Implementación de Técnicas de Abstracción Para Especificaciones DynAlloy

Proyecto Final (Cód. 1998)
Carrera: Analista en 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
—

10 de diciembre de 2009
Director: Dr. Nazareno Aguirre
Co-Director: Lic. Pablo Ponzio

Índice general

Capítulo 1

Introducción

Desde tiempos previos a la llamada *crisis del software*, se ha reconocido que la complejidad y el tamaño 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, diseñar 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 [15]). 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 daños 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 años, 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 diseño, 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 diseñador, entre diseñador 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 pequeños 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 tamaño. 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 diseñar 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 tamaño 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 tamaño, 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*[17], 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 tamaño del grafo de estados puede ser exponencial al tamaño del programa. Un programa concurrente con k procesos puede tener un grafo de estados de tamaño `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 especificamente, 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 ˆr & 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.

ˆ 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

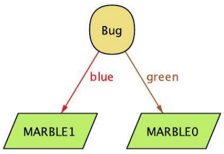


Figura 2.2: Instancia mostrada por el AlloyAnalyzer.

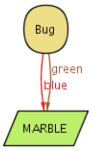


Figura 2.3: Otra instancia mostrada por el Alloy Analyzer.

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:

$$formula ::= \ldots \mid \underbrace{\{formula\}\ program\ \{formula\}}_{\textbf{“partial correctness”}}$$

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.

$$\begin{array}{lll} program ::= & (formula, formula)(\overline{x}) & \text{“acción atómica”} \\ & |formula? & \text{“test”} \\ & |program + program & \text{“choice no deterministico”} \\ & |program; program & \text{“composición secuencial”} \\ & |program^* & \text{“iteración”} \end{array}$$

Figura 3.1: Gramática para composición de acciones en DynAlloy.

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

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

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 diseño 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)

$$wlp : program \times formula \to formula$$

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:

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

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 $\{\alpha(\overline{y})\}P(\overline{y})\{\beta(\overline{y}, \overline{y'})\}$:

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

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 tamaño 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:

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

  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---------
 --auxiliars 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:

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

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:

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

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, pero no se tratará en el presente trabajo[1].

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.

---

[1] Véase Trabajos Futuros

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[2].

```
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'] }
```

---

[2] 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[3]. Esta vez, se conforma el vector como sigue:

- **empty**[s] ∧ **NegateonePred**[s] ∧ $TruePred[]$ ∧ `postAdd[s,s’]` ⇒ *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] ∧ **NegateonePred**[s] ∧ $TruePred[]$ ∧ `postAdd[s,s’]` ⇒ $Negateempty[s']$.

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

- **empty**[s] ∧ **NegateonePred**[s] ∧ $TruePred[]$ ∧ `postAdd[s,s’]` ⇒ *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] ∧ **onePred**[s] ∧ $Negateempty[s]$ ∧ `postDel[s,s’]` ⇒ *empty*[s'].

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

- **Negateempty**[s] ∧ **onePred**[s] ∧ $Negateempty[s]$ ∧ `postDel[s,s’]` ⇒ *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] ∧ **onePred**[s] ∧ $Negateempty[s]$ ∧ `postDel[s,s’]` ⇒ $NegateonePred[s']$.

  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

---

[3] 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[4].

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.

---

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

5.1. 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 carácterÃstica 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 mencionamos, esto tiene algunas restricciones, como la aplicación sólo a modelos finitos, y el problema 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.

5.2. Estructura y Diseño

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, ya

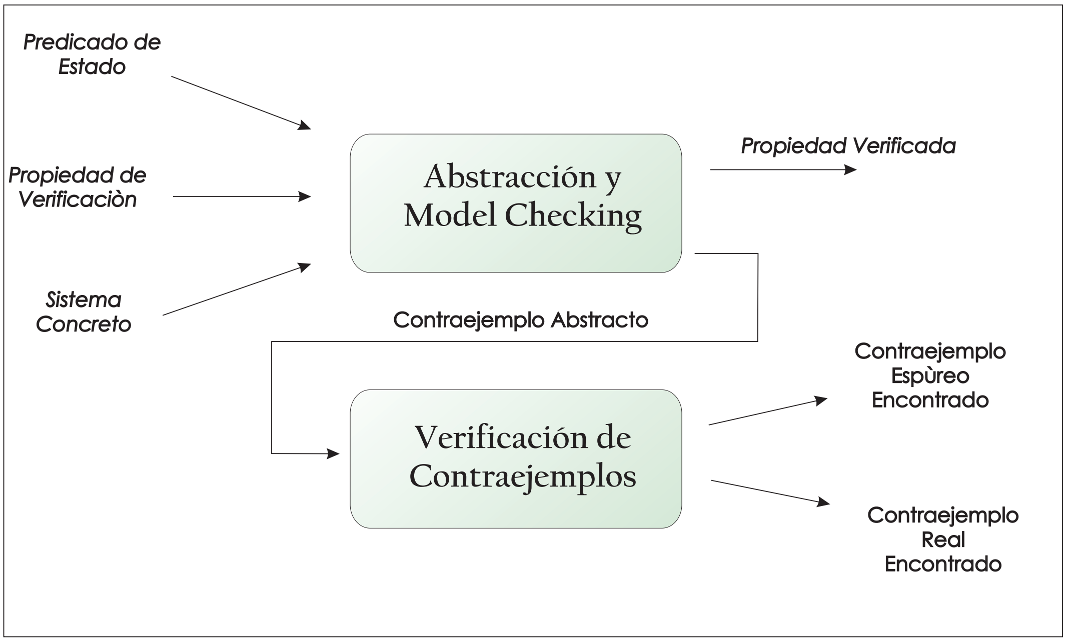


Figura 5.1: Algoritmo de Abstracción por Predicados.

sea real o espúreo (recordar que la parte de descubrimiento de predicados y refinamientos escapan de los objetivos de ésta parte del trabajo).

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.
- *AbstractorVisitor*: Es un módulo implementado usando el patrón de diseño *Visitor*, el cual es utilizado para recorrer el programa concreto y construir todas las trazas posibles de ejecución.
- *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.
- *AllTracesAbstractor*: Implementación de un algoritmo de abstracción, el cual es una especie de máquina abstracta, ya que toma como entrada todas las trazas construidas por AbstractorVisitor, y realiza la ejecución abstracta de las mismas. Al finalizar todas las posibles ejecuciones informa si encontró contraejemplos o la propiedad vale parcialmente (es decir, es válida para cierta cota).
- *OnDemandAbstractor*: Es la implementación de otro algoritmo de abstracción, el cual no utiliza el AbstractorVisitor para construir todas las trazas, si no que solamente genera 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.
- *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.

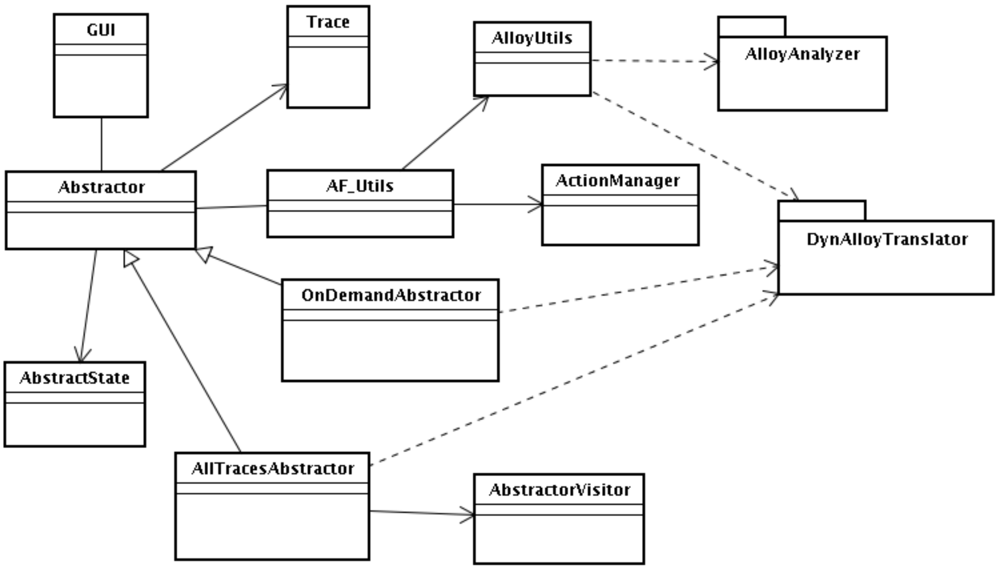


Figura 5.2: Diagrama de Clases Simplificado de la Herramienta.

- *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.

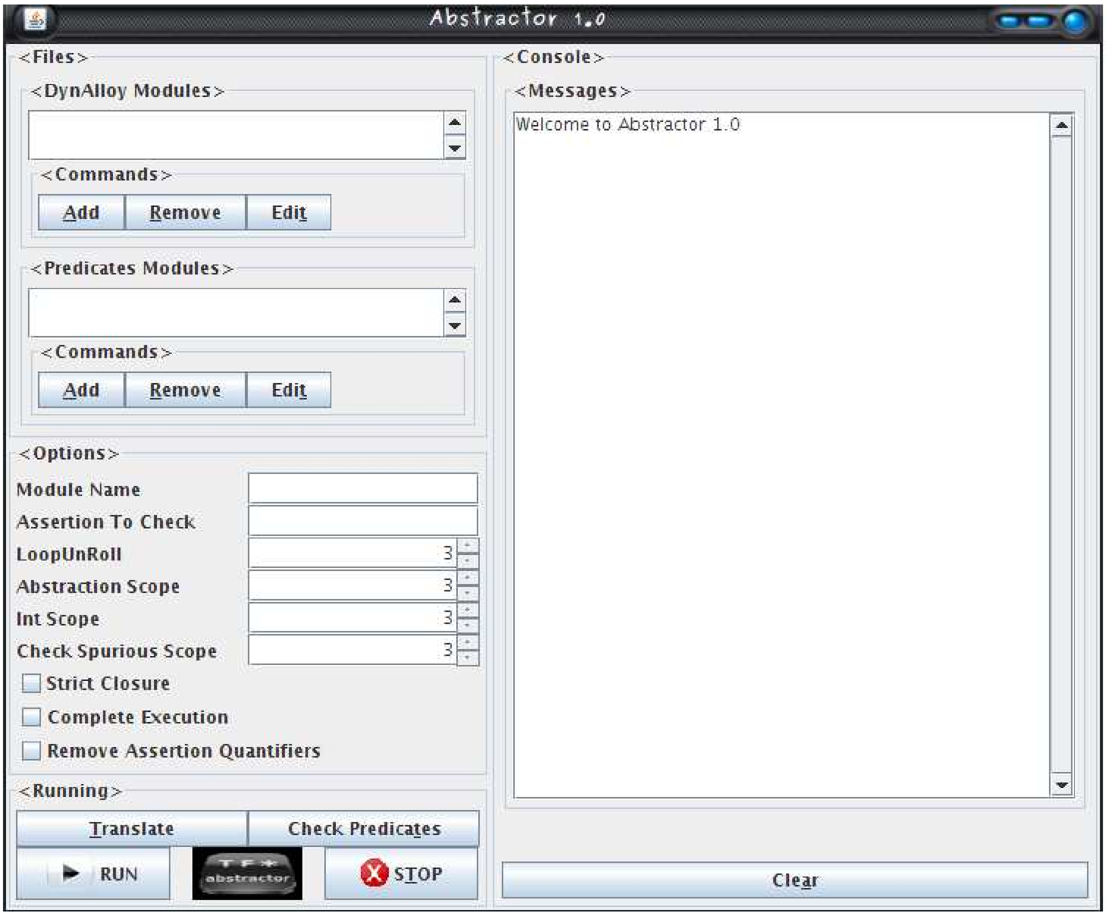
También vale la pena mencionar, que fue necesaria la implementación de un pequeño parser (utilizando gramáticas *ANTLR*), el cual verifica que los predicados de abstracción que el usuario ingresa están bien formados. De la verificación del módulo concreto se encargan las herramientas *DynAlloy Translator*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

En la figura 5.2 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.

5.3. 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.



Module Name
Assertion To Check
LoopUnRoll 3
Abstraction Scope 3
Int Scope 3
Check Spurious Scope 3
☐ Strict Closure
☐ Complete Execution
☐ Remove Assertion Quantifiers

Welcome to Abstractor 1.0

Figura 5.3: Interfaz de la Herramienta.

5.3.1. 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 5.3.

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.

Luego, se encuentran tres casillas de verificación, las cuales determinan elecciones del usuario. La primera, *Strict Closure*, si se encuentra seleccionada, indica que se correrá la abstracción para modelos con ciclos exactamente de la longitud que indica el parámetro Loop Unroll; caso contrario, se ejecutará para modelos con ciclos de longitud $0, 1, ..., LoopUnroll$. La casilla *Complete Execution*, indica qué algoritmo de abstracción se ejecutará. Si está marcada, se ejecutarán todas las trazas, mientras que de otra manera se ejecutará el algoritmo bajo demanda.

5.3.2. 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. Por último, el botón *Stop* sirve para detener la ejecución antes de que finalice.

5.3.3. 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:

```
===== 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 informarÃ¡ 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 *****"`. 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."`

5.3.4. 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[]`.

Otro problema a solucionar, fue el hecho de que para el parser de DynAlloy, la negación de un predicado no es un predicado, si no que es una fórmula. Entonces para poder invocar el método que tomaba como parámetro un predicado, se debía transformar la negación en ello. La manera de hacerlo fue pidiendo como restricción en el modelo DynAlloy que toma como entrada, que por cada predicado `p0[..]` definido (alcanza con que sólo sea para la definición de los predicados de abstracción), se declare un predicado Negatep0[..], con los mismos parámetros, y cuya definición sea la negación del predicado original (`!p0` o `not p0`). Ésto fue de gran utilidad a la hora de realizar la abstracción.

En esta etapa surge un nuevo 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.

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 RAM y sobre un Sistema Operaivo GNU/Linux 2.6. La versión de Alloy Analyzer utilizada fue la 4.1.8.

Se utilizó un modelo de listas, tal como se muestra en la figura 6.4, sobre el cual se quiso probar 2 propiedades : (P1) el método de eliminación preserva la aciclicidad de las listas, y (P2) los elementos eliminados durante el recorrido de la lista, no pueden aparecer antes de la posición corriente.

Sin usar abstracción, es decir, verificar esas propiedades con el Alloy Analyzer se obtuvieron los resultados mostrados en la tabla 6.1:

Propiedad	lurs	cota	tiempo
(P1)	20	21	18 min 36 seg
(P1)	24	25	no terminó
(P2)	15	16	27 min
(P2)	16	17	90 min 23 seg
(P2)	17	18	no terminó

Figura 6.1: Resultados obtenidos utilizando Alloy Analyzer.

Para realizar los experimentos utilizando abstracción se utilizaron dos implementaciones ya mencionadas de dicha técnica: **(A1)** *AllTracesAbstractor* y **(A2)** *OnDemandAbstractor*. Los predicados necesarios para poder utilizar abstracción son los siguientes:

Predicados para P1

1. noLoops[list, head, next, getValue]
2. notEmpty[list, head]
3. notNullNode[current]
4. notNullNode[prev]
5. noLoops[list,head,next++(prev->current.next),getValue]
6. noLoops[list,head++(list->current.next),next,getValue]

Predicados para P2

1. notAppearsBeforeCurrent[list,head,current,next,getValue,v1,v2,v3]
2. notEmpty[list, head]

3. notNullNode[current]
4. valueOk[current,v1,v2,v3,getValue]
5. notNullNode[prev]
6. notNullNode[prev]
7. partialPred1[current,prev,next]
8. partialPred2[list,head,current,next]
9. partialPred3[list,head,prev,next]
10. noLoops[list,head,next,getValue]

Las tablas 6.2 y 6.3 muestran los resultados obtenidos al verificar las propiedades utilizando las implementaciones A1 y A2 respectivamente. Considere las siguientes conveciones para interpretar lar tablas:

- **C.A.**: cota de abstracción.
- **C.E.**: cota para enteros.
- **C.S.**: cota para el chequeo de espureidad.
- **I.A.**: cantidad de veces que se invocó al Alloy Analyzer.
- **B.R.**: cantidad de veces que no se invocó a Alloy, mediante la reutilización de cálculos realizados anteriormente.

Propiedad	lurs	C.A.	C.E.	C.S.	tiempo	I.A.	B.R.
(P1)	18	3	3	19	7 min	507	13106888
(P1)	19	3	3	20	12 min 47 seg	508	26214088
(P1)	20	3	3	21	no terminó		
(P2)	18	3	3	19	12 min 19 seg	415	21495517
(P2)	19	3	3	20	no terminó		

Figura 6.2: Resultados obtenidos utilizando la técnica de abstracción A1.

Propiedad	lurs	C.A.	C.E.	C.S.	tiempo	I.A.	B.R.
(P1)	20	3	3	21	4 min 18 seg	477	52428513
(P1)	25	3	3	26	57 min 26 seg	492	1677721308
(P2)	20	3	3	21	4 min 23 seg	347	77594378
(P2)	25	3	3	26	76 min 18 seg	347	
(P2)	27	3	3	28	279 min 51 seg	347	

Figura 6.3: Resultados obtenidos utilizando la técnica de abstracción A2.

```
-- lists.dals

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

module lists

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

one sig List {}
sig Node {}
sig Char {}
one sig NullValue { }
notAppearsBeforeCurrent[list', head', prev', next', getValue', v1', v2', v3']
-- 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 )
}
-- no loops
pred noLoops[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))

}
pred NegatenoLoops[thisV:univ,
                   headV:univ -> one univ,
                   nextV : univ -> one univ,
                   valueV : univ -> one univ]
{
    not noLoops[thisV, headV, nextV , valueV]
}


-- no repeated elements
pred noRepeated[thisV:univ,
                headV:univ -> one univ,
                nextV : univ -> one univ,
                valueV : univ -> one univ]
{
 ( all n1, n2: Node |
             (n1+n2 in thisV.headV.(*nextV) and n1.valueV=n2.valueV)
                         implies n1=n2 )
}

-- v1, v2 and v3  do not appear on the list until 'prev' element including
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 ]
{
    ( all n: Node | n in (thisV.headV.(*nextV) - (currV.(*nextV))) implies
    n.valueV !in (current+prev+v1) )
}
-- 'prev' element precedes to current element
pred prevPrecedesCurrent[thisV:List,
                         headV:List -> one (Node + NullValue),
                         prevV: Node+NullValue,
                         currV: Node+NullValue,
                         nextV : Node -> one (Node + NullValue)]
{
    ((prevV = NullValue and currV = thisV.headV) or (prevV != NullValue and prevV.nextV = currV))
}

-- no loops and current always is reachable from head
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)]
{
  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))
}

-- Init Action
pred PpostInit[ 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),
                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)]
{
   (postInit[list, head, current', prev'] && (list' = list && v1' = v1 && v2' = v2 && v3' = v3 &&
            getValue' = getValue && next' = next && head' = head))
}

pred postInit[thisValue0: List,
              head0: List -> one (Node+NullValue),
              curr1: Node+NullValue,
              precurrent: Node+NullValue]
{
    (precurrent = NullValue && curr1 = thisValue0.head0)
}

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]
}

act init[ 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  { PpreInit[list, current, prev, v1, v2, v3, getValue, next, head]  }
   post  { PpostInit[list, current, prev, v1, v2, v3, getValue, next, head,
           list', current', prev', v1', v2', v3', getValue', next', head'] }
}
-- end of Init Action

-- setNext Action
pred PpostSetNext[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),
                  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)]
{
   ( overrideRelPost[next, next', prev, current.(next)] && current' = current.(next) &&
                    (list' = list && v1' = v1 && v2' = v2 && v3' = v3 && getValue' = getValue &&
                    prev' = prev && head' = 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)
}

act setNext[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 { PpreSetNext[list, current, prev, v1, v2, v3, getValue, next, head]  }
   post { PpostSetNext[list, current, prev, v1, v2, v3, getValue, next, head,
          list', current', prev', v1', v2', v3', getValue', next', head'] }
}
```

```
-- end of setNext Action

-- SetHead Action
pred PpostSetHead[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),
                  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)]
{
  ( postSetHead[list, current, next, head, head'] && current' = current.(next) &&
              (list' = list && prev' = prev &&  v1' = v1 && v2' = v2 && v3' = v3 &&
              getValue' = getValue && next' = next))
}
pred postSetHead[ thisValue0: List,
                  curr0: Node+NullValue,
                  next0: Node -> one (Node+NullValue),
                  head0: List -> one (Node+NullValue),
                  head1: List -> one (Node+NullValue)]
{
    head1 = head0 ++ (thisValue0 -> curr0.(next0))
}
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]
}
act setHead[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 { PpreSetHead[list, current, prev, v1, v2, v3, getValue, next, head]  }
  post { PpostSetHead[list, current, prev, v1, v2, v3, getValue, next, head,
         list', current', prev', v1', v2', v3', getValue', next', head'] }
}
```

```
-- end of SetHead Action

-- IncPrev Action
pred PpostIncPrev[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),
                  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)]
{
  ( postIncPrev[current, prev'] && current' = current.(next) &&
    (list' = list && v1' = v1 && v2' = v2 && v3' = v3 && getValue' = getValue &&
    next' = next && head' = head))
}
pred postIncPrev[curr0: Node+NullValue,
                precurrent: Node+NullValue]
{
   ( precurrent = curr0 )
}

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]
}

act incPrev[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 { PpreIncPrev[list, current, prev, v1, v2, v3, getValue, next, head]  }
   post { PpostIncPrev[list, current, prev, v1, v2, v3, getValue, next, head,
          list', current', prev', v1', v2', v3', getValue', next', head'] }
}
-- end of IncPrev Action
```

```
-- not empty list
pred notEmpty[list: List,
              head: List -> one (Node+NullValue)]
{
  (list.(head) != NullValue)
}

pred NegatenotEmpty[list: List,
                    head: List -> one (Node+NullValue)]
{
    not notEmpty[list, head]
}

-- item is not null
pred notNullNode[item: Node+NullValue] {
   ( item != NullValue )
}

pred NegatenotNullNode[item: Node+NullValue] {
    not notNullNode[item]
}

--- current value = v1 or current value = v2 or current value = v3
pred valueOk[ current: Node+NullValue,
              v1:  Char, v2: Char, v3: Char,
              getValue: Node -> one Char]
{
 (current.(getValue) in (v1+v2+v3))
}


-- prev is null
pred nullPrev[prev: Node+NullValue] {
  !notNullNode[prev]
}

-- not (current value = v1 or current value = v2 or current value = v3)
pred notValueOk[current: Node+NullValue,
                v1:  Char, v2: Char, v3: Char,
                getValue: Node -> one Char]
{
  !valueOk[current, v1, v2, v3, getValue]
}
-- current is null
pred nullCurrent[current: Node+NullValue] {
    !notNullNode[current]
}
-- RemAllCorrect Assertion
pred removeAllPre[thisV: univ,
                  valueV: univ -> one univ,
                  nextV: univ -> one univ,
```

```
                  headV: univ -> one univ]
{
    noCyclesNoRepeated[thisV,headV,nextV,valueV]

}

-- postcondition of the remAll operation: noCyclesNoRepeated and no iV's in the list
pred removeAllPost[ thisV: univ,
                    iVx0, iVy0, iVz0: univ,
                    valueV0: univ -> one univ,
                    nextV0: univ -> one univ,
                    headV0: univ -> one univ,
                    valueV1: univ -> one univ,
                    nextV1: univ -> one univ,
                    headV1: univ -> one univ]
{
    noCyclesNoRepeated[thisV,headV1,nextV1,valueV1] &&
    ((thisV.headV1.(*nextV1)).valueV1 = (((thisV.headV0.(*nextV0)).valueV0)-(iVx0+iVy0+iVz0)))
}

-- assert to check
-- initially not exists loops in list
-- then the nodes that contains one of the values v1, v2 or v3, are deleted from the list
-- Finally, the list should not contain loops
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 = {
    [ notEmpty[list, head] ]?;
    init[list,current,prev,v1,v2,v3,getValue,next,head];
    (
      (
        [notNullNode[current]]?;
      (
        (
          (
            [valueOk[current, v1, v2, v3, getValue]]?;
            (
              (
                [notNullNode[prev]]?;
                setNext[list,current,prev,v1,v2,v3,getValue,next,head]
              )
              +
              (
                [nullPrev[prev]]?;
                setHead[list,current,prev,v1,v2,v3,getValue,next,head]
```

```
              )
            )
          )
          +
          ([notValueOk[current, v1, v2, v3, getValue] ]?;
          incPrev[list,current,prev,v1,v2,v3,getValue,next,head]
          )
        )
      )
    )*
  );
  [nullCurrent[current]]?
  }
 post = { P1 o P2  }
}

P1 = noLoops[list’, head’, next’, getValue’]
P2 = notAppearsBeforeCurrent[list’, head’, prev’, next’, getValue’, v1’, v2’, v3’]
```

Figura 6.4: Módulo lists.dals.

Capítulo 7

Conclusiones y trabajo futuro

7.1. Conclusiones

Se presentó una forma de aplicar abstracción por predicados a especificaciones DynAlloy, para mejorar el análisis de las mismas. La abstracción por predicados es un mecanismo particular de abstracción, la cual es una de las técnicas más difundidas para poder contrarrestar el problema de la explosión combinatoria de estados. La implementación en el presente trabajo está basada en el mecanismo expuesto en [14], y se ha aplicado dicha técnica a especificaciones DynAlloy, el cual es un lenguaje que extiende a Alloy [1] para permitir expresar aspectos dinámicos de los sistemas, como es presentado en [12]. Es por esto, que fue necesaria la utilización de las herramientas *DynAlloy Translator* y *Alloy Analyzer*.

Una característica interesante de la herramienta es que permite aplicar la abstracción automáticamente. Dado un programa especificado en DynAlloy y un conjunto de predicados de estado sobre el mismo, se construye un modelo abstracto que representa dicho programa. Como ya fue mencionado, el método de abstracción utilizado es *débil*, es decir que si la propiedad es válida en el modelo abstracto, la propiedad también vale en el modelo concreto, pero por otro lado, si la propiedad no es válida en el modelo abstracto no necesariamente es inválida en el modelo concreto. En este caso decimos que se trata de un *contraejemplo espúreo*.

Se presentaron dos algoritmos de abstracción: el primero utiliza el árbol de parsing como estructura de control y ejecuta la abstracción para todas las trazas posibles en búsqueda de contraejemplos. El segundo, realiza una generación de trazas *bajo demanda*, es decir que genera una traza e intenta violar la propiedad para dicha traza, y sólo en caso de que la propiedad no sea violada, genera una nueva. En caso de que la mismsa resulte inválida, la ejecución de la abstracción se detiene.

Hemos realizado pruebas experimentales para ambos algoritmos de abstracción sobre los mismos casos de estudio. Se observó que el primer algoritmo de abstracción es poco eficiente en cuanto a espacio computacional, por este motivo se buscó una segunda implementación, llegando al algoritmo bajo demanda con el cual se logró un mayor rendimiento. Además fue posible la optimización del procedimiento de abstracción en ambos algoritmos mediante la reutilización de cálculos en los casos que fuera posible, es decir se obtuvo una minimización de invocaciones a Alloy Analyzer.

Luego de analizar los resultados obtenidos, podemos observar que la herramienta nos permite aumentar las cotas de iteración y de dominios, así como también mejorar en alguna medida los tiempos de verificación con respecto a lo realizado con Alloy Analyzer. De todas maneras, aunque los resultados resultan prometedores, nos encontramos trabajando para que dichas mejoras sean más notorias y la complejidad introducida por la abstracción resulte más fructífera (véase *Trabajo Futuro*). Como contrapartida, nuestra herramienta arrastra los inconvenientes asociados al uso de verificación automática utilizando métodos formales en cuanto a los límites de verificación. Además, aún no contamos con un mecanismo automático de refinamiento de especificaciones para los casos en los cuales se obtienen contraejemplos espúreos.

7.2. Trabajo futuro

A pesar de contar con resultados bastante satisfactorios, aún quedan pendientes varias mejoras por implementar. Todavía es posible mejorar considerablemente el espacio computacional utilizado para la representación abstracta, para lo cual nos encontramos adaptando la herramienta a una representación de modelos abstractos mediante un grafo, lo que además resulta más práctico a la hora de detectar ciclos (es decir evitar el proceso de abstracción en el caso de que ejecutar una secuencia de acciones que parte de determinada pre-condición, nos lleve a obtener siempre la misma post-condición). Esta última optmización sería de gran ayuda para mejorar los tiempos empleados en la verificación.

Otra mejora que se encuentra en desarrollo es una optmización del módulo encargado de almacenar los cálculos previos. La idea es construir un reticulado de las ejecuciones ya computadas, lo que mejoraría considerablemente las operaciones de búsqueda e inserción de estados abstractos.

Por último, existen predicados de estado que no se ven modificados por las acciones que se ejecutan (es decir la intersección entre sus parámetros actuales es vacía), por lo que en este caso la aplicación de la abstracción carece de sentido y lo más óptimo es evitarla.

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

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

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