especificación no se requiere una referencia explícita a la trazas de ejecución. Sin embargo, las trazas existen y son soportadas por la semántica de acciones.
También se observa, que la acción definió una iteración finita ilimitada(*), la cual tendrá que acotarse a un límite si se desea utilizar el proceso de análisis de Alloy (Alloy Analyzer). Estos detalles se explicarán con mayor profundidad a continuación.

3.2. Sintaxis y Semántica de DynAlloy

La sintaxis de DynAlloy extiende la de Alloy agregando la siguiente cláusula para construir declaraciones de corrección parcial:

$$formula ::= \ldots \mid \{formula\}\ program\ \{formula\}$$
"partial correctness"

La sintaxis para *programas*(figura 3.1), agrega una nueva regla para permitir la construcción de acciones atómicas a partir de su pre y post-condición ($\overline{x}$ denota una secuencia de parámetros formales). En la figura 3.2 se describe la semántica de Dynalloy.

$program ::= (formula, formula)(\overline{x})$	"acción atómica"
$\mid formula?$	"test"
$\mid program + program$	"choice no deterministico"
$\mid program; program$	"composición secuencial"
$\mid program^*$	"iteración"

Figura 3.1: Gramática para composición de acciones en DynAlloy.

$$M[\{\alpha\}p\{\beta\}]e = M[\alpha]e \implies \forall e'(\langle e, e'\rangle \in P[p] \implies M[\beta]e')$$

$$P : program \to P(env \times env)$$
$$P[\langle pre, pos\rangle] = A(\langle pre, post\rangle)$$
$$P[\alpha?] = \{\langle e, e'\rangle : M[\alpha]e \wedge e = e'\}$$
$$P[p_1 + p_2] = P[p_1] \cup P[p_2]$$
$$P[p_1; p_2] = P[p_1]; P[p_2]$$
$$P[p^*] = P[p]^*$$

Figura 3.2: Semántica de DynAlloy.

Ahora con una sintaxis y semántica definida para Dynalloy, a modo de explicación, se continua con el ejemplo del **problema de las bolitas** (figura 2.1): los predicados *rem1, rem2, rem3*, que representaban operaciones sobre el modelo, ahora se definen como acciones *rem1, rem2, rem3* :

```
-- Rule 1:take two marbles, if both are green,
    throw them out and put two blue marbles in the bag
act rem1 [s: Bag] {
    pre { preRem1[s] }
    post { postRem1[s,s'] }
}
-- Rule 2:take two marbles, if at least one of them is red,
    throw them out and put three new green marbles in the bag
act rem2 [s: Bag] {
    pre { preRem2[s]}
    post {  postRem2[s,s']  }
}

-- Rule 3:take two marbles, if both are blue, then throw only one out,
    and put the other one back in the bag
act rem3 [s :Bag] {
    pre { preRem3[s] }
    post{ postRem3[s,s'] }
}
```

De esta manera, la acción *rem1*, define en *pre*, el estado que se debe satisfacer para ejecutar la acción (*preRem1[s]*:"la cantidad de bolitas verdes debe ser mayor o igual a dos"), y en *post*, el estado que retorna de ejecutar dicha acción (*postRem2[s,s']*:"se agregan dos bolitas azules, se quitan dos verdes y mantiene el número de bolitas rojas"). Como se observa, las acciones pueden modificar el valor de todas sus variables. En las especificación anterior, la variable *s'* representa el valor de la variable *s* después de la ejecución de la acción, y se asume que aquellas variables que no ocurren en la post-condición de forma primada, retienen su valor inicial.

3.3. Análisis de especificaciones DynAlloy

El dise no de Alloy fue profundamente influenciado por la intención de producir un lenguaje automáticamente analizable. Como se explicó, DynAlloy extiende Alloy, y la traducción de un modelo a otro (DynAlloy a Alloy) es prácticamente simple. Además de no requerir la intervención del usuario, la principal ventaja es que mantiene la posibilidad de analizar automáticamente las especificaciones DynAlloy. El principal fundamento que hace posible el chequeo de propiedades, es la traducción de aserciones de corrección parcial a fórmulas Alloy de primer orden. La cual puede ser amplia y muy dificultosa para entender, entonces no es visible para el usuario final de DynAlloy, quien unicamenente accede a la especificación declarativa de DynAlloy. A continuación, tomado de [12], se tratará de explicar los principales aspectos que conducen a la traducción. Comienza definiendo una función que computa la pre-condición más débil (weakest liberal precondition) de una fórmula con respecto a un programa (composición de acciones)

$$wlp : program \times formula \rightarrow formula$$

Nota: En general se utilizan los nombres x_1,x_2.. para las variables de los programas, y se usan los nombres x'_1,x'_2,.. para los valores de las variables de los programas después de la ejecución de la acción. Y se denota con $\alpha|_x^v$ la sustitución de todas las ocurrencias libres de las variables x por la variable v en la fórmula α.

Cuando una acción atómica a especificada como $\langle pre, pos\rangle(\bar{x})$ es usada como una composición de acciones, los parámetros formales son sustituidos por los parámetros actuales. Asi, la función *wlp* se define:

$$wlp[a(\overline{y}), f] = \ pre|_{\overline{x}}^{\overline{y'}} \implies all \ \ \overline{n} \ (post|_{\overline{x'}}^{\overline{n}}|_{\overline{x}}^{\overline{y'}} \implies f|_{\overline{y'}}^{\overline{n}})$$

Algunos puntos de la fórmula anterior necesitan ser explicados:

- Se asume que las variables libres en f están en $\overline{y'}$, $\overline{x_0}$. Las variables en $\overline{x_0}$ son generadas por la traducción de *pcat* dada a continuación.
- $\overline{n}$ es un arreglo de variables nuevas, una por cada variable modificada por la acción.
- El resultado de la fórmula tiene otra vez sus variables libres en $\overline{y'}$, $\overline{x_0}$.

Esto es también preservado para los demás casos de la definición de la función *wlp*.
Para los restantes constructores de acciones, la definición de la función *wlp* es la siguiente:

$$\begin{aligned} wlp[g?, f] &= g \implies f \\ wlp[p_1 + p_2, f] &= wlp[p_1, f] \wedge wlp[p_2, f] \\ wlp[p_1; p_2, f] &= wlp[p_1, wlp[p2, f]] \\ wlp[p^*, f] &= \bigwedge_{i=0}^{\infty} wlp[p^i, f] \end{aligned}$$

Se observa que *wlp* produce fórmulas Alloy en todos sus casos, excepto para el contructor de iteración, donde la fórmula resultante puede ser infinita. Así, para obtener una fórmula Alloy, se tendrá que acotar la iteración.

Esto es equivalente a fijar una longitud máxima para las trazas.
La función Bwlp (bounded weakest liberal precondition) es definida exactamente como *wlp*, excepto la iteración, donde se define como:

$$Bwlp[p^*, f] = \bigwedge_{i=0}^{n} Bwlp[p^i, f]$$

El n es el límite (scope) de la cantidad de la iteraciones.
Ahora se define la función *pcat* que traduce aserciones de corrección parcial a fórmulas Alloy, dada una aserción de corrección parcial $\{\alpha(\overline{y})\}P(\overline{y})\{\beta(\overline{y}, \overline{y'})\}$:

$$pcat(\{\alpha\}P\{\beta\}) = \forall \overline{y}\ (\alpha\ \implies\ (Bwlp\ [p, \beta|\tfrac{\overline{x_0}}{\overline{y}}])\ |\tfrac{\overline{y}}{\overline{y'}}|\tfrac{\overline{y}}{\overline{x_0}})$$

Por supuesto, este método de análisis donde la iteración es restringida a una cota de profundida fija, no es completo, pero esta restricción en el tama no del dominio involucrado en la especificación permite introducir fórmulas de primer orden en fórmulas proposicionales.

3.4. La herramienta: DynAlloy Translator

Dynalloy Translator es la herramieta que realiza traducciones, en forma automática, de modelos Dynalloy a Alloy. Anteriormente se explicó como hacía posible mantener la posibilidad de analizar automáticamente las especificaciones Dynalloy, ahora describiremos como utilizar la herramienta para analizar propiedades.

Supongamos ahora que queremos analizar si una propiedad P es invariante en nuestro sistema. Una técnica conocida para verificar ésto se basa en:

- Probar que P se cumple en los estados iniciales del sistema.
- Probar que, para cada operación O de nuestro sistema, si P vale antes de ejecutar la operación, entonces P vale luego de su ejecución: $P(s)\ \wedge O(s, s') \Rightarrow P(s')$

Utilizando DynAlloy, supongamos que queremos probar que P es invariante respecto de las operaciones $O_1, ..., O_N$ de un cierto modelo:

- Primero se tiene que especificar la propiedad a verificar de la siguiente manera:

$$\begin{array}{l} assertCorrectness\ name\ (s{:}State)\{ \\ \quad pre = \{\ Init(\overline{x}) \wedge P(\overline{x})\ \} \\ \quad program = \{\ O_1 + ... + O_N\} \\ \quad post = \{\ P(\overline{x'})\} \\ \} \end{array}$$

  Intuitivamente lo que se está analizando aquí es que la propiedad P es invariante respecto de las operaciones $O_1, ..., O_N$. Los operadores utilizados en la especificación de la propiedad anterior son los que ya se han definido (figura 3.1).

- El paso siguiente consiste en ejecutar DynAlloy Translator para que genere una especificación Alloy a partir del modelo y la aserción anterior.
- La validación se realizará sobre el modelo Alloy generado, usando el Alloy Analyzer.

A modo de ejemplo, siguiendo la especifiación del **problema de las bolitas**, supongamos que necesitamos validar la propiedad definida por el predicado *atLeastTwoMarbles*(˜existen al menos dos bolitas˝) sobre la siguiente ejecución.

```
 ------ program1---------
 --auxiliar predicates
 pred atLeastTwoMarbles [st: Bag] {
     ((#st.green) + (#st.blue) + (#st.red)) >=2
}


assertCorrectness program1 [s: Bag] {
    pre = {atLeastTwoMarbles [s]}
    program = {
        ( rem1[s] + rem2[s])*
    }
    post = {atLeastTwoMarbles [s'] }
}
```

El programa esta representado por un ciclo (*) de la elección no determinista de las acciones *rem1 y rem2* ya definidas. Una vez hecha la traducción a Alloy, Alloy Analyzer nos devolverá si vale dicha propiedad (*atLeastTwoMarbles*) después de ejecutar n veces las acciones rem1 o rem2 arbitrariamente.

3.4.1. Ventajas

La ventaja más importante que tiene DynAlloy sobre Alloy en la verificación de propiedades de ejecución es que, gracias a la semántica de las acciones y los operadores que nos permiten componerlas, no necesitamos hacer mención explícita de las trazas de ejecución. Además, diversos estudios (véase [12]) mostraron que los modelos DynAlloy requieren un tiempo considerablemente menor que sus pares Alloy a la hora de validar propiedades. Otras de la ventajas de DynAlloy, es que permite especificar de manera sencilla algunas de las construcciones fundamentales de programas iterativos. Las últimas modificaciones soportan las sentencias IF-THEN-ELSE, repetir, y asignaciones:

```
if <pred> {
    S
} else {
    S'
};
```

ciclos:

```
repeat {
   S
};
```

y asignaciones:

```
a:= b;
```

También, facilitado por la inclusión de acciones, nos permite simular de manera sencilla algunas de las construcciones fundamentales de lenguajes orientados a objetos.

Por ejemplo para la creación de un objeto o de una clase C, se puede introducir una acción atómica ($NewC$), especifiacada de la siguiente manera:

$$
\begin{array}{l}
act\ NewC\ (o{:}C) \\
\quad pre = \{\ true\ \} \\
\quad post = \{\ \ o' \ !in\ Objects_C\ and\ o'\ in\ Objects'_C\ \}
\end{array}
$$

corresponde a: $o = new\ C()$;
Donde $Objects_C$ es una relación unaria que contiene el conjunto de los objetos existentes de la clase C.
También, brinda la posibildiad de setear el atributo f de un objeto o, por medio de la siguiente acción:

$$\begin{aligned} &act\ Setf\ (o : C, v : C', f : C \to C') \\ &\quad pre = \{\ o\ in\ Objects_C\} \\ &\quad post = \{\ f' = f + +(o \to v)\} \end{aligned}$$

corresponde a: $o.f = v$;

Por último, DynAlloy permite realizar validación incremental a través de un mecanismo llamado *atomización de programas*[11].

Capítulo 4

Abstracción por Predicados

4.1. Introducción

Los mecanismos algorítmicos de verificación funcionan, generalmente, mediante la construcción y búsqueda del espacio de estados de un programa, en la búsqueda de estados que violen ciertas propiedades. El principal problema, o la principal limitación, asociado con estos mecanismos es el conocido como *problema de la explosión combinatoria de estados*: el espacio de estados de un programa crece de manera combinatoria con el número de componentes del sistema y su estado.

En la búsqueda de técnicas que permitan a los mecanismos de verificación algorítmicos "escalar", es decir, hacerlos aplicables a sistemas más grandes y complejos, han surgido varias alternativas, una de las cuales es la *abstracción*. La abstracción consiste, a grandes rasgos, en hacer al estado del sistema menos detallado, sin perder los elementos esenciales para la descripción de la propiedad de interés (la propiedad a verificar) ni la posibilidad de analizar ésta propiedad. Automatizar la construcción de abstracciones adecuadas es, por supuesto, una actividad extremadamente compleja.

La abstracción por predicados [13] es un mecanismo particular de abstracción, que permite construir automáticamente una abstracción para un sistema dado, a partir de una familia de predicados de estados. Bajo ciertas circunstancias, se puede garantizar que la abstracción construida es *conservativa* con respecto a la verificación de una propiedad determinada, lo cual significa que si la propiedad se verifica en el sistema abstracto también se satisface en el sistema concreto, pero no necesariamente a la inversa.

En este capítulo describiremos abstracción por predicados, ya que nuestra herramienta utiliza esta técnica para el análisis de especificaciones DynAlloy.

4.2. Descripciones Concretas y Abstractas

En este capítulo, haremos referencia con bastante frecuencia a la noción de estado de un programa, el conjunto de estados de un programa, y la relación que, en un programa, permite pasar de un estado a otro. Es decir, interpretaremos programas como grafos dirigidos, donde los nodos representan estados de programa, y los arcos las transiciones de un estado a otro en la ejecución de un programa. También hablaremos de estados abstractos y concretos, pues tendremos dos (o más) versiones del grafo de ejecución de un mismo programa (la versión concreta, y la(s) abstracta(s), donde se elimina detalle).

El conjunto de estados abstractos y concretos pueden ser representados por fórmulas lógicas. Por ejemplo, el predicado concreto X representa el conjunto de estados concretos que satisfacen X. La idea principal de la abstracción por predicados es construir una abstracción conservativa del sistema concreto. Esto asegura que si se verifica una propiedad en el sistema abstracto, entonces dicha propiedad vale en el sistema concreto.

Más precisamente, según [14], el sistema concreto es representado por un conjunto de *estados iniciales* I_C, y una relación de transición R_C. $I_C(x)$ se satisface si y sólo si x es un estado inicial. Por otro lado, $R_C(x, y)$ es verdadera si y sólo si y es un sucesor de x. Una *ejecución* del sistema concreto está definida por

una secuencia de estados $x_0, x_1, .., x_M$, tal que cumple con lo siguiente:

$$I_C(x_0) \wedge \forall i : [0, M) \bullet R_C(x_i, x_i + 1).$$

Sea P una propiedad (una fórmula lógica que caracteriza ciertos estados), que se desea verificar para el sistema. Una traza $x_0, x_1, .., x_M$ corresponde a un *contraejemplo* si $\neg P(x_M)$ (es decir, si la ejecución caracterizada por la traza termina en un estado que viole la propiedad P).

La abstracción es construida a partir de un conjunto de N predicados $\phi_1, \phi_2, .., \phi_N$. Un *estado abstracto* es un vector booleano de longitud N (Cada asignación de valores de verdad para los predicados representa un estado abstracto diferente). La *función de abstracción* α, hace corresponder estados concretos a estados abstractos, mientras que la *función de concretización* γ, hace lo inverso.

Dados Q_C y Q_A, conjuntos de estados concretos y abstractos respectivamente, $\alpha(Q_C)$ es un predicado sobre estados abstractos tal que $\alpha(Q_C)(s)$ se satisface si y sólo si s es la abstracción de algún estado concreto $x \in Q_C$. De igual manera, $\gamma(Q_A)(x)$ vale exactamente cuando existe un estado abstracto $s \in Q_A$ tal que s es la abstracción de x.

Formalmente, α y γ están definidas de la siguiente manera:

- $\alpha(Q_S)(s) = \exists x \bullet Q_C(x) \wedge \bigwedge_{i \in [1,N]} \phi(x) \equiv s(i)$
- $\gamma(Q_C)(x) = \exists s \bullet Q_A(s) \wedge \bigwedge_{i \in [1,N]} \phi(x) \equiv s(i)$

Es posible definir el conjunto de *estados iniciales abstractos* como $I_A = \alpha(I_C)$, y la relación de transición abstracta como $R_A(s,t) = \exists x, y \bullet \gamma(s)(x) \wedge \gamma(t)(y) \wedge R_C(x,y)$.
Una *ejecución abstracta* es una secuencia de estados $s_0, s_1, .., s_M$ tal que se satisfacen:

$$I_A(s_0) \wedge \forall i : [0, M) \bullet R_A(s_i, s_i + 1).$$

Un *contraejemplo abstracto* es una ejecución abstracta $s_0, s_1, .., s_M$ tal que $\alpha(\neg P)(s_M)$. El sistema abstracto se construye como hemos descripto. Si la propiedad a verificar, P, está entre los predicados de abstracción se puede garantizar que si no existen violaciones abstractas de P, no existen violaciones concretas de P (la abstracción es conservativa). Caso contrario debemos examinar los contraejemplos abstractos. Éstos pueden corresponder a trazas concretas (es decir, son concretizables) y por lo tanto constituyen violaciones reales a la propiedad P, o no corresponderse con contraejemplos reales. En este último caso los contraejemplos se denominan *espúreos*. Los contraejemplos espúreos son síntomas de la debilidad de la abstracción construida, y existen algunas técnicas para intentar fortalecerlas.

4.2.1. Contraejemplos Concretizables y Espúreos

Como se mencionó anteriormente, en caso de que la propiedad no sea válida, se genera lo que se denomina un *contraejemplo*. Notemos que existe una traza $x_0, x_1, .., x_L$ que representa un contraejemplo concreto, correspondiente a una traza abstracta $s_0, s_1, .., s_L$, si se satisfacen las siguientes condiciones:

1. $\forall i : [0, L] \bullet \gamma(s_i)(x_i)$. Ésto significa que cada estado concreto x_i, se corresponde con un estado abstracto s_i en la traza.

2. $I_C(x_0) \wedge \neg P(x_L)$. La traza correspondiente al contraejemplo comienza en un estado inicial y termina en un estado que viola P.

3. $\forall i : [0, L) \bullet R_C(x_i, x_{i+1})$. Para todo i, se cumple que x_{i+1} es el sucesor de x_i.

Las condiciones (1) y (3) indican que al menos una traza concreta correspondiente a la traza abstracta efectivamente existe, mientras que la condición (2) determina que la traza comienza a partir de un conjunto de estados iniciales y termina en un estado que viola la condición de verificación.

Formalmente, se dice que un contraejemplo abstracto es real, si y sólo si la siguiente fórmula es válida:

$$\bigwedge_{i\in[1,L]} \gamma(s_i)(x_i) \wedge \bigwedge_{i\in[0,L)} R_C(x_i, x_{i+1})$$

Por otro lado, si la fórmula anterior en insatisfactible, se dice que el contraejemplo es espúreo, pues la traza abstracta no representa a ninguna traza concreta (no es "concretizable").

4.3. Abstracción Para Especificaciones DynAlloy

En el caso particular del presente trabajo, la técnica de abstracción por predicados se realiza sobre especificaciones DynAlloy de manera enteramente automática. Dado un programa especificado en DynAlloy y un conjunto de predicados de estado sobre el mismo, se construye un modelo abstracto que represente dicho programa. Como una restricción, es necesario mencionar que la propiedad que se desea verificar debe estar incluida dentro de los predicados de estado que se utilizan para la abstracción. Entonces el modelo abstracto representa una sobre-aproximación del modelo concreto, y se garantiza que la abstracción es conservativa.

Formalmente, la función de abstracción utilizada, α, se puede definir de la siguiente manera:

Sea φ una fórmula y $\varphi_1, \varphi_2, .., \varphi_N$ el conjunto de predicados de abstracción.

Entonces, $\alpha(\varphi) = [B_1, B_2, .., B_N]$, donde cada B_i es calculada de la siguiente manera:

- $B_i = True$ *sii* $\varphi \Rightarrow \varphi_i$ (φ_i es consecuencia lógica de φ)
- $B_i = False$ *sii* $\varphi \Rightarrow \neg\varphi_i$
- $B_i = *$, otro caso.

Cada $[B_1, B_2, .., B_N]$ representa una familia de estados abstractos (representa exactamente un estado abstracto cuando $\forall i : B_i \neq *$). Dado un vector $[B_1, B_2, .., B_N]$, la función de concretización se define de la siguiente manera:

$\bigwedge_{i\in[1,N]} \psi_i$, tal que,

- $\psi_i = \varphi_i$ *sii* $B_i = True$
- $\psi_i = \neg\varphi_i$ *sii* $B_i = False$
- $\psi_i = True$, otro caso.

Vale la pena destacar que la abstracción aplicada en este proyecto forma parte de las *abstracciones débiles*, es decir que si la propiedad a chequear es válida en el modelo abstracto, entonces la concretización de la propiedad vale en el modelo concreto. Pero por otro lado, la técnica no garantiza la ausencia de contraejemplos espúreos, es decir, que existen trazas abstractas que violan la propiedad, pero que no poseen una traza correspondiente en el modelo concreto. Para eliminar dichas trazas, es necesario aplicar alguna técnica de refinamiento de las abstracciones, la cual se trata en el siguiente capítulo.

Si se ha encontrado un contraejemplo abstracto, debemos comprobar si el mismo se corresponde o no con uno concreto. Para hacer esto, concretizamos la traza abstracta obtenida y comprobamos si la misma es ejecutable o no. Este proceso se hace de manera absolutamente automática, y nos permite discernir entre el caso en el cual hemos encontrado una violación a la propiedad planteada, y el caso en el cual hemos detectado una debilidad en la abstracción.

En la siguiente sección, se introducirá un ejemplo de abstracción para DynAlloy, para dejar en claro la aplicación de la técnica.

4.3.1. Verificando Propiedades Sobre Modelos Concretos y Abstractos

El siguiente ejemplo corresponde a la especificación de un sistema cuya única variable de estado es un conjunto de elementos, es decir, cuyo estado en un momento particular de la ejecución está dado por la forma en la que está constituido un conjunto dado de elementos. Lo primero a introducir es la signatura *Elem*, que representa el tipo de los elementos que almacenan los conjuntos, y dos predicados que representan valores booleanos[1].

```
sig Elem {}

pred TruePred[]{}

pred FalsePred[]{
   not TruePred[]
}
```

A continuación, se introducen los predicados que van a ser de utilidad a la hora de realizar la abstracción, entre los cuales se encuentra la propiedad a verificar.

```
pred empty[s: set Elem] {
  no s
}

pred Negateempty[s: set Elem]{
  not empty[s]
}

pred onePred[s: set Elem] {
  one s
}

pred NegateonePred[s:set Elem] {
  not onePred[s]
}
```

El primer predicado (*empty*) establece que el conjunto s no contiene elementos, mientras que el predicado *one* expresa que el conjunto posee exactamente un elemento. Nótese que para cada predicado *p0*, existe un predicado llamado *Negatep0*. La razón es que para cada predicado de abstracción, es necesaria la negación del mismo para la aplicación de la técnica. Pero para el parser de la herramienta *DynAlloy Translator*, la negación de un predicado es una fórmula (no un predicado), por lo que incluir los predicados "Negate" simplifica la representación y el chequeo de las implicaciones a la hora de construir la abstracción.

Luego, se introducen las acciones junto con los predicados necesarios para expresar pre y post condiciones de las mismas.

```
pred postAdd [s, s': set Elem]{
  some x : Elem-s | s' = s + x
}

act add[s: set Elem] {
  pre {TruePred[]}
  post { postAdd[s,s'] }
```

[1] Es necesario agregarlos ya que son utilizados en la abstracción.

```
}

pred postDel[s, s': set Elem]{
  some x : s | s' = s - x
}

act del[s: set Elem] {
  pre { Negateempty[s]}
  post {postDel[s,s'] }
}
```

La acción *add* agrega un elemento nuevo al conjunto, mientras que la acción *del* elimina del conjunto un elemento que efectivamente existe. Nótese que ésto queda garantizado por el predicado utilizado en la post-condición.

Por último, se introduce el programa al cual se le aplicará la abstracción. El mismo expresa que partiendo de un estado en el cual el conjunto es vacío, y aplicando las acciones *add* y *del* (en ése orden) una vez cada una, se llega a un estado en el cual el conjunto sigue siendo vacío.

```
assertCorrectness program1 [s: set Elem] {
  pre = { empty[s] }
  program = {
    add[s];
    del[s]
  }
  post = { empty[s'] }
}
```

Además de la especificación anterior, se debe contar con los predicados de entrada (ya que de ellos también depende la abstracción). Ésas son las dos entradas necesarias para implementar la técnica. En éste caso dichos predicados son:

```
empty[s]
onePred[s]
```

Ahora es posible aplicar la abstracción, de acuerdo con la técnica explicada en la sección anterior. En primer lugar, se construye la pre-condición abstracta del programa. Ésto se realiza chequeando si la pre-condición concreta implica a cada uno de los predicados de abstracción. Cada vector booleano que representa un estado abstracto está compuesto por dos variables booleanas, donde la primera representa el valor de *empty[s]* en el estado actual, y la segunda representa el valor de *onePred[s]* en el estado actual. Para el programa anterior tendremos tres estados abstractos: el inicial, el intermedio entre *add* y *del* (luego de la ejecución de la acción atómica *add*, y antes de la ejecución de *del*), y el estado abstracto correspondiente al final de la ejecución del programa. Para el caso del primero de estos estados, el mismo de construye de la siguiente manera:

- La primera posición, se conforma chequeando la implicación $empty[s] \Rightarrow empty[s]$. El resultado de la misma es **true**, por lo que se coloca **"T"**.
- En la posición dos, se chequea la implicación $empty[s] \Rightarrow onePred[s]$. Como el resultado no es **true**, es necesario chequear si $empty[s] \Rightarrow NegateonePred[s]$. Como ésta implicación sí vale, se coloca **"F"**.

De ésta manera, la abstracción del estado inicial queda definida como **[T , F]**. Una vez que tenemos el primer estado, se empieza a ejecutar el programa en abstracto.
Ahora, para construir en nuevo estado abstracto luego de la aplicación de la acción *add*, es necesario verificar

si la conjunción entre el estado anterior, la pre-condición y la post-condición de la acción implica a cada uno de los predicados de abstracción[2]. Esta vez, se conforma el vector como sigue:

- **empty**[s] $\wedge$ **NegateonePred**[s] $\wedge$ $TruePred[] \wedge$ `postAdd[s,s']` $\Rightarrow$ $empty[s']$.

  Como esta implicación no vale (introducimos postAdd[s,s'] en el antecedente, se agrega un elemento al conjunto, por lo que empty[s'] no vale), es necesario chequear si vale la negación del predicado:

  empty[s] $\wedge$ **NegateonePred**[s] $\wedge$ $TruePred[] \wedge$ `postAdd[s,s']` $\Rightarrow$ $Negateempty[s']$.

  Como esta implicación sí vale, se coloca **"F"** en la primer posición del estado abstracto.

- **empty**[s] $\wedge$ **NegateonePred**[s] $\wedge$ $TruePred[] \wedge$ `postAdd[s,s']` $\Rightarrow$ $onePred[s']$.

  Esta implicación es válida, por lo que en la segunda posición del vector que representa al estado abstracto se coloca **"T"**.

De ésta manera se conforma el segundo estado de la ejecución, correspondiente a aplicar en abstracto la operación add[s], el cual es el estado abstracto **[F , T]**.

De la misma forma se construye el siguiente estado abstracto, correspondiente a aplicar la operación del[s]. Se aplica el procedimiento anterior, como se explica a continuación:

- **Negateempty**[s] $\wedge$ **onePred**[s] $\wedge$ $Negateempty[s] \wedge$ `postDel[s,s']` $\Rightarrow$ $empty[s']$.

  La implicación es válida, por lo que en la primera posición se introduce **"T"**.

- **Negateempty**[s] $\wedge$ **onePred**[s] $\wedge$ $Negateempty[s] \wedge$ `postDel[s,s']` $\Rightarrow$ $onePred[s']$.

  Al eliminar un elemento a un conjunto con ésa cantidad de elementos, resulta obvio que no vale la implicación anterior, y que se cumple lo siguiente:

  Negateempty[s] $\wedge$ **onePred**[s] $\wedge$ $Negateempty[s] \wedge$ `postDel[s,s']` $\Rightarrow$ $NegateonePred[s']$.

  Por lo tanto, en la segunda posición, se coloca **"F"**.

Luego, el estado abstracto correspondiente es **[T , F]**. Como no quedan más acciones para ejecutar, dicho estado es la post-condición abstracta.

De ésta manera, se puede visualizar en la figura 4.1 la traza abstracta obtenida al ejecutar el programa utilizando abstracción, con las respectivas operaciones que fueron aplicadas. Dicha traza es única, debido a que en el programa concreto no existen elecciones (véase la sección de *Alloy* y *DynAlloy*), o sea no se producen ramificaciones.

[T, F]
`add[s]`
[F, T]
`del[s]`
[T, F]

Figura 4.1: Traza correspondiente a la ejecución abstracta del programa.

Aún resta por hacer lo más importante (o al menos la razón por la cual se implementa esta técnica), que es verificar si la post-condición del programa vale después de ejecutarlo en abstracto. Se chequea la posición del estado abstracto que se corresponde con el predicado a chequear, si es igual a **"T"** se puede afirmar que la post-condición vale, caso contrario se ha encontrado un contraejemplo. En el ejemplo, la post-condición es *empty[s]*, que se corresponde con la primera posición del estado abstracto. En dicha posición existe un

[2]Se resaltan con diferentes estilos de letra los distintos elementos que constituyen cada fórmula: **estado anterior**, *pre-condición* y `post-condición`.

"T", por lo que se puede concluir que como la post-condición vale en el modelo abstracto, es válida también en concreto. Es posible realizar dicha afirmación debido a que la abstracción es conservativa, como se mencionó anteriormente, aunque es posible no llegar a dicha conclusión si la cota propuesta para el chequeo de cada implicación es demasiado pequeña[3].

Se ha presentado en el ejemplo anterior cómo se aplica abstracción a especificaciones escritas en Dynalloy, y se mostró un caso en el cual la propiedad a verificar (la post-condición del programa) es válida luego de ejecutarse el mismo (se verá más adelante que esto significa que al menos hasta cierta cota no se encontraron ejecuciones que violen la propiedad). Para permitir que se comprenda mejor el alcance de la técnica, podemos mostrar un ejemplo en el cual la propiedad es claramente válida, pero no es posible afirmarlo.

Considérese el modelo anterior, y como predicado de abstracción solamente $empty[s]$. Ahora, se aplica la abstracción como en el ejemplo anterior. La pre-condición abstracta se construye de la siguiente manera:

- $empty[s] \Rightarrow empty[s]$. Esto vale, por lo tanto se coloca **"T"**. Como se consideró sólo un predicado de abstracción, el estado queda formado como [**T**].

Luego de aplicar la operación add[s], se procede a construir el estado abstracto correspondiente:

- **empty[s]** $\wedge\, TruePred[] \wedge$ `postAdd[s,s']` $\Rightarrow empty[s']$.

  Esta implicación no es válida, por lo tanto es necesario verificar si vale la negación del predicado:

  empty[s] $\wedge\, TruePred[] \wedge$ `postAdd[s,s']` $\Rightarrow Negateempty[s']$.

  Esta implicación sí vale, por lo tanto el estado abstracto obtenido es [**F**].

Ahora se debe construir el último estado, correspondiente a la aplicación de la operación del[s], la cual además es la post-condición del programa abstracto:

- **Negateempty[s]** $\wedge\, TruePred[] \wedge$ `postDel[s,s']` $\Rightarrow empty[s']$.

  Esta implicación no es válida, se verifica si vale la negación del predicado:

  Negateempty[s] $\wedge\, TruePred[] \wedge$ `postDel[s,s']` $\Rightarrow Negateempty[s']$.

  Esta implicación tampoco vale, luego no es posible afirmar que vale empty[s'], pero tampoco que no vale (o sea, que vale Negateempty[s']). Luego se coloca una **"X"** para representar ésta situación. Por lo tanto la post-condición abstracta es [**X**].

De esta manera, se ha detectado lo que se conoce como contraejemplo, ya que la posición correspondiente a la propiedad a verificar en la post-condición abstracta, es distinta de **"T"**. En este caso, el contraejemplo es espúreo, porque tal como se había mostrado anteriormente, la propiedad debería ser válida luego de ejecutar el programa en el modelo concreto.

El ejemplo presentado al final de esta sección es una claro ejemplo en el cual los predicados de abstracción no son suficientes, lo que prueba que la elección de los mismos es fundamental a la hora de utilizar ésta técnica.

[3]En el presente trabajo, dicha cota es fundamental a la hora de realizar la abstracción, por lo que se la llama cota de abstracción.

4.4. Refinamiento de Abstracciones Basado en Descubrimiento de Predicados

Como se mencionó en el capítulo anterior, muchas veces los predicados para construir una ejecución abstracta no son los adecuados, o son insuficientes para describir las características del sistema concreto. Esto quiere decir, que un gran número de veces luego de realizar la ejecución abstracta, obtenemos un contraejemplo de los llamados espúreos: una ejecución que viola la propiedad en el modelo abstracto, pero que no se corresponde con ninguna ejecución concreta. Mas aún, muchas veces no sabemos a priori cuales son los predicados que sirven para describir los estados del sistema, por lo cual aplicar la técnica de abstracción sería inútil sin una manera de determinar los mismos.
Es por eso, que en el presente trabajo, se implementó una técnica de refinamiento de abstracciones basado en descubrimiento de predicados, basada en [14]. La idea de la misma es agregar información relevante a la hora de describir los estados, y la misma se realiza agregando nuevos predicados de abstracción. De esta manera, no solo se eliminan los contraejemplos espúreos, si no que también es posible construir la abstracción del sistema utilizando solamente la post-condición del mismo como predicado de abstracción.

4.4.1. Descubrimiento de Predicados

En la figura 4.2, podemos observar la representación gráfica de una ejecución abstracta que produce un contraejemplo espúreo. Nótese que los círculos representan conjuntos de estados (correspondientes a $\gamma(s_1), \gamma(s_2)$, etc.) mientras que los puntos en el interior de ellos representan estados individuales.
La traza abstracta $s_1, s_2, ..., s_L$ es espúrea pero la traza parcial $s_2, s_3, ..., s_L$ es real. De esta manera podemos identificar dos tipos de estados concretos en $\gamma(s_1)$:

- Estados que son sucesores de los estados en $\gamma(s_1)$.
- Estados como x_2, que son parte de una traza concreta $s_2, s_3, ..., s_L$.

La idea de la técnica de refinamiento es que si los predicados agregados son aquellos que distinguen los dos tipos de estados que mencionamos anteriormente, entonces los contraejemplos espúreos deberían desaparecer.

Según la técnica presentada en [14] y el ejemplo anterior, como la traza $s_2, s_3, ..., s_L$ es real la siguiente fórmula debe ser satisfacible:

$$Q_0 = \bigwedge_{i \in [2,L]} \gamma(s_i)(x_i) \wedge \bigwedge_{i \in [2,L-1)} R_C(x_i, x_{i+1})$$

Entonces se debería encontrar una fórmula $\Psi_1(x_2) \wedge \Psi_2(x_2) \wedge ... \wedge \Psi_K(x_2) \wedge \theta(x_3, ..., x_L)$ que implica a Q_0, de manera tal que las Ψ_i's sean las condiciones que cualquier x_2 debe satisfacer para ser el primer estado de la traza concreta correspondiente $s_2, s_3, ..., s_L$. Entonces la conjunción:

$$\gamma(s_1)(x_1) \wedge R_C(x_1, x_2) \wedge \bigwedge_{i \in [1,K]} \Psi(x_2) \wedge \theta(x_3, ..., x_L)$$

debería ser insatisfacible, debido a que en caso contrario sería posible encontrar una traza concreta correspondiente a $s_1, ..., s_L$, lo cual como mencionamos anteriormente, no puede darse. Mas aún, si agregamos los predicados $\Psi_1 \wedge \Psi_2 \wedge ... \wedge \Psi_K$ al conjunto de predicados de abstracción y volvemos a realizar una ejecución abstracta, el contraejemplo espúreo correspondiente a esa traza debería desaparecer.

En el caso particular de abstracción para especificaciones DynAlloy, el método descripto anteriormente no es completamente viable. El tamaño de las trazas obtenidas de estas ejecuciones abstractas es bastante elevado, por lo que descubrir nuevos predicados de esta manera resulta bastante ineficiente. Es por este motivo que en el presente trabajo se implementó un mecanismo de refinamiento basado en el anterior, pero con varias modificaciones para adaptarse al tipo de especificaciones que se están utilizando. El procedimiento utilizado para implementar esta idea se detallará en capítulos posteriores.

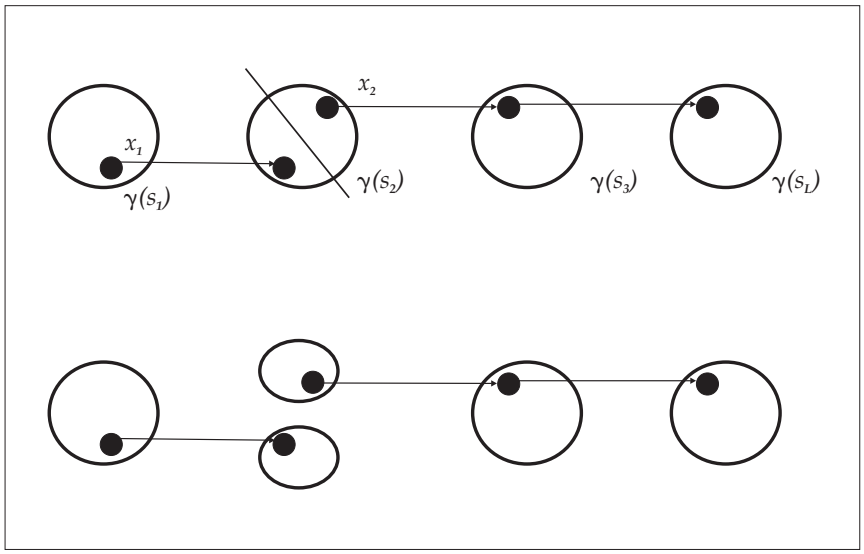

Figura 4.2: Refinamiento de la abstracción.

Capítulo 5

Implementación de Abstracción por Predicados

En los capítulos previos, se introdujo una técnica de abstracción por predicados, junto con los conceptos subyacentes, además de elementos auxiliares utilizados como lo son el lenguaje Alloy y DynAlloy. En el presente capítulo, se introducirá una detallada descripción de la implementación de una herramienta que da soporte a dicha técnica, la cual es el componente principal del presente trabajo.

Objetivos

Los principales objetivos de este trabajo final son:

- Implementar un algoritmo acorde a la técnica de abstracción presentado en [14].
- Utilizar un grafo como estructura de control de la ejecución para lograr la detección de ciclos.
- Optimizar la búsqueda de cálculos previos, evitando así la llamada al SAT Solver (en nuestro caso, Alloy Analyzer).
- Implementar un módulo que refine abstracciones y descubra nuevos predicados acorde a [14].
- Inclusión de un *timeout* en la GUI, para limitar el tiempo empleado en las ejecuciones.
- Actualización de la técnica de abstracción acorde a la nueva sintaxis de DynAlloy (en particular, se pueden especificar asignaciones).

Alcance de la Herramienta

Los mecanismos formales automáticos de análisis de sistemas o especificaciones de sistemas sufren del problema de la explosión combinatoria: la complejidad de las técnicas, en tiempo y utilización de recursos, crece exponencialmente con la complejidad de los sistemas o especificaciones a analizar. Esto tiene como consecuencia que, en general, para que estos análisis puedan ser utilizados en la práctica deben implementarse técnicas que mejoren la aplicación de estos análisis. La composicionalidad de las propiedades es una pieza clave a la hora de pensar en eficiencia, debido a que es posible explotar la estructura modular de los sistemas. Esto no es una característica exclusiva de los sistemas: las especificaciones de sistemas suelen tener una organización modular, que también puede aprovecharse.

Una de las características más importantes en los mecanismos de verificación y validación de software más populares en la actualidad (principalmente en Model Checking y SAT-Solving) es su absoluta automatización. Como era de esperar, esto tiene algunas restricciones, como la aplicación sólo a modelos finitos, y el problema

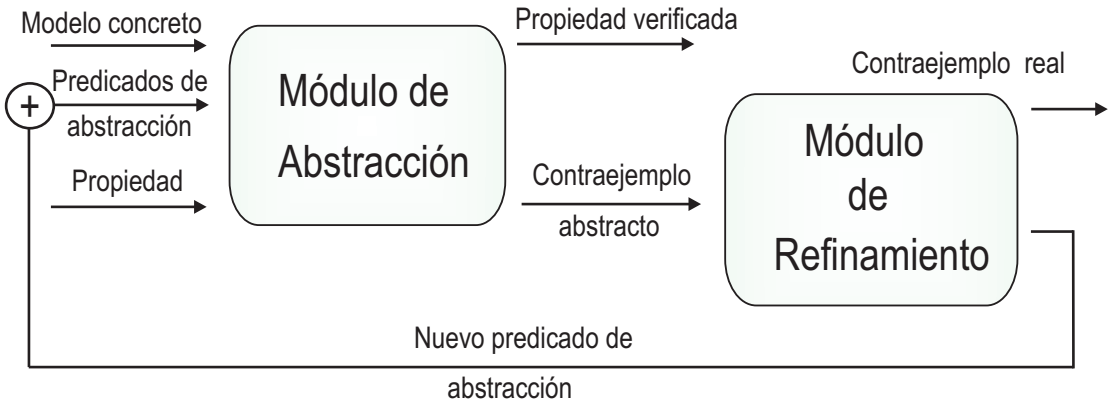

Figura 5.1: Algoritmo de Abstracción por Predicados.

de la explosión combinatoria de estados. Sin embargo, es posible complementar estas técnicas automatizables con otras que no lo son, como deducción asistida. Una técnica que permite sortear el problema de la explosión combinatoria de estados es la abstracción por predicados (véase capítulo 4). Dicha técnica está basada en interpretación abstracta, es decir, que dado un modelo concreto, se construye una abstracción del mismo (es decir, un modelo más simple que el anterior, con una menor cantidad de estados) y las propiedades a verificar sobre el modelo concreto se verifican sobre el modelo abstracto. Si la propiedad vale en el modelo abstracto, es posible afirmar que vale en el modelo concreto, pero no necesariamente a la inversa.

5.1. Algoritmos de Abstracción

La herramienta está basada en el algoritmo de abstracción presentado en [14], y que se describe gráficamente en la figura 5.1. La idea esencial es la siguiente: dado un modelo concreto (especificado en DynAlloy), un conjunto de predicados de abstracción y una propiedad a verificar, se computa un modelo abstracto aproximado del modelo concreto, utilizando los predicados de abstracción y la propiedad a verificar. Este modelo abstracto es chequeado (en la herramienta implementada se utiliza Alloy Analyzer) en busca de contraejemplos. El proceso termina cuando la propiedad vale (para cierta cota), o se encuentra un contraejemplo real. En el caso en que se encuentra un contraejemplo espúreo se utiliza el módulo que refina ejecuciones abstractas y descubre nuevos predicados de abstracción, para luego realizar nuevamente la ejecución abstracta del modelo, en este caso con un nuevo predicado de abstracción.

A continuación se presenta en notación pseudo-código la idea de los algoritmos de las técnicas de abstracción que se desarrollaron en este proyecto final.

5.1.1. Ejecución Bajo Demanda

El algoritmo al que llamamos **OnDemandAbstractor**, es el más simple que desarrollamos y fue el punto de partida para obtener los demás algoritmos. En el cuadro 5.1 se puede visualizar dicho algoritmo.

```
1  -- ondemand
2  Type enum TraceType =  { T, F, X }
3
4  TraceType abstractExecution ( l:List<DynAlloyProgram>,
5                                preds: List<DynAlloyPredicate>,
6                                currst: AbstractState
7                              )
8  {
9    if(l.empty()){
10     return currst.getBit(postBit);
11   }
12   else{
```

```
13      //let l = p:ls
14      DynAlloyProgram p = l.pop();
15      if(p instanceof Action){
16        return abstractExecution(l,preds,AbsFunction(p,currst));
17      }
18      else if (p instanceof TestPredicate){
19        return abstractExecution(l,preds,AbsFunction(p,currst));
20      }
21      else if (p instanceof Composition(p1,p2)){
22        return abstractExecution(p1:p2:l,preds,currst);
23      }
24      else if (p instanceof Choice(p1,p2)){
25        TraceType e = abstractExecution(p1:l,preds,currst);
26        if ( e = T )
27          return abstractExecution(p2:l,preds,currst);
28        else
29          return e;
30      }
31      else if (p instanceof Closure(c,n){
32        return abstractExecution(c;...;c:l , preds, currst); // c n times
33      }
34    }
35  }
```

Cuadro 5.1: OnDemandAbstractor : Algoritmo de Abstracción que ejecuta trazas bajo demanda.

- Los casos bases de la recursión son la ejecución de una acción o un test.
- Obsérve que cuando se ejecuta un Choise, primero se ejecuta una de las ramas y en el caso en el que no se encuentre contraejemplo, se ejecuta la otra rama (asi evita generar trazas innecesariamente). De esta manera las trazas son ejecutadas bajo demanda.
- El el caso de la composición lo que se hace es descomponerla, e introducir los comandos que la componen en orden dentro de la lista.
- Cuando se trata de una clausura, se introduce el cuerpo de ésta dentro de la lista n veces.

5.1.2. Ejecución Bajo Demanda + Refinamiento de Abstracciones

Luego de contar con un módulo de refinamiento en la herramienta, el OnDemandAbstractor fue modificado (o actualizado) para que permita refinar abstracciones: cuadro 5.2.

```
1   -- ondemand with refinement
2   // pre: l = l_bkp
3   Type enum TraceType =  { T, F, X }
4   Type Trace{
5     DynAlloyProgram [] operations,
6     AbstractState [] states
7   }
8   Boolean abstractExecution ( l,l_bkp:List<DynAlloyProgram>,
9                               preds: List<DynAlloyPredicate>,
10                              initst, currst: AbstractState,
11                              currtr:Trace
12                            )
13  {
14    if(l.empty()){
15      if (currst.getBit(postBit)==T)
16        return true;
17      else{
18        if(isSpurious(currtr)){
19          List<DynAlloyPredicate> newPreds = refineAbstraction( currtr );
20          return abstractExecution(l_bkp, l_bkp, preds.union(newPreds) , initst, initst, new Trace↩
                ());
21        }
22        else
23          return false;
```

```
24      }
25    }
26    else{
27      //let l = p:ls
28      DynAlloyProgram p = l.pop();
29      if(p instanceof Action){
30        AbstractState newst = AbsFunction(p,currst);
31        currtr.add(p,newst);
32        return abstractExecution(l,l_bkp,preds,initst,newst,currtr);
33      }
34      else if (p instanceof TestPredicate){
35        AbstractState newst = AbsFunction(p,currst);
36        currtr.add(p,newst);
37        return abstractExecution(l,l_bkp,preds,initst,newst,currtr);
38      }
39      else if (p instanceof Composition(p1,p2)){
40        return abstractExecution(p1:p2:l,l_bkp,preds,initst,currst,currtr);
41      }
42      else if (p instanceof Choice(p1,p2)){
43        Trace tr1 = currtr.copy();
44        Trace tr2 = currtr.copy();
45        if (!abstractExecution(p1:ls, l_bkp, preds, initst, currst, tr1 ))
46          return false;
47        else
48          return abstractExecution( p2:ls, l_bkp, preds, initst, currst, tr2 )
49      }
50      else if (p instanceof Closure(c,n){
51        return abstractExecution(c;...;c:l ,l_bkp,preds,initst,currst,currtr); // c n times
52      }
53    }
54  }
```

Cuadro 5.2: OnDemandAbstractor + RefinementModule.

- A diferencia del algoritmo presentado en la sección 5.1.1, este algoritmo permite refinar abstracciones.
- El el caso en que se detecta un contraejemplo espúreo, se utiliza el módulo de refinamiento el cuál agrega nuevos predicados de abstracción. Luego se comienza nuevamente el procedimiento de abstracción.
- La ejecución termina cuando la propiedad vale o se encuentra un contraejemplo real.

5.1.3. Detección de ciclos

La inclusión de esta técnica en el algoritmo de abstracción presentado en la figura 5.1 puede ser considerada la "*mejor*"optimización realizada en este trabajo final. Una de las construcciones sintácticas predominantes en modelos DynAlloy es el operador de *clausura* o iteración (acotada). El programa `(c)*` indica que los comandos que aparecen en `c` se van a ejecutar al menos **n** veces, donde n es un parámetro de entrada que indica la cota para iteraciones. Por lo tanto si n es 3, `(c)* = c;c;c`.

Para comprender la motivación de la detección de ciclos considere que el programa $c* = (c_1 + c_2)*$, el cual contiene una elección no determinística (choice). Si el programa `c` se ejecuta una sola vez, serán dos trazas las que determinen el comportamiento de `c*` (c_1 y c_2). Si la cota de iteración es 2, las trazas posibles son cuatro ($c_1;c_1$, $c_1;c_2$, $c_2;c_1$, $c_2;c_2$) y si la cota es 3 las trazas posibles son ocho, y así sucesivamente. Por lo tanto, al haber elecciones no determinísticas dentro de iteraciones, el número de trazas crece exponencialmente (respecto al número de elecciones).

El objetivo de la detección de ciclos es evitar ejecutar todas iteraciones, indicadas por un parámetro, de no ser necesario. Para comprender esta idea, observe el siguiente ejemplo:

- Considere el modelo de *Set* presentado en la sección 4.3.1 con sus respectivas acciones de agregar y quitar elementos (add y del, respectivamente).
- Sea el siguiente programa DynAlloy:

```
assertCorrectness programExample [s: set Elem]{
  pre = { empty[s] }
  program = {
   (add[s];del[s]+skip)*
  }
  post = { empty[s'] }
}
```

- El número de trazas posibles es 2^n, donde n es la cantidad de iteraciones.
- Acorde a la especificación de las acciones add y del que aparecen en la sección 4.3.1, ocurre que ejecutar `add[s];del[s]` es lo mismo que ejecutar `skip`.
- Por lo tanto, después de realizar una iteración se puede observar que no importa que rama tome en la ejecución de la clausura, siempre me va a llevar al estado en que comenzo a ejecutarse.
- Luego se *detecta un ciclo* sobre `(add[s];del[s]+skip)*` y no es necesario que se ejecute n veces.
- En este caso basta con iterar una vez para detectar ciclos, en otros casos puede ocurrir que se necesiten más iteraciones y en otros casos puede que la iteración nunca se estabilice sobre un estado.
- Para implementar esta técnica utilizamos un grafo como estructura de control para la ejecución. El grafo resultante de la ejecución del ejemplo :

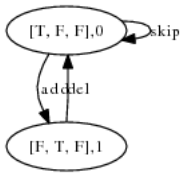

Figura 5.2: Grafo resultante de la ejecución de programExample.

5.1.4. Ejecución Bajo Demanda + Refinamiento de Abstracciones + Detección de Ciclos

El algoritmo al que hacemos referencia como **AbstractorWithGraph** utiliza un grafo como estructura de control de la ejecución, facilitando la detección de ciclos (sección 5.1.3) y así optimizar (de gran manera) el tiempo empleado para la ejecución abstracta del modelo DynAlloy. El algoritmo es presentado en el cuadro 5.3.

```
1  -- graph with refinement
2  Type LNode {
3    AbstractState state,
4    Label label
5  }
6
7  Type LEdge {
8    LNode source,
9    DynAlloyProgram action,
10   LNode target
11 }
12
13 Type Graph {
14   SetOf(LNode) nodes,
15   SetOf(LEdge) edges
16 }
17
18 Type DynLabeledProgram {
```

```
    DynAlloyProgram program,
    Label initLabel,
    Label lastLabel
}

Type Trace{
  DynAlloyProgram [] operations,
  AbstractState [] states
}

Graph g = new Graph(SetOf(LNode(AbsFunction(preCondition),0)),EmptySet())
// pre: l = l_bkp
Boolean abstractExecution ( l, l_bkp:List<DynLabeledProgram>,
                            preds: List<DynAlloyPredicate>,
                            initNode, currNode: LNode,
                            currtr:Trace
                          )
{
  if(l.empty()){
    if (currst.getBit(postBit)==T)
      return true;
    else{
      if(isSpurious(currtr)){
        List<DynAlloyPredicate> newPreds = refineAbstraction( currtr );
        return abstractExecution(l_bkp, l_bkp, preds.union(newPreds) , initst, initst, new Trace↩
            ());
      }
      else
        return false;
    }
  }
  else{
    //let l = p:ls
    DynAlloyProgram p = l.pop();
    if(p instanceof Action){
      AbstractState newst = AbsFunction(p, currst);
      LNode newNode = new LNode(newst,p.getLastLabel());
      g.addNode(newNode);
      g.addEdge(currNode, p, newNode);
      currtr.add(p,newst);
      return abstractExecution(l,preds,initNode,newNode,currtr);
    }
    else if (p instanceof TestPredicate){
      AbstractState newst = AbsFunction(p, currst);
      LNode newNode = new LNode(newst,p.getLastLabel());
      g.addNode(newNode);
      g.addEdge(currNode, p, newNode);
      currtr.add(p,newst);
      return abstractExecution(l,preds,initNode,newNode,currtr);
    }
    else if (p instanceof Composition(p1,p2)){
      return abstractExecution(p1:p2:l,preds,currst);
    }
    else if (p instanceof Choice(p1,p2)){
      Trace tr1 = currtr.copy();
      Trace tr2 = currtr.copy();
      if (!abstractExecution(p1:ls, l_bkp, preds, initst, currst, tr1 ))
        return false;
      else
        return abstractExecution( p2:ls, l_bkp, preds, initst, currst, tr2 )
    }
    else if (p instanceof Closure(c,n){
      LNode same_currentNode = null;
      for(LNode v : g.getNodes()){
        if(v!=currVertex and v.getState()==currNode.getState() and v.getLabel()==currNode.↩
            getLabel()){
          same_currentNode = v;
          break;
        }
      }
      if(same_currentNode!=null)
        return abstractExecution(l,l_bkp, preds,initNode,v,currtr);
      else
        return abstractExecution(c:Closure(c,n-1);l,l_bkp, preds,initNode,v,currtr);
    }
  }
}
```

Cuadro 5.3: Algoritmo de Abstracción utilizando un grafo como estructura de control.

- Este algoritmo también realiza una ejecución bajo demanda.
- La idea es generar un grafo que representa la ejecución, nos permite la detección de ciclos (sección 5.1.3).
- La detección de ciclos se da en la clausura y permite evitar ejecutar el cuerpo de esta de no ser necesario.
- En casos de contraejemplos espúreos, se utiliza el módulo de refinamiento para descubrir nuevos predicados de abstracción.
- La ejecución termina cuando la propiedad vale o se encuentra un contraejemplo real.

5.2. Almacenamiento de Cálculos Previos

A partir del nuevo diseño, se mejoraron algunas implementaciones con respecto a la primera versión de la herramienta, haciendo hincapié sobre la eficiencia. Se desarrolló una nueva implementación del módulo ActionManager, que es el encargado de guardar los cálculos previos de la ejecución abstracta, . Para tratar de minimizar el tiempo de la búsqueda de estados, debido a que se lo utiliza repetidamente y en una gran cantidad de veces, se utilizaron estructuras de datos eficientes ya implementadas por las librerías de Java (*java.util*). En un primer lugar, para guardar las acciones se utilizó la clase *HashMap*, la cual brinda una búsqueda prácticamente constante sobre sus elementos, y para almacenar los estados abstractos se utilizó la clase *LinkedLink*. Además se reorganizó la forma de introducir los estados abstractos, de tal manera que su búsqueda sea mas eficiente. Esta nueva organizacón permite recorrer menos estados para poder encontrar *el mejor* que la satisfaga.

5.3. Refinamiento de Abstracciones

En este trabajo final se propuso como objetivo la implementación de un módulo que permita refinar abstracciones. Este módulo respeta la técnica mencionada en la sección 4.4 y en el artículo bibliográfico [14].

El proceso de chequear espureidad es altamente costoso, implica invocar al Alloy Analyzer y verificar si una traza dada es un contraejemplo espúreo o no. En el proceso de abstracción, las invocaciones a Alloy permiten calcular cada estado abstracto de la ejecución y podrían considerarse ˜menos costosas ″(obviando que la invocación es un problema NP-Completo) que las invocaciones para chequear espureidad. Esto es visible en la amplia diferencia en el tamaño de las cotas, tanto para abstracción como para verificación de espureidad.

Suele ocurrir que es una de las primeras acciones en una traza la que introduce espureidad a la misma. Por lo tanto, en estos casos, sería ineficiente chequear toda la traza, bastaría con observar sólo una parte. En los casos en que la traza es un contraejemplo real, es inminente el chequeo completo de la traza, pero como resultado se encontraría un error en la especificación de nuestro sistema.

Sea una traza $\{s_0\}a_0\{s_1\}\ldots\{s_n\}a_n\{s_{n+1}\}$, donde s_i es el i-ésimo estado abstracto y a_i es la i-ésima acción ejecutada. El chequeo de espureidad se realiza de manera incremental, es decir, la primer traza parcial que se considera es $\{s_n\}a_n\{s_{n+1}\}$, la segunda es $\{s_{n-1}\}a_{n-1}\{s_n\}a_n\{s_{n+1}\}$, y así sucesivamente, pero partiendo del final de la traza (a diferencia de [14]). De ésta manera no sólo se optimiza el chequeo de espureidad, sino que además se obtiene la mínima traza espúrea necesaria para descubrir un nuevo predicado de abstracción tal como se menciona en la sección 4.4.

La técnica, en casos de espureidad, para descubrir un nuevo predicado requiere que las acciones que componen la traza sean inversibles, es decir, que se puedan escribir usando sólo asignaciones y tests. A pesar de ser una restricción importante, no es un inconveniente para este trabajo final, debido a que se considera

que los modelos con los que se trabajará derivan de código Java. Luego, descubrir un nuevo predicado de abstracción se resume a la tarea de calcular la *weakest precondition* sobre la mínima traza espúrea encontrada.

Cálculo del wp (weakest precondition) sobre un comando DynAlloy:

- $wp.(x := E).\{R\} = R[x \backslash E]$
- $wp.[P]?.\{R\} = R \wedge P$

Continuando con la traza $\{s_0\}a_0\{s_1\}\ldots\{s_n\}a_n\{s_{n+1}\}$, a modo de ejemplo consideremos que la traza parcial $\{s_{n-1}\}a_{n-1}\{s_n\}a_n\{s_{n+1}\}$ es espúrea. El primer paso es construir el siguiente predicado:

$$P = wp.(a_{n-1}; a_n).s_{n+1} = wp.a_{n-1}.(wp.a_n.s_{n+1})$$

Dicho predicado constituye uno de los nuevos predicados de abstracción, pero el procedimiento no termina aquí: se construye otro predicado de la siguiente manera:

$$Q = wp.a_{n-2}.P$$

De esta manera consideramos que la acción que hace la traza espúrea es la anterior al estado que queremos refinar (en el ejemplo anterior la acción a_{n-2} es quien introduce espureidad a s_{n-1}). Finalmente, los predicados P y Q son agregados al conjunto de predicados de entrada y el procedimiento de abstracción se vuelve a ejecutar con esta nueva información de los estados.

Vale la pena mencionar dos características de este mecanismo:

- la primera es que para que los predicados que se descubren sean más "finos", se separan las conjunciones en predicados individuales, es decir que si $P = P_1 \wedge P_2 \wedge \ldots \wedge P_n$ y $Q = Q_1 \wedge Q_2 \wedge \ldots \wedge Q_m$, los nuevos predicados de abstracción serán todos los P_i y los Q_i . Esto sirve para obtener información mas precisa de los estados que queremos refinar. En caso de que los nuevos predicados decubiertos no agreguen información porque la ya eran predicados de abstracción, utilizamos una heurística que consiste en seguir refinado considerando las acciones anteriores a la traza minimal hasta que se encuentre un nuevo predicado de abstracción. En nuestro ejemplo calcularíamos $R = wp.a_{n-3}.Q$, si R no contiene un nuevo predicado, se continua con $R' = wp.a_{n-4}.R$, y asi sucesivamente.
- La segunda es que el proceso de refinar la abstracción continúa mientras se encuentren trazas espúreas (por lo tanto, puede ciclar infinitamente o hasta que expire el timer) o se recorra la traza completa y no se obtenga nueva información.

5.4. Estructura y Diseño

La estructura principal de la herramienta puede reducirse a los siguientes módulos:

- *AbstractState*: Representa un determinado estado abstracto. Simplemente contiene un vector de valores booleanos (o *"X"*).
- *Trace*: Representa una traza abstracta. Contiene una secuencia de estados abstractos y de operaciones, tal como se mostró en la figura 4.1.
- *Abstractor*: Interface que define operaciones comunes entre los diferentes algoritmos de abstracción. Los diferentes algoritmos utilizan a los demás módulos para realizar el proceso de abstracción.
- *OnDemandAbstractor*: Es la implementación de un algoritmo de abstracción, el cuál genera solamente una traza y realiza su ejecución abstracta. En caso de que dicha ejecución no viole la propiedad a verificar, se genera una nueva traza y se la ejecuta. Se continúa el proceso hasta que se hayan generado todas las trazas, o se viole la propiedad.

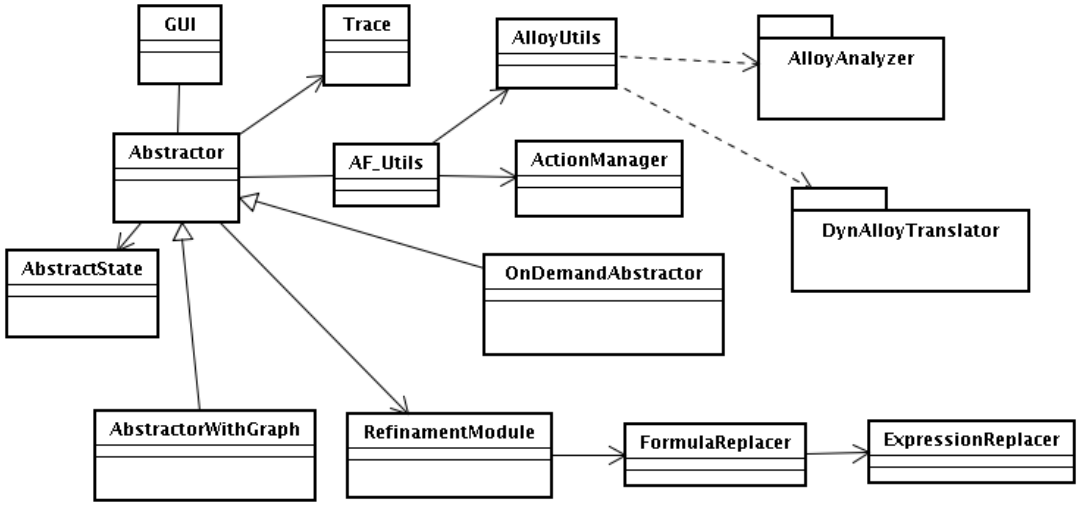

Figura 5.3: Diagrama de Clases Simplificado de la Herramienta.

- *AbstractorWithGraph*: Es otra implementación de un algoritmo de abstracción, el cuál está basado en el algoritmo OnDemandAbstractor. AbstractorWithGraph hace uso de un grafo como estructura de control de la ejecución, de esta manera facilita la detección de ciclos y nos brinda un elemento gráfico para observar el avance de la ejecución.
- *AbstractorGUI*: Es la implementación de la interfaz de usuario, la cual es la encargada de recibir las entradas que debe proveer el usuario e invocar a Abstractor.
- *AF_Utils*: Define las funciones de abstracción y de concretización.
- *AlloyUtils*: En este módulo se implementó una función que verifica si un contraejemplo es real o espúreo, y provee una rutina de invocación a Alloy Analyzer (necesaria para la abstracción y para el chequeo de espúreo).
- *ActionManager*: Módulo que almacena resultados de transiciones realizadas previamente, y provee un fácil acceso a dicha información, que será utilizada para evitar invocaciones a Alloy Analyzer, mejorando la eficiencia de la abstracción.
- *RefinementModule*: Este módulo permite chequear si el contraejemplo encontrado es espúreo o real y en el caso de espureidad, refinar la abstracción y descubrir un nuevo predicado para agregarlo a los predicados que sirven de entrada para el algoritmo de abstracción elegido.
- *FormulaReplacer*: Visitor utilizado para realizar el cálculo de la pre-condición más débil. El mismo se realiza de manera sintáctica.
- *ExpressionReplacer*: Visitor que tiene el mismo propósito que el módulo anterior, pero trabaja sobre expresiones en vez de fórmulas. Ambos son utilziados por el módulo de refinamiento.

También vale la pena mencionar, que fue necesaria la implementación de un pequeño parser (utilizando gramáticas *ANTLR*), el cual verifica que los predicados de abstracción que el usuario ingresa están bien formados. De la verificación del módulo concreto se encargan las herramientas *DynAlloy Translator* y *Alloy Analyzer*.

En la figura 5.3 puede visualizarse una simplificación del diagrama de clases de la herramienta.

Como se mencionó en capítulos anteriores, al existir invocaciones a Alloy, es necesario fijar las cotas para las mismas. Se deben especificar dos cotas, una correspondiente a las invocaciones para construir el

modelo abstracto, y la otra al chequeo de contraejemplos espúreos. Sin embargo, estas cotas pueden ser demasiado pequeñas, es decir que se puede obtener como resultado que ciertas aserciones son válidas, cuando en realidad tienen un contraejemplo de tamaño mayor, lo que producirá un modelo abstracto impreciso, con propiedades inválidas en abstracto que son válidas en concreto. Por otra parte, aumentando el tamaño de las cotas produciría un aumento considerable en los tiempos de análisis, ya que Alloy Analyzer utiliza un algoritmo de análisis exponencial con respecto al tamaño de sus dominios. En conclusión, el tamaño de las cotas depende de cada caso particular, y no hay una forma de determinar un tamaño óptimo.

5.5. Uso de la Herramienta

Luego de introducir detalles de implementación, es necesario describir conceptos un poco más interesantes desde el punto de vista del usuario. Por lo tanto se comenzarán por explicar brevemente las consideraciones más importantes que se deben tener en cuenta para la utilización de la herramienta.

5.5.1. Carga de Parámetros

La herramienta cuenta con una interfaz, que permite cargar toda la información necesaria para aplicar abstracción, de una manera simple y cómoda. La misma puede visualizarse en la Figura 5.4.

Como se puede ver, en la parte izquierda de la interfaz, se encuentran todos los campos a llenar. Lo primero que aparece es la parte de carga de archivos, en la sección $\langle Files \rangle$. En ésta parte se deben cargar el módulo DynAlloy y los predicados de abstracción, en los campos $\langle DynAlloyModules \rangle$ y $\langle PredicatesModules \rangle$ respectivamente. Presionando el botón *Add* respectivo a cada campo, se abre un explorador de archivos que permite seleccionar los archivos deseados. Por otra parte, ambos campos cuentan con un botón *Remove*, para quitar los módulos. Además, una vez cargado los archivos, presionando el botón *Edit*, se ejecuta un editor de texto, el cual permite realizar modificaciones.

Una vez elegidos el módulo y los predicados de abstracción, se debe proceder a completar las opciones que se encuentran en el panel $\langle Options \rangle$. El nombre del módulo (*Module Name*), es el nombre del módulo DynAlloy cargado. Cabe destacar, que debido a que así lo exige DynAlloy Translator, dicho nombre debe ser el mismo que el nombre del archivo. En el campo *Assertion To Check*, se debe indicar el nombre de la aserción DynAlloy que se va a chequear, es decir, el programa a ejecutar. Los cuatro campos siguientes, corresponden a las cotas necesarias para chequear una aserción:

- *Loop Unroll*: Es la cota para los ciclos del programa.
- *Abstraction Scope*: Es la cota para la búsquedas de instancias, utilizadas en la abstracción.
- *Int Scope*: Es la cota para las instancias del tipo *int*.
- *Check Spurious Scope*: Es la cota para búsquedas de instancias en la aserción que se construye para determinar si un contraejemplo es real o espúreo.

A continuación se encuentra una casilla de verificación, para indicar si queremos que una vez que se encontrón un contra-ejemplo se detenga la ejecución o no [1]. Esto tiene sentido debido a que podemos desear construir todas las posibles trazas. Debajo hay un menú desplegable que permite elegir que algoritmo se quiere ejecutar: *On demand* (ejecuta el algoritmo bajo demanda) o *With graph* (algoritmo que construye un grafo durante la ejecución). Por último, en la parte inferior de la interfaz, se encuentra en campo *Time out*, que indica el tiempo que va a transcurrir antes de que la ejecución se corte automáticamente (una hora por defecto).

[1] Nota: esto solo tiene sentido cuando se ejecuta el algoritmo con grafo, debido a que es el único que almacena las trazas ejecutadas, y visualizarlas en forma de grafo

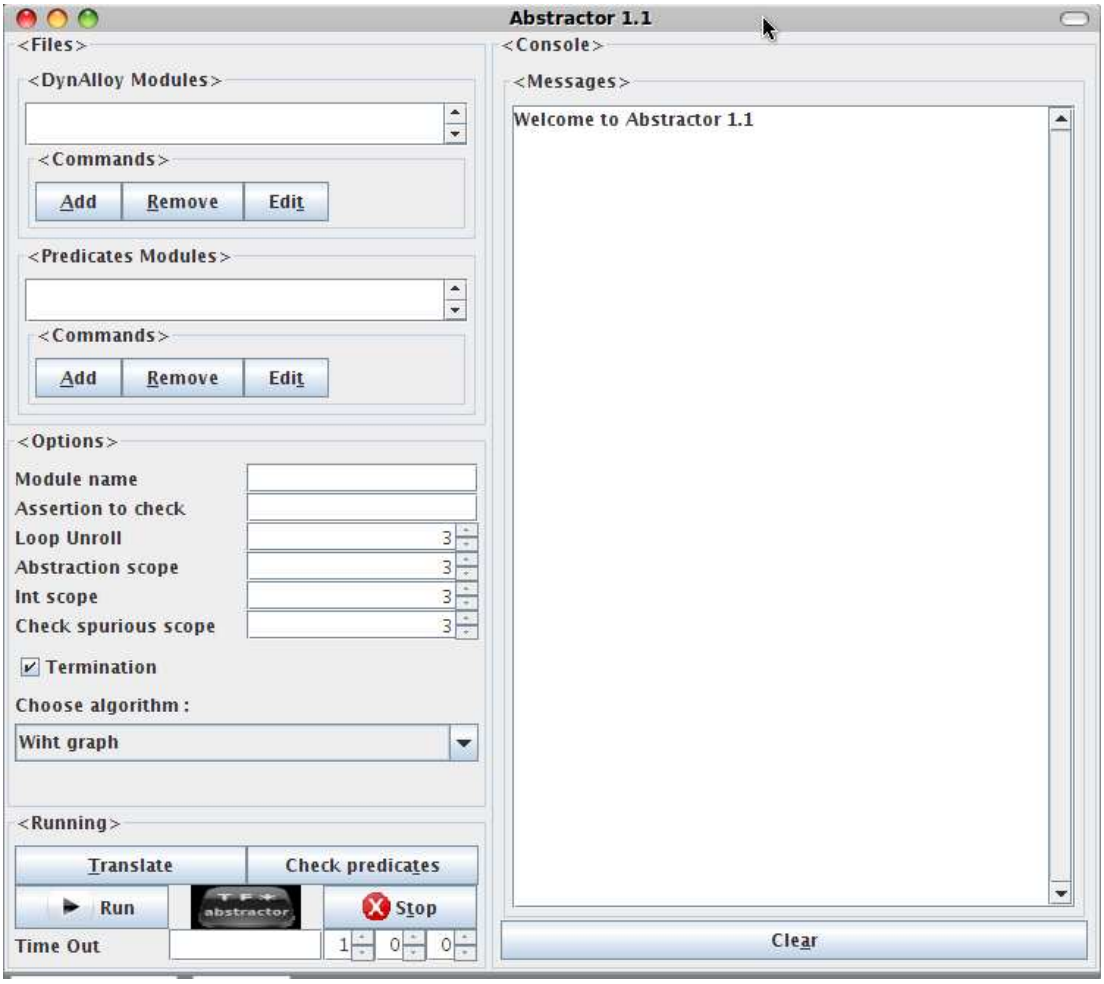

Figura 5.4: Interfaz de la Herramienta.

5.5.2. Ejecución

Una vez que se llenaron todos los campos anteriores, es posible proceder a ejecutar la abstracción. El panel ⟨*Running*⟩, contiene cuatro botones: *Translate, Check Pedicates, Run* y *Stop*. Translate invoca a DynAlloy Translator para traducir el módulo de DynAlloy a Alloy, mientras que *Check Predicates* se encarga de invocar al parser de los predicados de abstracción. Estas dos acciones no son siempre necesarias, debido a que presionando el botón *Run*, estas dos acciones también se ejecutan, además de comenzar a correr la abstracción con la configuración obtenida. Cuando se termina la abstracción y el cálculo de resultados, se ejecutará un editor de textos mostrando las trazas obtenidas. En caso de que la herramienta descubra nuevos predicados, se irá ejecutando el editor de texto mostrando la traza correspondiente a la ejecución incluyendo los nuevos predicados de abstracción, recordando que este proceso puede no terminar. En caso de que expire el timer, la ejecución será interrumpida. Por último, el botón *Stop* sirve para detener la ejecución antes de que finalice.

5.5.3. Interpretación de resultados

Ya fueron introducidos los pasos necesarios para la utilización de la herramienta, pero resta describir cómo interpretar los resultados que brinda la interfaz de usuario. En primer lugar se debe mencionar el comportamiento de la herramienta cuando no se produce ningún tipo de error en la ejecución. Lo primero que aparecerá en la consola, son los siguientes mensajes:

```
Translation successful.
Checking syntax of input predicates --> OK.
Running .....
```

Dichos mensajes indican que la traducción de DynAlloy a Alloy fue satisfactoria y que la sintaxis de los predicados de abstracción es correcta; finalmente la herramienta informa que ha comenzado la ejecución del proceso de abstracción. Luego de transcurrido un tiempo, se informarán los resultados de la corrida. Pueden ocurrir tres cosas: que no se encuentren contraejemplos para la propiedad, que se encuentre un contraejemplo real, o que se encuentre un contraejemplo espúreo. En el primer caso, se imprimirá un mensaje como el siguiente mensaje, indicando la corrección parcial de la propiedad (es decir que la propiedad vale para cierta cota), y el tiempo que demandó la corrida:

```
===== Results of the run =====

No counterexample found. Predicate pred_name is valid within the provided bounds.
Time of execution : 0 mins 1 secs
```

Por otra parte, si se encontró un contraejemplo real, el mensaje que la herramienta mostrará en la consola será el siguiente:

```
===== Results of the run =====

***** Counterexample found is not spurious ****
@ See the Counterexample in: "filename.als"
@ Trace number 1 in file file-traces
Time of execution : 0 mins 1 secs
```

En el caso de encontrar un contraejemplo espúreo, se podrá visualizar el siguiente mensaje, pero la herramienta continuará su ejecución en busca de nuevos predicados de abstracción (aunque se puede cortar la ejecución presionando el botón *Stop*):

```
===== Results of the run =====
```

```
***** Spurious Counterexample found *****
@ See the Counterexample in: "filename.als"
@ Trace number 1 in file file-traces
Time of execution : 0 mins 1 secs
```

En caso de encontrar un contraejemplo, ya sea real o no, es posible realizar una depuración del mismo. De hecho, ésa es la finalidad del archivo `filename.als` que se introduce en el mensaje, dado que corriendo la especificación almacenada en dicho archivo usando Alloy Analyzer, es posible visualizar el contraejemplo. Pero si al usuario sólo le interesa conocer la traza abstracta, la herramienta almacena la misma en el archivo `file-traces` especificado en el mensaje.

Por último, cualquiera sea el resultado del proceso de abstracción, la herramienta lanzará un editor de textos mostrando las trazas obtenidas en la corrida.

Los escenarios que acabamos de describir corresponden a los escenarios de uso exitosos de la herramienta. En caso de que existan problemas con los parámetros de entrada de la aplicación, sean estos archivos de especificaciones, o campos no completados adecuadamente, la herramienta informará sobre el problema. Por ejemplo, si no se cargaron los archivos de módulos o el de predicados de estado, se informará el problema por consola con los mensajes: `''No DynAlloy modules to translate!''` y `''No predicates modules to check!''`, respectivamente. En cambio, si los nombres de módulo o de la aserción a verificar son incorrectos, los mensajes respectivos son los siguientes:

```
*** Fatal error: check module identifier ***

*** Fatal error: check assertion identifier ***
```

Otra posible salida de error se produce cuando existe algún error sintáctico en el módulo. El mensaje correspondiente es: `''*** Syntax error found *** @Debug using debug file''`. El archivo de depuración, puede ser analizado con Alloy para verificar cuál es el error del módulo. Cuando ocurre un error en el chequeo de contraejemplos espúreos, se informa por consola con el mensaje `''***** Check counterexample spuriousness failed *****''`. En caso de que se introduzca un modelo en cual se incluyan acciones atómicas y se detecta un contraejemplo espúreo, el proceso de refinamiento no es soportado (como dijimos anteriormente, el cálculo de precondición más débil se puede aplicar sobre asignaciones y tests), lanzando el mensaje `"*** Predicate discover fail: wp not supported! ***''`. En caso de que la herramienta detecte que no se están introduciendo nuevos predicados, el mensaje el el siguiente:

```
*** Incomplete refinement : No more predicates could be discovered.
    Try to introduce new predicates and restart the excecution
```

Por último, en varias ocasiones, incluso cuando existe un error sintáctico, no es posible realizar la traducción de DynAlloy a Alloy, lo cual se informa con: `"Translation unsuccessful."`

5.5.4. Algunas restricciones a la hora de utilizar la herramienta

Como ya se mencionó anteriomente, la herramienta implementada toma como entrada, un módulo DynAlloy y un conjunto de predicados de abstracción, razón por la cual es inevitable la interacción con DynAlloy Translator. Esto trajo aparejado varias ventajas a la hora de reutilización de código y chequeo de modelos. Pero como era de esperar, el hecho de amoldarse a una implementación en particular, de alguna manera trae consigo la necesidad de introducir algunas restricciones a la hora de utilizar la herramienta, para solucionar algunos problemas que escapan al de la abstracción. Algunas de dichas restricciones son las palabras reservadas que contiene la herramienta, y que ya no son identificadores posibles en el modelo que introduce

el usuario. El primer caso nace desde la necesidad de contar con tipos booleanos (necesarios en la abstracción, por ejemplo para tener el elemento neutro de la conjunción), por lo cual se introdujeron los predicados `TruePred[]` y `FalsePred[]`.

En esta etapa surge otro problema: para implementar la función de abstracción, es necesario construir una aserción para invocar a Alloy Analyzer, a la cual debe asignarse un nombre. Como es de suponer, ese nombre no puede ser el nombre de una aserción en el modelo, con lo cual `AFCheck` (el nombre elegido para la misma), es otra palabra reservada. Lo mismo ocurre al momento de chequear si un contraejemplo es espúreo, caso para el cual el nombre de la aserción es `spuriousCounterExampleCheck`. En este último caso, además de dicha palabra reservada, se incluyen otras como `auxAbsState`+i y `absState`+i (donde i es un número entero) como nombre de predicados, ya que son utilizadas para construir un modelo DynAlloy con el contraejemplo encontrado.

Por último, en necesario que el modelo DynAlloy no incluya cláusulas `check` ni `run`, debido a que la herramienta invoca a DynAlloy Translator, quien intentará ejecutar dichos comandos.

5.5.5. Desarrollo y Avance de la Herramienta

En esta sección se describen algunos aspectos importantes durante el desarrollo y avance de la herramienta, como así también los inconvientes que fueron surgiendo y las soluciones de los mismos.

Comienzo

En el comienzo del proyecto, se tenía la necesidad de utilizar muchas de las construcciones de DynAlloy, por lo que la primera tarea de los desarrolladores fue interpretar el código fuente de la herramienta *DynAlloy Translator*, que de a poco fue influenciando nuestro diseño. En un principio, se desarrollaron los módulos principales: el módulo *GUI* (representa la interfaz del programa), *Abstractor* (módulo encargado de realizar la ejecución abstracta), *ActionManger* (almacena estados calculados previamente) y el módulo *abstractionsManager* (se comunica con AlloyAnalyzer y DynalloyTranslator), todos con sus respectivas interfaces, para poder ser adapatadas a nuevas implementaciones. Además se incluyó un conjunto de programas DynAlloy para poder realizar el testing de la herramienta. A medida que fue avanzando el desarrollo, fueron necesarias algunas clases auxiliares, que se ubicaron en el módulo *Util*. La mayoría de estas clases son implementaciones de recorridas sobre estructuras de datos mediante el patrón de diseño *visitor*, el cual es muy utilizado en la herramienta *DynAlloy Translator*, con el objetivo de separar el algoritmo de la estructura de un objeto [19]. Como ya se aclaró, se manipuló gran parte de las construcciones de DynAlloy Translator, de tal manera que se redefinieron algunas implementaciones de los visitors para poder recorrer sus estructuras acorde a lo que se necesitaba.

Uno de los inconvenientes que se tuvo que resolver a causa de la utilización de DynAlloy Translator, fue que dicha herramienta no cuenta con un parser para los predicados de un módulo Dynalloy, debido a que la traducción a Alloy es directa. Entonces se realizó dicho parser para los predicados, utilizando una gramática *ANTLR*, y de esta manera poder parsear los predicados de entrada de nuestra herramienta, y chequear posibles errores del usuario en la especificación de los mismos. Se agregó un nuevo módulo *parser*, que contiene las clases para solucionar el problema.

Mientras la investigación fue avanzando, se descubrieron nuevos métodos de abstracción más eficientes, que fueron fácil de adaptar y eliminar (en el caso de descubrir uno mejor) facilitado por el diseño de la herramienta. El último que se implemento fue el algoritmo *AbstractorWithGraph*, que utiliza una estructura de grafo para controlar la ejecución. Se utilizó la *notación DOT* para representar el grafo de ejecución en un archivo de salida, a modo de ayuda e información para el usuario. El módulo *lebeledProgram* es el encargado, de etiquetar un programa Dynalloy, tranformarlo a un grafo y luego traducirlo a un archivo de salida, para poder visualizarlo con alguna herramienta visor de imagen.

Lo último que se implementó de la herramienta, que ya estaba previsto por el diseño, fue el módulo *abstractionRefinement*, encargado de realizar la técnica de refinamiento de abstracciones ya explicada.

Herramientas utilizadas

Eclipse

El desarrollo de la herramienta se realizó completamente en Eclipse, el cual nos resulto fácil usarlo y nos brindó muchas ventajas sin usar todas las posibilidades que ofrece.
Eclipse es un entorno de desarrollo integrado de código abierto multiplataforma para desarrollar [21]. Esta plataforma, típicamente ha sido usada para desarrollar entornos de desarrollo integrados (IDE), como el IDE de Java llamado Java Development Toolkit (JDT) y el compilador (ECJ) que se entrega como parte de Eclipse (y que son usados también para desarrollar el mismo Eclipse). Sin embargo, también se puede usar para otros tipos de aplicaciones cliente.
Eclipse dispone de un Editor de texto con resaltado de sintaxis. La compilación es en tiempo real. Tiene pruebas unitarias con JUnit, control de versiones con CVS, integración con Ant, asistentes (wizards) para creación de proyectos, clases, tests, etc., y refactorización. Asimismo, a través de plugins libremente disponibles es posible añadir control de versiones con *Subversion* e integración con *Hibernate*.

ANTLR

ANother Tool for Language Recognition (otra herramienta para reconocimiento de lenguajes) es una herramienta creada principalmente por Terence Parr [20], que opera sobre lenguajes, proporcionando un marco para construir reconocedores (parsers), intérpretes, compiladores y traductores de lenguajes a partir de las descripciones gramaticales de los mismos (conteniendo acciones semánticas a realizarse en varios lenguajes de programación).

El lenguaje DOT

Es un lenguaje estándar de descripción de grafos que es usado dentro de algunas aplicaciones. Este lenguaje describe grafos como texto, listo para ser leído e interpretado por alguna otra herramienta.
Dot reconoce tres tipos de objetos: nodos, aristas y subgrafos, y puede incluir un grupo de atributos para cada uno de esos elementos. Son estos atributos los que le confieren a DOT una gran flexibilidad al momento de crear el grafo, ya que los atributos pueden ser cosas tan simple como el tamaño de la fuente de un nodo o cosas algo más complejas, como los distintos tipos de formas que puede tomar un nodo. Puede verse la sintaxis del lenguaje en [22].

Capítulo 6

Casos de estudio

En este capítulo se presentan algunos resultados experimentales obtenidos con la herramienta implementada. Los experimentos se llevaron a cabo sobre un procesador Intel Core 2 Duo de 2Ghz, con 2Gb de memoria RAM y sobre un Sistema Operativo GNU/Linux 2.6. La versión de Alloy Analyzer utilizada fue la 4.1.8. El timeout considerado para los ejemplos corridos con nuestra herramienta fue de 2 horas.

Para analizar el comportamiento de la herramienta se debieron considerar dos posibles situaciones:

- La eficiencia en el procedimiento de abstraccción de la herramienta.
- La respuesta del módulo de refinamiento, en caso de existir contraejemplos espúreos.

La primera situación se trata de ejecutar el algoritmo de abstracción con todos los predicados necesarios para que la propiedad sea válida (dentro de las cotas provistas).

En la segunda situación los predicados introducidos son insuficientes para verificar la propiedad por lo que se encuentra contraejemplo espúreo. En este caso es el módulo de refinamiento el que se encarga de descubrir nuevos predicados, para que la abstracción verifique la propiedad.

Los modelos concretos (DynAlloy) completos pueden visualizarse en el apéndice A.

6.1. Abstracción sin necesidad de refinamiento

En esta sección se consideran modelos cuyos predicados de abstracción son suficientes para verificar las propiedades dentro de las cotas provistas. Los algoritmos para realizar la ejecución abstracta son los mencionados en la sección 5.1.

Para poder observar y analizar los resultados de la técnica de abstracción es necesario tener un parámetro que nos permita inferir si la técnica es eficiente o no. En nuestro caso, ese parámetro lo brinda la herramienta Alloy Analyzer, la cual fue utilizada para verificar las propiedades sobre los modelos concretos. Los tiempos obtenidos de estas ejecuciones son el objetivo a mejorar.

Todos los modelos que aparecen en esta sección están basados en la siguiente especificación de listas: cuadro 6.1.

```
sig List {}
sig Node {}
one sig NullValue { }
-- auxiliary preds to represent true and false
pred TruePred[] { }

pred FalsePred[] {
  not TruePred[]
}
```

Cuadro 6.1: Representación de listas en modelos DynAlloy.

6.1.1. La eliminación preserva aciclicidad de listas

En nuestro primer caso de estudio se trabajó con un programa que, dada una lista acíclica, elimina los nodos que contienen ciertos valores (en dicho modelo se identifican tres valores a eliminar: v1, v2 y v3). La propiedad que se desea verificar sobre este programa es que la lista resultante preserva aciclicidad. El programa especificado en el lenguaje DynAlloy es presentado en el cuadro 6.1.1.

```
1  assertCorrectness RemAllCorrect [list: List, current: Node+NullValue, prev: Node+NullValue, v1: ↩
       Char, v2: Char, v3: Char, getValue: Node -> one Char, next: Node -> one (Node+NullValue), ↩
     head: List -> one (Node+NullValue)] {
2    pre = { removeAllPre[list, getValue, next, head] }
3    program = {
4      assume notEmpty[list, head];
5      assume PpreInit[list, current, prev, v1, v2, v3, getValue, next, head];
6      prev:=NullValue;
7      current:=list.(head);
8      repeat{
9        assume notNullNode[current];
10         (
11           (
12             assume valueOk[current, v1, v2, v3, getValue];
13             (
14               (
15                 assume notNullNode[prev];
16                 assume PpreSetNext[list, current, prev, v1, v2, v3, getValue, next, head];
17                 next:= next++(prev -> current.next);
18                 current:=current.next
19               )
20               +
21               (
22                 [nullPrev[prev]]?;
23                 assume PpreSetHead[list, current, prev, v1, v2, v3, getValue, next, head];
24                 head:= head++(list -> current.next);
25                 current:= current.(next)
26               )
27             )
28           )
29           +
30           (
31             assume notValueOk[current, v1, v2, v3, getValue];
32             assume PpreIncPrev[list, current, prev, v1, v2, v3, getValue, next, head];
33             prev:= current;
34             current:= current.(next)
35           )
36         )
37   };
38   assume nullCurrent[current]
39   }
40  post = { noLoops[list', head', next', getValue']  }
41 }
```

Cuadro 6.2: La eliminación preserva aciclicidad de listas.

Para que la herramienta verifique la propiedad fueron necesarios los siguientes predicados de abstracción:

```
noLoops[list, head, next, getValue] --post
removeAllPre[list, getValue, next, head] --pre
notEmpty[list, head]
notNullNode[current]
valueOk[current, v1, v2, v3, getValue]
notNullNode[prev]
noLoops[list,head,next++(prev->current.next),getValue]
noLoops[list,head++(list->current.next),next,getValue]
```

Luego de realizar las respectivas ejecuciones mencionadas anteriormente, tanto ejecuciones abstractas como ejecuciones concretas con el Alloy Analyzer, los tiempos obtenidos fueron los presentados en la tabla 6.1.

Loop unrolls	Alloy	OnDemand	WithGraph
15	17m42s	1h 46m 42s	15s
16	28m 31s	TIMEOUT	15s
17	MEM error		15s
...			

Figura 6.1: Tiempos de test de la eliminación preserva aciclicidad.

6.1.2. Asignación de valores a una lista según un flag

El segundo caso de estudio que presentamos corresponde a la especificación de un programa DynAlloy que partiendo de una lista vacía y de un flag arbitrario, crea una lista incrementalmente asignando valores a la misma, dependiendo de los cambios que se produzcan en el flag. En el caso de que el flag contenga el valor 1, se asigna el valor 1 a la posición corriente, mientras que si el valor de flag es 0, se asigna el valor 2. Finalmente, cuando se completa la creación de la lista, se marca el final de la misma con el valor 3. La propiedad a verificar indica que si el nodo que le sigue al corriente es igual a null (o sea que el nodo corriente corresponde al final de la lista), entonces el valor del corriente es igual a 3, y en caso contrario el valor es igual a 1 o 2, según el valor del flag descripto anteriormente.

La especificación del programa escrita en el lenguaje Dynalloy se presenta en el cuadro 6.1.2.

```
1  assertCorrectness prop1 [l: List, head: List -> one (Node+NullValue), next: Node -> one (Node+↩
       NullValue), val: Node -> one Int, curr: Node+NullValue, n: Int, i: Int, flag: Int] {
2    pre = { prop1Pre[l, head, flag, i, n] }
3    program = {
4      init[l, head, next, curr];
5      repeat{
6        assume NegatetermCond[i,n];
7        (
8          assume flagSet[flag];
9          setValue1[val,curr]
10       )
11       +
12       (
13         assume flagNotSet[flag];
14         setValue2[val,curr]
15       );
16       insertl[l,head,next,curr] -- t = new; t->n = null; curr->n = t; curr = curr->n
17     };
18     assume termCond[i,n];
19     setValue3[val,curr]
20     }
21   post = { prop1Post[l', head', next', val', flag'] }
22 }
```

Cuadro 6.3: Asignación de valores según un flag.

Por otro lado, los predicados de abstracción necesarios son los siguientes:

```
prop1Post[l, head, next, val, flag]
flagNotSet[flag]
flagSet[flag]
nullNode[curr,next]
```

```
invariant[l, head, next, val, flag, curr]
valueCurr[curr, val, 1]
valueCurr[curr, val, 2]
valueCurr[curr, val, 3]
currReach[l, head, next, curr]
```

Cuadro 6.4: Predicados de abstracción.

Luego de realizar las respectivas ejecuciones los tiempos obtenidos son los presentados en la tabla 6.2.

Loop unrolls	Alloy	OnDemand	WithGraph
4	2s	10s	8s
10	1m 20s	1m 8s	10s
15	15m 30s	1m 8s	10s
25	77m 32s	2m 59s	10s
35	MEM error	6m 5s	10s
80		41m 17s	10s
100		70m	10s
120		100m 30s	10s
121		TIMEOUT	10s
...			

Figura 6.2: Tiempos de test de listas con un flag.

6.1.3. División de una lista en dos

El último caso que utilizamos para verificar la eficiencia de los algoritmos de abstracción y compararlos con la verificación de propiedades sobre modelos concretos, también tiene que ver con un problemas bastante intuitivo sobre listas. Se divide la lista original en dos nuevas listas, según el valor de sus elementos: si el elemento es un 1 se lo asigna a una lista, y si es un 2 a la otra. La propiedad a verificar indica que una lista contienen todos 1 y la otra todos 2.

La especificación del programa escrita en el lenguaje Dynalloy se presenta en el cuadro 6.1.3.

```
1  assertCorrectness prop1 [l: List, head: List -> one (Node+NullValue), next: Node -> one (Node+↩
       NullValue), val: Node -> one Int, curr: Node+NullValue, temp: Node+NullValue, l1: List, l2: ↩
       List] {
2    pre = { prop1Pre[l,l1,l2, head, next, val] }
3    program = {
4      assume notEmpty[l, head];
5      assume preInit[l, head, curr];
6      curr := l.head;
7      repeat{
8        assume NegatevalThree[val, curr];
9        assume preAstempcurr[next, curr, temp];
10       temp:= curr;
11       curr:= curr.next;
12       (
13         ( assume valOne[val, temp];
14           assume preInsertl[l1, head, next, temp];
15           next := next ++ (temp->l1.head);
16           head := head ++ (l1->temp)
17         )
18        +
19         ( assume NegatevalOne[val, temp]
20         )
21       );
22       (
23         ( assume valTwo[val, temp];
24           assume preInsertl[l2, head, next, temp];
25           next := next ++ (temp->l2.head);
```

```
26            head := head ++ (l2->temp)
27          )
28          +
29          ( assume NegatevalTwo[val, temp]
30          )
31        )
32      };
33      assume valThree[val, curr]
34    }
35    post = { prop1Post[l1', l2', head', next', val'] }
36  }
```

Cuadro 6.5: División de una lista en dos.

Por otro lado, los predicados de abstracción necesarios para verificar la propiedad son los que figuran en el cuadro 6.1.3.

```
prop1Post[l1, l2, head, next, val] --post
prop1Pre[l,l1,l2, head, next, val] --pre
notEmpty[l, head]
valThree[val, curr]
valOne[val, temp]
valTwo[val, temp]
diff[l1,l2]
-- auxpreds l1
prop1Post[l1, l2, (head)++((l1)->(temp)), next, val]
prop1Pre[l,l1,l2, (head)++((l1)->(temp)), next, val]
notEmpty[l, (head)++((l1)->(temp))]
prop1Post[l1, l2, head, (next)++(((temp)->(l1)).head), val]
prop1Pre[l,l1,l2, head, (next)++(((temp)->(l1)).head), val]
-- auxpreds l2
prop1Post[l1, l2, (head)++((l2)->(temp)), next, val]
prop1Pre[l,l1,l2, (head)++((l2)->(temp)), next, val]
notEmpty[l, (head)++((l2)->(temp))]
prop1Post[l1, l2, head, (next)++(((temp)->(l2)).head), val]
prop1Pre[l,l1,l2, head, (next)++(((temp)->(l2)).head), val]
```

Cuadro 6.6: Predicados de abstracción.

Los tiempos obtenidos luego de realizar las ejecuciones abstractas y las ejecuciones concretas fueron los presentados en la tabla 6.3.

Loop unrolls	Alloy	OnDemand	WithGraph
4	1m	1m 46s	1m 5s
10	2m 10s	2m 44s	1m 9s
12	45m 20s	21m 45s	1m 9s
13	TIME OUT	70m 30s	1m 12s
14		TIMEOUT	1m 12s
...			
100			1m 12s
...			

Figura 6.3: Resultados de test de división de lista en dos.

6.2. Abstracción utilizando refinamiento

En esta sección se analizará el desempeño del módulo de refinamiento propuesto en este trabajo final en las secciones 4.4 y 5.3. Cabe destacar que la implementación desarrollada es una versión experimental. El

campo de aplicación de esta técnica amerita una investigación a fondo lo cuál escapa a los objetivos de este trabajo final.

A medida que la implementación del módulo de refinamiento avanzaba, nos encontramos con ciertos resultados que nos obligaron a investigar sobre otras teorías que trataran el tema. Los principales cambios en la implementación se debieron a que los resultados no eran los esperados cuando se estudió la teoría, y en nuestro caso la técnica de refinamiento resultaba inútil.

Veamos un breve ejemplo sobre lo mencionado anteriormente. Considere el modelo DynAlloy y los predicados de abstracción que figuran en el cuadro 6.7.

```
-- Modelo DynAlloy
sig Elem {}
pred equal[x,y:Elem]{
 x = y
}
assertCorrectness swap [a,b,c,d : Elem]{
  pre = { equal[a,b]}
  program = {
    c:=a;
    d:=b
  }
  post = { equal[c',d'] }
}

-- Predicados de Abstraccion
equal[c,d]
equal[a,b]
```

Cuadro 6.7: Modelo DynAlloy y Predicados de Abstracción.

Al ejecutar dicho modelo con algunas de las técnicas mostradas en la sección 5.1, la ejecución encuentra un contraejemplo espúreo. En este modelo existe una única traza (cuadro 6.8) y además se da el caso de que toda la traza es espúrea.

```
[X, T]
c:=a
[X, T]
d:=b
[X, T]
```

Cuadro 6.8: Traza espúrea del programa swap.

Siguiendo la técnica mencionada en la sección 4.4 el predicado que se descubre expresa lo siguiente : `not equal[a,b] and equal[a,b]`. Al principio nos resulto cómico que la técnica descubra una contradicción, pero al analizar en detalle el nuevo predicado observamos que la implementación era acorde a la teoría, por lo tanto nos encontramos en un serio problema. Para resolverlo tuvimos que pensar en alguna heurística, la cual mencionamos en la sección 5.3.

Luego de lograr el efecto deseado en el algoritmo de refinamiento y obtener un módulo acorde a la sección 5.3, se procedió a realizar experimentos con los modelos ya mencionados en la sección de casos de estudio sin utilizar el refinador.

6.2.1. La eliminación preserva aciclicidad de listas + Refinamiento

Esta sección está basada sobre el modelo DynAlloy presentado en la sección 6.1.1. La principal diferencia con esa sección, son los predicados de abstracción que sirven de entrada para la ejecución abstracta. En este caso, queremos lograr que los predicados sean insuficientes para verificar la propiedad y así la herramienta encuentre contraejemplo espúreo, para luego utilizar el refinador y analizar la eficiencia de la técnica.

En los experimentos se utilizó el algoritmo de abstracción *AbstractorWithGraph*. En la figura 6.4 se presenta una comparación ente los tiempos obtenidos utilizando el Alloy Analyzer y la ejecución abstracta con el refinador.

Loop unrolls	Alloy	WithGraph+Refinamiento
15	17m 42s	31s
16	28m 31s	31s
17	MEM error	31s
18	MEM error	31s
...		
100	MEM error	31s
...		

Figura 6.4: Alloy Analyzer vs. WithGraph+Refinamiento.

Capítulo 7

Conclusiones y trabajo futuro

7.1. Conclusiones

En el presente trabajo se presentó una técnica de abstracción por predicados según [14] aplicada sobre especificaciones escritas en DynAlloy. La idea es a partir de un modelo de un sistema escrito en dicho lenguaje y un conjunto de predicados de entrada (los cuales brindan información acerca de los estados del programa) construir un modelo aproximado del sistema (es decir con menos información) y verificar propiedades sobre este modelo abstracto. Dicha técnica se implementó mediante dos algoritmos: en el primero la construcción de las trazas abstractas se va realizando *bajo demanda*, es decir, que cuando se construye una traza, primero se verifica la propiedad sobre la misma antes de construir una nueva, para no generar trazas innecesariamente. Además, las trazas anteriores no son almacenadas, para evitar el uso excesivo de memoria. El segundo algoritmo, utiliza un grafo como estructura de control de la ejecución. El mismo sirve para optimizar el proceso de abstracción mediante la detección de ciclos. Esto quiere decir que, cuando en el programa tenemos un ciclo y a partir de cierta iteración partimos de un estado inicial del ciclo llegando siempre al mismo estado final, entonces la ejecución del resto de las iteraciones es innecesaria, mejorando así el tiempo de la ejecución abstracta. También para ambos algoritmos se implementó una estructura de datos (basada en tablas de hash) para el almacenamiento de cálculos previos, lo cual permite una búsqueda más eficiente.

Los tiempos obtenidos para la verificación de propiedades con el primer algoritmo reflejan que no se obtuvo una mejora significativa con respecto a la verificación de propiedades en concreto (con Alloy Analyzer). Pero por otro lado, los tiempos obtenidos en el algoritmo con grafo superaron nuestras expectativas: se lograron incrementar las cotas de iteración y de dominios con respecto a la verificación en el modelo concreto. Más aún, si se detecta que la ejecución de las iteraciones converge a partir de cierta cantidad de ciclos, desde ese punto es posible incrementar la cota de iteración tanto como se lo desee sin incrementar el tiempo de ejecución. Gracias a esto se logran mejoras en los tiempos con respecto a la verificación en concreto.

Es necesario mencionar que la técnica de abstracción mencionada anteriormente posee dos características importantes: por un lado si una propiedad es válida en el modelo abstracto, también podemos afirmar que vale en el modelo concreto (la abstracción es "conservativa"). Pero por otro lado, si se encuentra una violación de la propiedad en el modelo abstracto (contraejemplo) no podemos afirmar que existe una violación en el modelo concreto (la abstracción es "débil"). En caso de encontrar un contraejemplo abstracto, es necesario depurarlo para verificar si se corresponde con alguna ejecución concreta. Si esto ocurre decimos que el contraejemplo es *real*, en caso contrario decimos que es *espúreo*. Cuando nos encontramos en esta última situación, es necesario refinar el modelo abstracto mediante alguna técnica.

En este trabajo se implementó una técnica de refinamiento completamente experimental basada en [14]. La idea de la misma es descubrir nuevos predicados de abstracción que agreguen la información necesaria para eliminar los contraejemplos espúreos, es decir, obtener un modelo abstracto más detallado. Una vez que se ha obtenido un contraejemplo espúreo, el primer paso es encontrar que parte de la traza es la mínima traza espúrea (siempre comenzando desde el final de la traza, y agregando acciones anteriores mientras la traza no sea espúrea). Luego se realiza un cálculo de wp (weakest precondition o pre-condición más débil)

de la primera acción considerada, con el estado final. Por último se calcula el wp de la acción anterior a la primera acción considerada en la traza minimal con el wp calculado anteriormente, y se procede a reiniciar la ejecución abstracta, considerando los wp mencionados anteriormente como predicados de abstracción. En caso de que estos predicados no agreguen nueva información, se utiliza una heurística basada en considerar la acción anterior a la primera acción de la traza minimal como dentro de esta, y repetir el proceso anterior. Una restricción que presenta la técnica es que las acciones que componen los programas deben se *inversibles*, es decir que puedan ser escritas mediante asignaciones y tests. En nuestro caso esto no es un inconveniente grave, debido a que asumimos que las especificaciones con las que se utiliza la herramienta provienen de programas Java. Utilizando tales especificaciones observamos que los resultados obtenidos indican que encontramos una variedad de casos en los cuales la técnica se puede utilizar y obtener tiempos razonables de ejecución, pero también casos en los cuales la herramienta itera infinitamente (o hasta que expire el timer) descubriendo nuevos predicados que no agregan la información deseada. Este es el motivo por el cual es necesario seguir investigando en este campo.

7.2. Trabajo futuro

Aunque como mencionamos en la sección anterior los resultados obtenidos son bastante prometedores, es posible mejorar algunos aspectos de la herramienta. En primer lugar, la principal debilidad de este trabajo final reside en el módulo de refinamiento. Consideramos que es muy importante realizar un estudio más a fondo acerca de esta teoría, debido a que no existe demasiada información al respecto. Lo ideal sería dejar de utilizar la heurística mencionada en capítulos anteriores y lograr la implementación de una técnica de refinamiento de abstracciones, con una fuerte base teórica, que siempre permita eliminar contraejemplos espúreos.

Por otra parte, la idea de la herramienta desde un principio (pero aún pendiente) fue integrar la presente implementación con una herramienta que traduce código Java anotado a una especificación DynAlloy [15]. De esta manera es posible verificar eficientemente propiedades sobre programas escritos en Java.

Apéndice A

Especificaciones completas de casos de estudio

A.1. La eliminación preserva aciclicidad de listas

```
-- list.dals

-- list : List
-- current : Current
-- prev : Prev
-- v1 : x
-- v2 : y
-- v3 : z
-- getValue : value
-- next : next
-- head : head

module list

abstract sig BooleanValue {}
one sig FalseValue extends BooleanValue {}
one sig TrueValue extends BooleanValue {}

one sig List {}
sig Node {}
sig Char {}
one sig NullValue { }

-- auxiliary preds to represent true and false
pred TruePred[] {
}

pred FalsePred[] {
  not TruePred[]
}

-- auxiliary pred. for modelling rel. overriding
pred overrideRelPost[r0,r1:univ->univ,l,r: univ] {
    r1=r0++(l->r)
}

-- invariant for lists: no cycles and no repeated elems
pred noCyclesNoRepeated[thisV:univ, headV:univ -> one univ, nextV : univ -> one univ, valueV : univ -> one univ] {
   ( all n: Node |
        n in thisV.headV.(*nextV) implies n !in n.nextV.(*nextV) &&
        all n1, n2: Node |
            (n1+n2 in thisV.headV.(*nextV) and n1.valueV=n2.valueV)
                implies n1=n2 )
}

```

```
46 -- no loops
47 pred noLoops[thisV:univ, headV:univ -> one univ, nextV : univ -> one univ, valueV : univ -> one univ]{
48       (all n: Node |
49             n in thisV.headV.(*nextV) implies n !in n.nextV.(*nextV))
50
51 }
52
53
54 -- no repeated elements
55 pred noRepeated[thisV:univ, headV:univ -> one univ, nextV : univ -> one univ, valueV : univ -> one univ] {
56  ( all n1, n2: Node |
57              (n1+n2 in thisV.headV.(*nextV) and n1.valueV=n2.valueV)
58                           implies n1=n2 )
59 }
60
61 -- v1, v2 and v3  do not appear on the list until 'prev' element including
62 pred notAppearsBeforeCurrent[thisV:List, headV:List -> one (Node + NullValue), currV: Node+NullValue, nextV : Node -> one (Node + NullValue), valueV : Node -> one Char, current, prev, v1: Char ] {
63        ( all n: Node | n in (thisV.headV.(*nextV) - (currV.(*nextV))) implies n.valueV !in (current+prev+v1) )
64 }
65
66 -- 'prev' element precedes to current element
67 pred prevPrecedesCurrent[thisV:List, headV:List -> one (Node + NullValue), prevV: Node+NullValue, currV: Node+NullValue, nextV : Node -> one (Node + NullValue)] {
68        ((prevV = NullValue and currV = thisV.headV) or (prevV != NullValue and prevV.nextV = currV))
69 }
70
71 -- no loops and current always is reachable from head
72 pred currentInList[list: List, current: Node+NullValue, prev: Node+NullValue, v1:  Char, v2: Char, v3: Char, getValue: Node -> one Char, next: Node -> one (Node+NullValue), head: List -> one (Node+NullValue)] {
73   noLoops[list, head, next, getValue] && current in list.head.(*next) && prev in list.head.(*next) && ((prev = NullValue and current = list.head) or (prev != NullValue and prev.next = current))
74 }
75
76 -- not empty list
77 pred notEmpty[list: List, head: List -> one (Node+NullValue)] {
78   (list.(head) != NullValue)
79 }
80
81
82 -- item is not null
83 pred notNullNode[item: Node+NullValue] {
84    ( item != NullValue )
85 }
86
87 --- current value = v1 or current value = v2 or current value = v3
88 pred valueOk[current: Node+NullValue, v1:  Char, v2: Char, v3: Char, getValue: Node -> one Char]{
89  (current.(getValue) in (v1+v2+v3))
90 }
91
92
93 -- prev is null
94 pred nullPrev[prev: Node+NullValue] {
95   !notNullNode[prev]
96 }
97
98
99 -- not (current value = v1 or current value = v2 or current value = v3)
100 pred notValueOk[current: Node+NullValue, v1:  Char, v2: Char, v3: Char, getValue: Node -> one Char] {
101   !valueOk[current, v1, v2, v3, getValue]
102 }
103
104 -- current is null
105 pred nullCurrent[current: Node+NullValue] {
106     !notNullNode[current]
107 }
108
109
```

```
pred PpreInit[list: List, current: Node+NullValue, prev: Node+NullValue, v1:  Char, v2: Char, v3: Char, getValue: Node -> one Char, next: Node -> one (Node+NullValue), head: List -> one (Node+NullValue)] {
    notEmpty[list, head]
}

pred PpreSetNext[list: List, current: Node+NullValue, prev: Node+NullValue, v1:  Char, v2: Char, v3: Char, getValue: Node -> one Char, next: Node -> one (Node+NullValue), head: List -> one (Node+NullValue)] {
    notNullNode[current] &&  valueOk[current, v1, v2, v3, getValue] && (prev != NullValue)
}

pred PpreSetHead[list: List, current: Node+NullValue, prev: Node+NullValue, v1:  Char, v2: Char, v3: Char, getValue: Node -> one Char, next: Node -> one (Node+NullValue), head: List -> one (Node+NullValue)] {
    notNullNode[current] && valueOk[current, v1, v2, v3, getValue] && nullPrev[prev]
}


pred PpreIncPrev[list: List, current: Node+NullValue, prev: Node+NullValue, v1:  Char, v2: Char, v3: Char, getValue: Node -> one Char, next: Node -> one (Node+NullValue), head: List -> one (Node+NullValue)] {
  notNullNode[current] && notValueOk[current, v1, v2, v3, getValue]
}


-- RemAllCorrect Assertion
pred removeAllPre[thisV: univ, valueV: univ -> one univ, nextV: univ -> one univ, headV: univ -> one univ] {
    noCyclesNoRepeated[thisV,headV,nextV,valueV]

}
-- assert to check
-- initially not exists loops in list, then after delete v1,v2 and v3 elements from list, should not exists loops in list
assertCorrectness RemAllCorrect [list: List, current: Node+NullValue, prev: Node+NullValue, v1: Char, v2: Char, v3: Char, getValue: Node -> one Char, next: Node -> one (Node+NullValue), head: List -> one (Node+NullValue)] {
  pre = { removeAllPre[list, getValue, next, head] }
  program = {
    assume notEmpty[list, head];

    assume PpreInit[list, current, prev, v1, v2, v3, getValue, next, head];
    prev:=NullValue;
    current:=list.(head);
    repeat{
      assume notNullNode[current];
        (
          (
            assume valueOk[current, v1, v2, v3, getValue];
            (
              (
                assume notNullNode[prev];
                  assume PpreSetNext[list, current, prev, v1, v2, v3, getValue, next, head];
                  next:= next++(prev -> current.next);
                  current:=current.next
              )
             +
              (
                [nullPrev[prev]]?;
                  assume PpreSetHead[list, current, prev, v1, v2, v3, getValue, next, head];
                  head:= head++(list -> current.next);
                  current:= current.(next)
              )
            )
          )
         +
          (
            assume notValueOk[current, v1, v2, v3, getValue];
             assume PpreIncPrev[list, current, prev, v1, v2, v3, getValue, next, head];
              prev:= current;
              current:= current.(next)

          )
        )
  };
  assume nullCurrent[current]
```

```
175   }
176  post = { noLoops[list', head', next', getValue']  }
177 }
```

A.2. Asignación de valores a una lista según un flag

```
1  -- list_flag.dals
2
3  -- list : List
4  -- current : Current
5  -- prev : Prev
6  -- v1 : x
7  -- v2 : y
8  -- v3 : z
9  -- getValue : value
10 -- next : next
11 -- head : head
12
13 module list_flag
14
15 sig List {}
16 sig Node {}
17 one sig NullValue { }
18
19 -- auxiliary preds to represent true and false
20 pred TruePred[] {
21 }
22
23 pred FalsePred[] {
24   not TruePred[]
25 }
26
27 -- Action add1
28 pred postAdd1[i, i': Int] {
29   i' = i + 1
30 }
31
32 pred preAdd1[i: Int] {
33
34 }
35
36 act add1[i: Int] {
37    pre  { preAdd1[i] }
38    post  { postAdd1[i,i'] }
39 }
40 -- End Action add1
41
42
43 -- Action init
44 pred postInit[l,l': List, head,head': List -> one (Node+NullValue), next, next': Node -> one (↩
       Node+NullValue), curr,curr': Node+NullValue] {
45     curr' != NullValue and next' = next ++ (curr'->NullValue) and l = l' and head' = head ++ (l↩
           ->curr')
46 }
47
48 pred preInit[l: List, head: List -> one (Node+NullValue), next: Node -> one (Node+NullValue), ↩
       curr: Node+NullValue] {
49
50 }
51
52 act init[l: List, head: List -> one (Node+NullValue), next: Node -> one (Node+NullValue), curr: ↩
       Node+NullValue] {
53    pre  { preInit[l, head, next, curr]  }
54    post  { postInit[l,l', head,head', next,next', curr,curr'] }
55 }
56 -- End Action init
57
58 -- Action setValue1
59 pred postsetValue1[val,val': Node -> one Int, curr,curr': Node+NullValue] {
60     curr = curr' and val' = val ++ (curr->1)
61 }
62
```

```
63  pred presetValue1[val: Node -> one Int, curr: Node+NullValue] {
64
65  }
66
67  act setValue1[val: Node -> one Int, curr: Node+NullValue] {
68     pre  { presetValue1[val, curr]  }
69     post  { postsetValue1[val,val',curr,curr'] }
70  }
71  -- End Action setValue1
72
73
74  -- Action setValue2
75  pred postsetValue2[val,val': Node -> one Int, curr,curr': Node+NullValue] {
76      curr = curr' and val' = val ++ (curr->2)
77  }
78
79  pred presetValue2[val: Node -> one Int, curr: Node+NullValue] {
80
81  }
82
83  act setValue2[val: Node -> one Int, curr: Node+NullValue] {
84     pre  { presetValue2[val, curr]  }
85     post  { postsetValue2[val,val',curr,curr'] }
86  }
87  -- End Action setValue2
88
89
90  -- Action setValue3
91  pred postsetValue3[val,val': Node -> one Int, curr,curr': Node+NullValue] {
92      curr = curr' and val' = val ++ (curr->3)
93  }
94
95  pred presetValue3[val: Node -> one Int, curr: Node+NullValue] {
96
97  }
98
99  act setValue3[val: Node -> one Int, curr: Node+NullValue] {
100    pre  { presetValue3[val, curr]  }
101    post  { postsetValue3[val,val',curr,curr'] }
102 }
103 -- End Action setValue3
104
105
106 -- Action inccurr
107 pred postInccurr[next: Node -> one (Node+NullValue), curr: Node+NullValue] {
108     curr = curr.next
109 }
110
111 pred preInccurr[next: Node -> one (Node+NullValue), curr: Node+NullValue] {
112
113 }
114
115 act inccurr[next: Node -> one (Node+NullValue), curr: Node+NullValue] {
116    pre  { preInccurr[next, curr]  }
117    post  { postInccurr[next, curr] }
118 }
119 -- End Action inccurr
120
121 -- Action insertl
122 pred postInsertl[l,l': List, head,head': List -> one (Node+NullValue), next,next': Node -> one (↩
        Node+NullValue), curr,curr': Node+NullValue] {
123   -- t = new; t->n = null; curr->n = t; curr = curr->n
124   (some n: Node | n not in l.head.*next and next' = next ++ (curr->n) ++ (n->NullValue) and l = ↩
        l' and head = head' and curr' = n)
125 }
126
127 pred preInsertl[l: List, head: List -> one (Node+NullValue), next: Node -> one (Node+NullValue),↩
        curr: Node+NullValue] {
128
129 }
130
131 act insertl[l: List, head: List -> one (Node+NullValue), next: Node -> one (Node+NullValue), ↩
        curr: Node+NullValue] {
132    pre  { preInsertl[l, head, next, curr]  }
133    post  { postInsertl[l,l', head, head', next,next', curr,curr'] }
134 }
135 -- End Action insertl
```

```
136
137
138 pred termCond[i: Int, n: Int] {
139   i = n
140 }
141
142 pred NegatetermCond[i: Int, n: Int] {
143   !termCond[i,n]
144 }
145
146 pred flagSet[flag: Int] {
147   flag = 1
148 }
149
150 pred NegateflagSet[flag: Int] {
151   !flagSet[flag]
152 }
153
154 pred flagNotSet[flag: Int] {
155   flag = 0
156 }
157
158 pred NegateflagNotSet[flag: Int] {
159   !flagNotSet[flag]
160 }
161
162 pred notEmpty[list: List, head: List -> one (Node+NullValue)] {
163   (list.(head) != NullValue)
164 }
165
166 pred NegatenotEmpty[list: List, head: List -> one (Node+NullValue)] {
167         not notEmpty[list, head]
168 }
169
170 -- shape analysis preds
171 pred currReach[list: List, head: List -> one (Node+NullValue), next: Node -> one (Node+NullValue↩
      ), curr: Node+NullValue] {
172   curr in list.head.*next
173 }
174
175 pred NegatecurrReach[list: List, head: List -> one (Node+NullValue), next: Node -> one (Node+↩
      NullValue), curr: Node+NullValue] {
176   !currReach[list,head,next,curr]
177 }
178
179 -- end of shape analysis preds
180
181
182 pred prop1Pre [l: List ,head: List -> one (Node+NullValue), flag: Int, i: Int, n: Int] {
183   l.head = NullValue and (flag = 1 or flag = 0) and i = 1 and n > 0
184 }
185
186 pred prop1Post [l: List, head: List -> one (Node+NullValue), next: Node -> one (Node+NullValue),↩
      val: Node -> one Int, flag: Int] {
187    (all n: Node | (n in l.head.*next =>
188                      ((n.next = NullValue => n.val = 3)
189                      and
190                      (n.next != NullValue => (flag = 1 => n.val = 1) and (flag = 0 => n.val = 2))↩
                           )
191                   )
192    )
193 --(all n: Node | (n in l.head.*next => n.val = 3))
194 -- l.head = NullValue
195 }
196
197 pred invariant[l: List, head: List -> one (Node+NullValue), next: Node -> one (Node+NullValue), ↩
      val: Node -> one Int, flag: Int, curr: Node+NullValue]{
198  (all n: Node | (n in (l.head.*next - curr) =>  (flag = 1 and n.val = 1) or (flag = 0 and n.val ↩
       = 2 )))
199 }
200
201 pred Negateinvariant[l: List, head: List -> one (Node+NullValue), next: Node -> one (Node+↩
      NullValue), val: Node -> one Int, flag: Int, curr: Node+NullValue]{
202         not invariant[l, head, next, val, flag, curr]
203 }
204
205 pred valueCurr[curr:Node+NullValue, val: Node -> one Int, v: Int]{
```

```
206         curr.val = v
207 }
208
209 pred NegatevalueCurr[curr:Node+NullValue, val: Node -> one Int, v:Int]{
210         not valueCurr[curr, val, v]
211 }
212
213 pred nullNode[curr: Node+NullValue, next: Node -> one (Node+NullValue)]{
214         curr.next=NullValue
215 }
216
217 pred Negateprop1Post [l: List, head: List -> one (Node+NullValue), next: Node -> one (Node+↩
      NullValue), val: Node -> one Int, flag: Int] {
218    !prop1Post[l,head,next,val, flag]
219 }
220
221 pred NegatenullNode[curr: Node+NullValue,next: Node -> one (Node+NullValue)]{
222         not nullNode[curr,next]
223 }
224
225
226
227
228 --Property to check
229 assertCorrectness prop1 [l: List, head: List -> one (Node+NullValue), next: Node -> one (Node+↩
      NullValue), val: Node -> one Int, curr: Node+NullValue, n: Int, i: Int, flag: Int] {
230   pre = { prop1Pre[l, head, flag, i, n] }
231   program = {
232     init[l, head, next, curr];
233     repeat{
234       assume NegatetermCond[i,n];
235       (
236         assume flagSet[flag];
237         setValue1[val,curr]
238       )
239       +
240       (
241         assume flagNotSet[flag];
242         setValue2[val,curr]
243       );
244       insertl[l,head,next,curr] -- t = new; t->n = null; curr->n = t; curr = curr->n
245     };
246     assume termCond[i,n];
247     setValue3[val,curr]
248     }
249   post = { prop1Post[l', head', next', val', flag'] }
250 }
```

A.3. División de una lista en dos

```
1  -- splice.dals
2
3
4  -- list : List
5  -- current : Current
6  -- prev : Prev
7  -- v1 : x
8  -- v2 : y
9  -- v3 : z
10 -- getValue : value
11 -- next : next
12 -- head : head
13
14 module splice
15
16 sig List {}
17 sig Node {}
18 one sig NullValue { }
19
20 -- auxiliary preds to represent true and false
21 pred TruePred[] {
22 }
```

```
23
24  pred FalsePred[] {
25    not TruePred[]
26  }
27
28  pred postInit[l,l': List, head,head': List -> one (Node+NullValue), curr,curr': Node+NullValue] ↩
        {
29      curr' = l.head and l = l' and head = head'
30  }
31
32  pred preInit[l: List, head: List -> one (Node+NullValue), curr: Node+NullValue] {
33
34  }
35
36
37  pred postAstempcurr[next,next': Node -> one (Node+NullValue), curr,curr': Node+NullValue, temp,↩
        temp': Node+NullValue] {
38      temp' = curr and curr' = curr.next and next = next'
39  }
40
41  pred preAstempcurr[next: Node -> one (Node+NullValue), curr: Node+NullValue, temp: Node+↩
        NullValue] {
42
43  }
44
45
46  pred postInccurr[next,next': Node -> one (Node+NullValue), curr,curr': Node+NullValue] {
47      curr' = curr.next and next' = next
48  }
49
50  pred preInccurr[next: Node -> one (Node+NullValue), curr: Node+NullValue] {
51
52  }
53
54
55  pred postInsertl[l,l': List, head,head': List -> one (Node+NullValue), next,next': Node -> one (↩
        Node+NullValue), temp,temp': Node+NullValue] {
56      next' = next ++ (temp->l.head) and head' = head ++ (l->temp) and l = l' and temp = temp'
57  }
58
59  pred preInsertl[l: List, head: List -> one (Node+NullValue), next: Node -> one (Node+NullValue),↩
        temp: Node+NullValue] {
60
61  }
62
63  pred valThree[val: Node -> one Int, curr: Node+NullValue] {
64    curr.val = 3
65  }
66
67  pred NegatevalThree[val: Node -> one Int, curr: Node+NullValue] {
68    !valThree[val,curr]
69  }
70
71  pred valTwo[val: Node -> one Int, curr: Node+NullValue] {
72    curr.val = 2
73  }
74
75  pred NegatevalTwo[val: Node -> one Int, curr: Node+NullValue] {
76    !valTwo[val,curr]
77  }
78
79  pred valOne[val: Node -> one Int, curr: Node+NullValue] {
80    curr.val = 1
81  }
82
83  pred NegatevalOne[val: Node -> one Int, curr: Node+NullValue] {
84    !valOne[val,curr]
85  }
86
87
88  pred notEmpty[list: List, head: List -> one (Node+NullValue)] {
89    (list.(head) != NullValue)
90  }
91
92  pred NegatenotEmpty[list: List, head: List -> one (Node+NullValue)] {
93          not notEmpty[list, head]
94  }
```

```
95
96  -- shape analysis preds
97  pred cicn[list: List, head: List -> one (Node+NullValue), next: Node -> one (Node+NullValue)] {
98    some n: list.head.*next | n in n.next.*next
99  }
100
101 pred Negatecicn[list: List, head: List -> one (Node+NullValue), next: Node -> one (Node+↩
        NullValue)] {
102   !cicn[list,head,next]
103 }
104
105 pred isn[list: List, head: List -> one (Node+NullValue), next: Node -> one (Node+NullValue)] {
106   all n: list.head.*next | !(some n1, n2: list.head.*next | n1 != n2 and n1->n in next and n2->n↩
          in next)
107 }
108
109 pred Negateisn[list: List, head: List -> one (Node+NullValue), next: Node -> one (Node+NullValue↩
        )] {
110   !isn[list,head,next]
111 }
112
113 pred currReach[list: List, head: List -> one (Node+NullValue), next: Node -> one (Node+NullValue↩
        ), curr: Node+NullValue] {
114   curr != NullValue and
115   curr in list.head.*next
116 }
117
118 pred NegatecurrReach[list: List, head: List -> one (Node+NullValue), next: Node -> one (Node+↩
        NullValue), curr: Node+NullValue] {
119   !currReach[list,head,next,curr]
120 }
121
122 pred tempReach[list: List, head: List -> one (Node+NullValue), next: Node -> one (Node+NullValue↩
        ), temp: Node+NullValue] {
123   temp != NullValue and
124   temp in list.head.*next
125 }
126
127 pred NegatetempReach[list: List, head: List -> one (Node+NullValue), next: Node -> one (Node+↩
        NullValue), temp: Node+NullValue] {
128   !tempReach[list,head,next,temp]
129 }
130 -- end of shape analysis preds
131
132 pred prop1Pre [l,l1,l2: List, head: List -> one (Node+NullValue), next: Node -> one (Node+↩
        NullValue), val: Node -> one Int] {
133   (all n1, n2: Node | n1 in l.head.*next and n2 in l.head.*next and n1.next = n2 =>
134                                                                  (
135                                                                        (n2.next = NullValue => (n2.val↩
                                                                             = 3)) and
136                                                 (n2.next!= NullValue => ((n1.val = 1 and n2.val = 2) or↩
                                                      (n1.val = 2 and n2.val = 1))  )
137                                               )
138   ) and ((l.head.next = NullValue and l.head.val = 3) or (l.head.next != NullValue and l.head.↩
          val = 1))
139   and
140   Negatecicn[l,head,next]
141   and
142   isn[l,head,next]
143   and l1.head = NullValue and l2.head = NullValue and l1 != l2
144   and l.head != NullValue
145 }
146
147 pred prop1Post [l1: List, l2: List, head: List -> one (Node+NullValue), next: Node -> one (Node+↩
        NullValue), val: Node -> one Int] {
148    (all n: Node | (n in l1.head.*next => n.val = 1) and
149    all n: Node | (n in l2.head.*next => n.val = 2))
150 }
151
152 pred Negateprop1Post [l1: List, l2: List, head: List -> one (Node+NullValue), next: Node -> one ↩
        (Node+NullValue), val: Node -> one Int] {
153    !prop1Post[l1,l2,head,next,val]
154 }
155
156 pred diff [l1: List, l2: List] {
157   l1 != l2
158 }
```

```
159
160 pred Negatediff [l1: List, l2: List] {
161   !diff[l1,l2]
162 }
163
164
165 assertCorrectness prop1 [l: List, head: List -> one (Node+NullValue), next: Node -> one (Node+↩
        NullValue), val: Node -> one Int, curr: Node+NullValue, temp: Node+NullValue, l1: List, l2: ↩
        List] {
166   pre = { prop1Pre[l,l1,l2, head, next, val] }
167   program = {
168     assume notEmpty[l, head];
169     --init[l,head,curr];
170     assume preInit[l, head, curr];
171     curr := l.head;
172     repeat{
173       assume NegatevalThree[val, curr];
174         assume preAstempcurr[next, curr, temp];
175         temp:= curr;
176         curr:= curr.next;
177         (
178           ( assume valOne[val, temp];
179             assume preInsertl[l1, head, next, temp];
180             next := next ++ (temp->l1.head);
181             head := head ++ (l1->temp)
182           )
183           +
184           ( assume NegatevalOne[val, temp]
185           )
186         );
187         (
188           ( assume valTwo[val, temp];
189             assume preInsertl[l2, head, next, temp];
190             next := next ++ (temp->l2.head);
191             head := head ++ (l2->temp)
192           )
193           +
194           ( assume NegatevalTwo[val, temp]
195           )
196         )
197     };
198     assume valThree[val, curr]
199   }
200   post = { prop1Post[l1', l2', head', next', val'] }
201 }
```

Bibliografía

[1] *"Software Abstractions: Logic, Language, and Analysis".* D. Jackson

[2] *"Formal Methods: State of the Art and Future".* Edmund M. Clarke and Jeannette M. Wing

[3] *"An Axiomatic Basis for Computer Programming".* C.A.R Hoare

[4] *"Assigning meanings to programs", in Proceeding of a Symposium on Applied Mathematics, Vol. 19, Mathematical Aspects of Computer Sciencie, Schwartz,J.T.(ed.),19-32,1967.* Floyd, R. W

[5] *"Systems Simulation: The Art and Science". IEEE Transactions on Systems, Man and Cybernetics 6(10). pp. 723-724.* R. Shannon, J. Johannes.

[6] *"Software Testing. A Craftsman's Approach".* P. Jorgensen

[7] *"Formal Methods: Theory and Practice".* P. N. Scharbach

[8] *"A Discipline of Programming".* E. W. Dijkstra

[9] *"The Science of Programming".* D. Gries

[10] *"A Micromodularity Mechanism", en 9th. ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM Press, Vienna, Austria.*
D. Jackson, I. Shlyakhter, M. Sridharan

[11] *"Alloy Analyzer+PVS in the Analysis and Verification of Alloy Specifications", en 13th. International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Volume TDB, page TDB -Abril 2007.*
M. Frias, C. G. López Pombo, M. Moscato

[12] *"DynAlloy: Upgrading Alloy with Actions", en Proceedings of the 27 th International Conference on Software Engineering ICSE 2005, St.Louis,Missouri, USA,ACM Press,2005.*
M. Frias, J. P. Galeotti, C.G López Pombo, N .M. Aguirre

[13] *"Experience with Predicate Abstraction".* S. Das, D. Dill, S. Park

[14] *"Counter-Example Based Predicate Discovery in Predicate Abstraction".* S. Das, D. Dill

[15] *" Towards Abstraction for DynAlloy Specifications".* N. M. Aguirre, M. F. Frías, P. Ponzio, B. J. Cardiff, J. P. Galeotti, G. Regis

[16] *Effective CMM-Based Process Improvement.* Mark C. Paulk, Software Engineering Institute, 1996.

[17] *Dynalloy as a Formal Method for the Analysis of Java Programs.* J. P. Galeotti, M. Frias

[18] *Bounded Model Checking Using Satisfiability Solving.* Edmund Clarke , Armin Biere , Richard Raimi , Yunshan Zhu

[19] *Design Patterns: Elements of Reusable Object-Oriented Software.* Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides

[20] *The Definitive ANTLR Reference: Building Domain-Specific Languages.* Terence Parr

[21] *Sintaxis Dot: http://www.eclipse.org*

[22] *Sintaxis Dot: http://www.graphviz.org/doc/info/lang.html*