Departamento de Cs. e Ingeniería de la Computación
Universidad Nacional del Sur

Métodos Formales para Ingeniería de Software

Resumen Teórico N° 3

Alloy

Segundo Cuatrimestre de 2022

Adelantamos al comienzo del curso que íbamos a utilizar una herramienta de nombre *Alloy*. Hay que utilizar el nombre con cuidado, ya que hace referencia a más de una cosa. Alloy es un lenguje formal de modelado pero también es el nombre de la herramienta de verificación. El verificador suele llamarse **Alloy Analyzer**, y es el encargado de analizar de manera automática propiedades sobre modelos definidos en Alloy (lenguaje de especificación).

Las especificaciones en Alloy se escriben en modo texto, y dentro de la herramienta pueden visualizarse en forma gráfica. Las representaciones visuales son similares a los diagramas de clase UML o los diagramas de entidad-relación. Como todo lenguaje, tiene definida su sintaxis y semántica formal.

Alloy fue creado con el propósito de hacer la verificación de modelos a través del uso de una herramienta que fuera:

- *Liviana:* el tamaño y la facilidad de uso fueron tenidos particularmente en cuenta. Que se trate un único archivo sin más requerimientos que la máquina virtual de JAVA y que tenga todo lo necesario en el mismo ambiente facilitan su uso. Resulta también liviano en el sentido de que se pueden expresar las propiedades más comunes de manera natural.

- *Precisa:* el lenguaje tiene una semántica matemática simple y uniforme.

- *Tratable:* el análisis semántico del lenguaje puede resolverse de manera eficiente y completamente automatizada (siempre y cuando se establezcan limitaciones de alcance o *scope* adecuadas).

Al momento de diseñar el lenguaje los diseñadores buscaron inspiración en los lenguajes disponibles para modelado. Es así como *Alloy* está relacionado con lenguajes como *UML* y *Z*.

El vínculo con UML se nota particularmente en los diagramas y forma de navegación de Alloy. Estas características están fuertemente inspiradas en UML. Si bien en UML también hay una notación gráfica y se pueden expresar restricciones lógicas, no puede decirse que el lenguaje sea preciso o liviano. Claramente UML tiene nociones de modelado que Alloy dejó de lado, como diagramas de estado, casos de uso, etc.

Por otra parte, el vínculo con *Z* aparece por el lado de la formalidad. *Z* es un lenguaje de especificación muy rico y preciso, pero desgraciadamente intratable. *Z* es más expresivo que Alloy pero notablemente más complejo, tanto como para que un verificador del primero sea imposible de construir. Los diseñadores de *Alloy* se inspiraron en *Z* para definir su semántica basada en conjuntos.

Tanto el lenguaje como el analizador fueron desarrollados por Daniel Jackson y su grupo de trabajo en el MIT. La documentación y versión actual de Alloy está disponible en: `http://alloytools.org`

Referencias:

- *"Alloy: A Lightweight Object Modeling Notation"* Daniel Jackson, ACM Transactions on Software Engineering and Methodology (TOSEM), Volume 11, Issue 2 (April 2002), pp. 256-290.

- "*Software Abstractions: Logic, Language and Analysis*" by Daniel Jackson. MIT Press, 2012 (o la version original de 2006)

1. Alloy: el lenguaje

Como lenguaje de especificación formal fue diseñado con la idea de poder definir modelos de datos orientados a objetos. Esta particularidad puede notarse en la facilidad para modelar datos en general, sin importar el dominio de individuos y las relaciones entre ellos. En particular, resulta natural especificar clases y asociaciones entre ellas, como así también restricciones sobre esas asociaciones.

1.1. ¿Cómo es el lenguaje Alloy?

El lenguaje de especificación está inspirado en las nociones presentadas en el resumen anterior.

Como mencionamos en la introducción, hay un vínculo con la programación orientada a objetos; muchas veces el punto de partida es un diagrama UML, habitualmente un diagrama de clases. Al traducirlo al modo texto del lenguaje Alloy notaremos que cada clase se corresponde a un conjunto base, es decir, un conjunto de objetos. Los objetos son una entidad abstracta, atómica e inmodificable, por lo que terminan representados a través de átomos lógicos. La estructura interna de los objetos no es relevante, pero lo que sí resultará relevante son las relaciones entre los objetos.

La semántica de los modelos estará determinada por tres aspectos:

- La pertenencia de los átomos a conjuntos.
- Las relaciones entre átomos.
- El cambio de los átomos y relaciones con el transcurso del tiempo.

La herramienta requiere de un lenguaje para escribir el modelo (átomos, conjuntos y relaciones) y un lenguaje para expresar propiedades y/o restricciones.

Todo modelo en Alloy se construye a partir de *átomos* y *relaciones*. El *átomo* es la entidad primitiva. Como mencionamos anteriormente, es abstracto, atómico, inmodificable e ininterpretable. Una *relación* es un estructura que relaciona átomos, de esta forma cada tupla es una secuencia de átomos.

Si consideramos por ejemplo una biblioteca, podemos pensarla como que contiene una relación que mapea títulos a autores. Concretamente, si pensamos en una bibioteca tendríamos tres relaciones: algunas unarias, una binaria y una ternaria. Específicamente, contaríamos con tres relaciones unarias (o conjuntos base):

- $Biblioteca = \{(B0)\}$
- $Libro = \{(L0), (L1), (L2)\}$

- $Autor = \{(A0), (A1)\}$

Estas relaciones unarias están representas a través de tuplas con un único átomo. Por ejemplo, en este caso, tendríamos una biblioteca, tres libros y dos autores.

La siguiente relación binaria asociaría libros con autores:

$$escritoPor = \{(L0, A0), (L1, A1), (L2, A0)\}$$

Continuando con nuestro ejemplo, tendríamos que cada libro se encuentra vinculado a un autor, donde el primer y el tercer libro estarían escritos por el mismo autor. Por último la relación ternaria asociaría la biblioteca a libros y autores:

$$coleccion = \{(B0, L0, A0), (B0, L1, A1)\}$$

Los contenidos de los conjuntos son sólo un ejemplo. Podríamos asociar la cantidad de átomos que queramos y mantener las relaciones como se nos ocurra.

Como mencionamos en el resumen anterior, toda relación tiene dos valores asociados: el *tamaño de la relación (#)* y la *aridad de la relación*. El tamaño hace referencia a la cantidad de elementos o tuplas en la relación, es decir, su cardinalidad. Por otra parte, la aridad hace referencia a la cantidad de átomos que hay en cada tupla de la relación. Si volvemos la mirada al ejemplo anterior podemos decir que $Biblioteca = \{(B0)\}$ es una relación de aridad 1 o unaria con tamaño o cardinalidad 1, mientras que $coleccion = \{(B0, L0, A0), (B0, L1, A1)\}$ es una relación de aridad 3 o ternaria con tamaño o cardinalidad 2.

Cada modelo tiene una cantidad (de momento ilimitada) de instancias asociadas. Dentro de la herramienta, la cantidad de instancias posibles depende la las limitaciones de **scope** (que discutiremos más adelante). La instancia deja establecido el contenido de los conjuntos y relaciones del modelo. Veamos algunas instancias del siguiente modelo (los detalles de la sintaxis los discutimos en la siguiente subsección):

sig $Biblioteca\{coleccion : Libro \rightarrow Autor\}$
sig $Libro\{escritoPor :$ **set** $Autor\}$
sig $Autor\{\}$
sig $Novelista, Poeta, Periodista$ **extends** $Autor\{\}$

Una instancia en particular resuelve qué es lo que va en lugar de ?? a continuación, respetando lo que indique el modelo:

$$\begin{aligned} Biblioteca &= \{??\} \\ Libro &= \{??\} \\ Autor &= \{??\} \\ Novelista &= \{??\} \\ Poeta &= \{??\} \\ Periodista &= \{??\} \\ coleccion &= \{??\} \\ escritoPor &= \{??\} \end{aligned}$$

Veamos algunas instancias concretas:

$$\begin{aligned} Biblioteca &= \{(B0)\} \\ Libro &= \{(L0), (L1)\} \\ Autor &= \{(A0), (A1), (A2)\} \\ Novelista &= \{\} \\ Poeta &= \{\} \\ Periodista &= \{\} \\ coleccion &= \{\} \\ escritoPor &= \{\} \end{aligned}$$

$$
\begin{aligned}
Biblioteca &= \{(B0)\} \\
Libro &= \{(L0), (L1), (L2)\} \\
Autor &= \{(A0), (A1), (A2), (A3)\} \\
Novelista &= \{(A0)\} \\
Poeta &= \{(A1), (A2)\} \\
Periodista &= \{(A3)\} \\
coleccion &= \{(B0, L2, A3), (B0, L2, A0)\} \\
escritoPor &= \{(L1, A2)\}
\end{aligned}
$$

Las instancias son auto-contenidas: los átomos que se mencionan en ellas dependen de los átomos que haya en los conjuntos base. Esto significa que en las relaciones no pueden aparecer átomos que no estén en los conjuntos base de esa instancia.

¿Cómo definimos modelos como el del ejemplo? El lenguaje provee varios mecanismos de especificación:

- Signaturas
- Campos
- Hechos
- Aserciones
- Comandos y alcances
- Predicados y funciones

Los primeros están destinados a definir los conjuntos base y las relaciones. Los siguientes nos permiten refinar y verificar lo expresado a través de los conjuntos y relaciones.

1.2. Cuestiones básicas

Para arrancar cualquier tipo de modelado resulta fundamental definir conjuntos y relaciones. Alloy utiliza la noción de signatura para definir conjuntos y la noción de campo para definir relaciones.

1.2.1. Signaturas o signatures

Una signatura representa un conjunto base, es decir, define la semántica de un conjunto de átomos.

La sintaxis para defnir una signatura es:

sig *nombre*

la palabra reservada **sig** seguida de un identificador *nombre*. De esta manera, por ejemplo, la declaración **sig** A define un conjunto de nombre A.

Las signaturas declaradas de manera independiente se conocen como **top-level**. Este tipo de signaturas son disjuntas entre sí.

Como mencionamos anteriormente, el lenguaje está vinculado al paradigma de orientación a objetos, y es por ello que el lenguaje de especificación provee dos maneras de modificar la semántica del conjunto base:

- Posibilidad de definir signaturas como extensiones de otra:

  La sintaxis hace referencia a la palabra reservada **extends**. Por ejemplo:

sig A, B **extends** C

La semántica indica que todos los átomos de A y B son átomos de C, y también implica que A y B son **disjuntos**.

- Posibilidad de definir signaturas abstractas:

  La sintaxis en este caso apela a la palabras reservada **abstract**. Desde el punto de vista semántico, si una signatura está definida como abstracta no posee átomos propios, ajenos a las signaturas que la extienden, a menos que no tengan signaturas que la extiendan.

abstract sig A

Existe también una manera de utilizar el vínculo de subconjunto entre conjuntos base. En este caso la motivación está en el uso de lógica relacional subyacente. La sintaxis en este caso apela a la palabra reservada **in**:

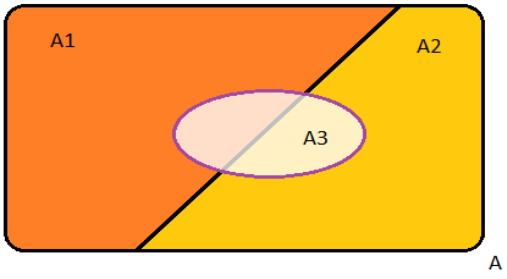
sig $A1$ **in** A

De esta manera se indica que los átomos de $A1$ son un subconjunto de A, es decir, $A1 \subseteq A$. Tengan presente que la definición de varias signaturas a través de **in** no garantiza que sean disjuntas como en el caso del **extends**.

Analicemos algunos ejemplos dejar claros estos detalles:

abstract sig A {}
sig B {}
sig $A1$ **extends** A {}
sig $A2$ **extends** A {}
sig $A3$ **in** A {}

En este caso A y B son signaturas top-level y por lo tanto son disjuntas. Los átomos de las signaturas $A1$, $A2$ y $A3$ son átomos de A. En particular, como $A1$ y $A2$ son extensiones de A y A es abstracta, entre las dos cubren todos los átomos de A. Además, $A1$ y $A2$ son disjuntas. Como $A3$ es un subconjunto de A, en este caso puede tener átomos de $A1$ y/o de $A2$. La siguiente representación gráfica ilustra la situación:



Consideremos ahora la siguiente variación del modelo anterior:

abstract sig A {}
abstract sig B {}
sig $A1$ **extends** A {}
sig $A2$ **extends** A {}
sig $A3$ **in** A {}
sig $B1$ **in** B {}

Si modificamos la signatura B haciéndola abstracta y con un subconjunto $B1$, lo que podemos notar es que A y B siguen siendo disjuntas. Pero, contrariamente a lo que sucede cuando una signatura abstracta es extendida, B puede tener átomos propios, es decir átomos que no pertenezcan a $B1$.

1.2.2. Campos o fields

Los *campos* o fields son la forma mediante la cual podemos declarar relaciones en Alloy. Las relaciones se definen dentro de una signatura, siendo un átomo de la misma quién ocupe la primera posición de las tuplas que se forman en esa relación (es decir, la signatura donde se coloca el campo será quien determine el dominio de la relación). Consideremos la siguiente definición:

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

se introduce una relación de nombre f tal que, como mencionamos en el párrafo anterior, su dominio es o está incluido en A y el rango está dado por la expresión e. Así, por ejemplo:

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

define una relación binaria, subconjunto de $A \times A$, mientras que

$$\textbf{sig } B \ \{f : A \text{ -> } A\}$$

define una relación ternaria, subconjunto de $B \times A \times A$.

1.2.3. Multiplicidades

El uso de modificadores de multiplicidad permite definir restricciones con respecto al tamaño de las relaciones, ya sean conjuntos (relaciones de aridad 1) o relaciones de aridad mayor a uno.

A la hora de modelar la multiplicidad con el lenguaje, se establece escribiendo una de las palabras destinadas a tal fin, afectando a una signatura o campo.

Alloy cuenta con cuatro palabras reservadas o *keywords* para establecer limitaciones de multiplicidad:

- **set:** cualquier número
- **some:** uno o más
- **lone:** cero o uno
- **one:** exactamente uno

Cuando el modificador afecta a una signatura se coloca al principio de la definición:

$$\textbf{m sig } A\{\}$$

La semántica de esta acción es limitar el tamaño del conjunto, es decir la cantidad de átomos que define esa signatura particular.

Cuando la restricción se establece sobre los campos o fields se pueden generar las siguientes situaciones:

$$\textbf{sig } A \ \{f : \textbf{m } e\}$$
$$\textbf{sig } A \ \{f : e1 \textbf{ m -> n } e2\}$$

Estas limitaciones de multiplicidad pueden pensarse como una restricción, ya que lo que indican es la siguiente idea: *para todo átomo a perteneciente al conjunto A, las tuplas que quedan de f eliminando a pertencen al conjunto e*, o en lógica:

$$\forall a : A \mid a.f \in e$$

Cuando la expresión que denota al conjunto tiene como prefijo un modificador de multiplicidad, se está estableciendo la cantidad de elementos que quedan en $a.f$, es decir, cuántos elementos de e están asociados a a en la relación f. Esto es:

sig A $\{f$: **one** $e\}$ para cada átomo de $a \in A$ hay un elemento de e
sig A $\{f$: **some** $e\}$ para cada átomo de $a \in A$ hay al menos un elemento de e
sig A $\{f$: **lone** $e\}$ para cada átomo de $a \in A$ hay a lo sumo un elemento de e
sig A $\{f$: **set** $e\}$ para cada átomo de $a \in A$ hay cualquier cantidad de elementos de e

La multiplicidad por defecto para las relaciones binarias es *one*, es por eso que la omisión del modificador de multiplicidad resulta en:

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

Cuando se trata de relaciones que no son binarias, hay dos modificadores de multiplicidad m y n. Supongamos para nuestro análisis que $e1$ y $e2$ son dos conjuntos A y B

$$\textbf{sig } X\{f : A \ \textbf{m} \ \text{->} \ \textbf{n} \ B\}$$

El significado general puede reducirse a dos restricciones:

$$\forall x : X, \forall a : A \mid n \ a.(x.f)$$
$$\forall x : X, \forall b : B \mid m \ (x.f).b$$

Teniendo en cuenta se puede notar que:

- A -> **one** B: define una función total con dominio en A.
- A **one** -> B: define una relación inyectiva cuyo rango es B.
- A -> **lone** B: define una función parcial sobre el dominio A.
- A **one** -> **one** B: define una función inyectiva con dominio A y rango B, también conocida como biyección de A a B.
- A **some** -> **some** B: define una relación con dominio en A y rango en B.

1.2.4. Disjuntos

Los campos pueden agruparse si comparten la misma expresión. El lenguaje cuenta con una palabra reservada o keyword, **disj** en este caso, para indicar que los campos que aparecen agrupados son mutuamente disjuntos. Entonces, una declaración como

$$\textbf{sig } A \ \{ \ \textbf{disj } f, g : e\}$$

implica que la intersección de los conjuntos f y g es siempre vacía.

Tomemos el ejemplo de libro de Jackson "*Un gato tiene tres nombres que son diferentes entre sí*". Esta afirmación puede modelarse como:

$$\textbf{sig } Cat \{ \textbf{ disj } daily, peculiar, ineffable : Name\}$$
$$\textbf{sig } Name \{\}$$

De esta manera, para cualquier gato c, los conjuntos $c.daily$, $c.peculiar$, y $c.ineffable$ son diferentes nombres.

Este modificador también puede utilizarse sobre signaturas. Tiene sentido utilizarlos sólo en aquellas que están definidas como subconjunto de otras, utilizando **in**. Así por ejemplo, la definición:

$$\textbf{disj } Gato, Perro\textbf{in}Mascota \{\}$$

estaría garantizando que los conjuntos $Gato$ y $Perro$ no comparten átomos. Recuerden que la misma definición pero sin la keyword **disj** permite el solapamiento de los conjuntos.

Por último, la palabra reservada **disj** también puede utilizarse en la definición de restricciones adicionales del modelo, para identificar átomos que deban ser distintos entre sí.

1.3. Agregando restricciones

Los modelos que podemos definir con lo presentado en la sección anterior son irrestringidos. Lo son porque, a pesar de que probablemente se tenga conocimiento del dominio, no hay manera agregar este conocimiento con la sintaxis presentada hasta ahora. Si bien es habitual que en las primeras etapas del proceso de desarrollo nos encontremos con modeles sin restricciones, es igual de cierto que a medida que avancemos las restricciones van a aparecer y su incorporación al modelo es fundamental. En este sentido el analizador de Alloy es muy útil ya que provee una respuesta rápida sobre las debilidades del modelo, permitiendo de esta manera el refinamiento del mismo a través de la incorporación de restricciones que deben satisfacerse por las instancias del modelo. Este proceso continúa hasta que se alcance una situación donde estamos satisfechos con el modelo conseguido. Si bien con algunas de la palabras reservadas o keywords presentadas hasta ahora se pueden expresar algunas restricciones, resulta imposible agregar restricciones más interesantes, cosas simples como determinar que queremos que una relación sea simétrica o aspectos particulares sobre la naturaleza de la misma no pueden ser expresadas. *¿Como agregamos este tipo de restricciones entonces?*. Claramente seguir agregando sólo keywords no va a ser una solución, necesitamos dos cosas:

- el lenguaje lógico lo suficientemente rico para expresarlas y,
- la sintaxis apropiada para incorporar estas fórmulas lógicas en nuestro modelo.

Para ayudar en esta tarea de definir restricciones, Alloy agrega tres constantes: **none**, **univ** e **iden**. Cada una de estas constantes define un conjunto. Así, **none** representa al conjunto vacío; **univ** es el conjunto universal, es decir está formado por todos los átomos de una instancia particular del modelo; finalmente, **iden** es el conjunto identidad, donde sus elementos son tuplas que representan la relación de cada uno de los átomos de una instancia consigo mismos.

De esta manera para una instancia particular de un modelo, como por ejemplo:

- $Biblioteca = \{(B0)\}$
- $Libro = \{(L0), (L1), (L2)\}$
- $Autor = \{(A0), (A1)\}$

los conjuntos predefinidos quedarían definidos de esta manera:

$$\begin{aligned} \textbf{none} &= \{\} \\ \textbf{univ} &= \{(B0), (L0), (L1), (L2), (A0), (A1)\} \\ \textbf{iden} &= \{(B0, B0), (L0, L0), (L1, L1), (L2, L2), (A0, A0), (A1, A1)\} \end{aligned}$$

La semántica de las expresiones y restricciones termina siendo bastante simple de determinar ya que en Alloy **todo es un conjunto**.

1.3.1. El lenguaje lógico

Para poder trabajar Alloy incorpora una rica colección de cuantificadores. Recordemos que la lógica que utiliza cuenta con el poder la lógica de primer orden, pero extendida con el cálculo relacional.

all $x : S \mid F$	F se verifica para cada atómo x en S
some $x : S \mid F$	F se verifica para algún átomo x en S
no $x : S \mid F$	F no se verifica para cada átomo x en S
lone $x : S \mid F$	F se verifica para a lo sumo un átomo x en S
one $x : S \mid F$	F se verifica para exactamente un átomo x en S

Debemos tener cuidado con estas palabras reservadas, ya que en este contexto no son los modificadores de multiplicidad que vimos antes sino un cuantificador. Alloy determina cuál es por el contexto de uso. Fíjense que si afectan a una expresión tienen la semántica que vimos antes:

some e	e tiene tuplas, es decir, no es vacío
no e	e no tiene tuplas, es decir, es vacío
lone e	e tiene a lo sumo una tupla
one e	e tiene exactamente una tupla

Como mencionamos antes, todo es un conjunto en Alloy. No hay escalares, ni se pueden mencionar átomos o tuplas de relaciones directamente. Como esto es a veces necesario, el lenguaje nos provee una manera de definir relaciones "singleton"para poder escribir las restricciones que necesitemos. Esta relación singleton nos va a permitir identificar un átomo o tupla particular. Por ejemplo:

$$\textbf{let } Nombre = \textbf{one } Libro$$

Nos permite identificar un átomo particular del conjunto *Libro*. Esto mismo sucede cuando utilizamos los cuantificadores

$$\textbf{all } x : S \mid \ \ldots \ x \ \ldots$$

indica que $x = \{t\}$ para cada átomo de la signatura o conjunto S.

Habiendo clarificado este detalle y que, por lo tanto, la máxima de que todo es un conjunto siempre vale, podemos incorporar en nuestro lenguaje todas las operaciones de conjuntos para expresar restricciones:

- unión de conjuntos: $+$
- intersección de conjuntos: $\&$
- diferencia de conjuntos: $-$

- subconjuto: **in**
- igualdad de conjuntos: =

De esta manera, tomando el modelo de la biblioteca como ejemplo, la restricción "*Todos los autores periodistas de un libro*" podría expresarse como:

$$\{ \textbf{let } miLibro = \textbf{one } Libro \ \{miLibro.escritoPor \ \& \ Periodista\}\}$$

Puede notarse que en la definición apelamos a la intersección de conjuntos. Recordemos el modelo:

sig *Biblioteca* {*coleccion* : *Libro* –> *Autor*}
sig *Libro* {*escritoPor* : **set** *Autor*}
sig *Autor* {}
sig *Novelista*, *Poeta*, *Periodista* **extends** *Autor* {}

Contamos además con los operadores relacionales:

- producto: –>

  Este operador se aplica sobre relaciones y permite la construcción de una nueva relación. Si p y q son relaciones, $p \text{ -> } q$ también es una relación. Las tuplas de $p \text{ -> } q$ son tales que los primeros elementos son una tupla de p y los últimos una tupla de q. Veamos un ejemplo. Si tenemos la siguiente instancia:

  $$Biblioteca = \{(B0)\}$$
  $$Libro = \{(L0), (L1)\}$$
  $$Autor = \{(A0), (A1)\}$$

  el producto de *Libro* y *Autor*, llamémoslo *escritoPor*, sería:

  $$Libro \text{ -> } Autor = escritoPor = \{(L0, A0), (L0, A1), (L1, A0), (L1, A1)\}$$

  Luego, el producto entre *Biblioteca* y *escritoPor* sería:

  $$\underbrace{Biblioteca \text{ -> } escritoPor}_{Biblioteca \ \text{->} Libro \ \text{->} Autor} = \{(B0, L0, A0), (B0, L0, A1), (B0, L1, A0), (B0, L1, A1)\}$$

- traspuesta: $\sim$

  Esta operación está pensada para construir una relación que tiene la misma cantidad de tuplas que la relación original pero con las posiciones de los átomos invertidas. Así, por ejemplo, si tenemos la relación:

  $$ejemplo = \{(L0, A2, B0, C0), (L1, A2, B1, C3), (L2, A0, B3, C2)\}$$

  la traspuesta será:

  $$\sim ejemplo = \{(C0, B0, A2, L0), (C3, B1, A2, L1), (C2, B3, A0, L2)\}$$

- join o dot join: .

  Es *join* es una operación binaria definida sobre relaciones, tal que al menos una de ellas debe ser de aridad mayor a uno para que la operación esté definida.

  Sea p una relación de aridad m y q una relación de aridad n, el join será una relación de aridad $m+n-2$, tal que las tuplas que contiene surgen de combinar los primeros $m-1$ elementos de cada tupla de p y los últimos $n-1$ elementos de cada tupla de q, siempre y cuando el join entre las tuplas exista. Intuitivamente, el join existe si el último elemento de la tupla de p coincide con el primer elemento de la tupla de q. Más formalmente: para cada par de tuplas $(s_1, \ldots, s_m)$ en p y $(t_1, \ldots, t_n)$ en q

  - Si $s_m != t_1$ entonces no se agrega nada al conjunto $p.q$
  - Si $s_m = t_1$ entonces $(s_1, \ldots, s_{m-1}, t_2, \ldots, t_n)$ está en la relación $p.q$

  Veamos un par de ejemplos:

$$\underbrace{\{(a,b)\}}_{p}.\underbrace{\{(a,c)\}}_{q} = \{\}$$
$$\{(a,b)\}.\{(b,c)\} = \{(a,c)\}$$
$$\{(x,b),(y,b),(z,a),(z,c)\}.\{(b,d),(c,e)\} = \{(x,d),(y,d),(z,e)\}$$

  El primer ejemplo da como resultado el conjunto vacío ya que no hay ningún par de tuplas que satisfagan el segundo ítem de la definción. Noten que debe tratarse del mismo átomo, y la igualdad garantiza eso.

- box join:[ ]

  Este join resulta útil cuando queremos tomar los argumentos en orden inverso, ya que la semántica de la operación es idéntica a la del dot join, sólo que toma los argumentos de la operación en otro orden. La definición sería:

$$p[q] \equiv q.p$$

  De esta manera, una restricción como la que figura entre llaves a continuación:

$$\textbf{let } mateo = \textbf{one } Autor \ \{ \ mateo[escritoPor] \ \}$$

  que determina los libros de autor identificado como "mateo" es equivalente a

$$\textbf{let } mateo = \textbf{one } Autor \ \{ \ escritoPor.mateo \ \}$$

  Noten que, al utilizar el box join, los átomos que deben coincidir son el *primero* de la primer relación con el *último* de la segunda.

  Obervación: Esta equivalencia entre los dos tipos de join y el uso de la traspuesta, hacen que la misma expresión pueda escribirse de varias maneras equivalentes:

$$p[q] \equiv q.p \equiv \sim p. \sim q \equiv \sim q[\sim p]$$

- clausura transitiva: $\wedge$

  La definición intuitiva de este operador se reduce a la aplicación recursiva del operador join sobre la misma relación. El proceso recursivo termina cuando no se agregan más tuplas a la relación resultante.

  Sea $p = \{(s0, s1), (s1, s2), (s2, s3), (s4, s7)\}$ una relación de $S \times S$, la clausura transitiva será:

  $$^{\wedge}p = \{\underbrace{(s0, s1), (s1, s2), (s2, s3), (s4, s7)}_{p}, \underbrace{(s0, s2), (s1, s3)}_{p.p}, \underbrace{(s0, s3)}_{p.p.p}\}$$

  Esta operación puede utilizarse sólo sobre relaciones binarias.

- clausura reflexivo-transitiva: $\star$

  Las clausura reflexivo-transitiva, como lo indica su nombre, es el conjunto resultante de la clausura transitiva unida al conjunto identidad. Formalmente:

  $$^{\star}r \equiv ^{\wedge} r + iden$$

  Sea $p = \{(s0, s1), (s1, s2), (s2, s3), (s4, s7)\}$ una relación de $S \times S$, vimos que la clausura transitiva es:

  $$^{\wedge}p = \{(s0, s1), (s1, s2), (s2, s3), (s4, s7), (s0, s2), (s1, s3), (s0, s3)\}$$

  El conjunto identidad es:

  $$iden = \{(s0, s0), (s1, s1), (s2, s2), (s3, s3), (s4, s4), (s7, s7)\}$$

  Luego:

  $$^{\star}p = \{ \overbrace{\underbrace{(s0, s1), (s1, s2), (s2, s3), (s4, s7), (s0, s2), (s1, s3), (s0, s3),}_{iden} \underbrace{(s0, s0), (s1, s1), (s2, s2), (s3, s3), (s4, s4), (s7, s7)\}}}^{^{\wedge}p}$$

- restricción de dominio: $<:$

  Este operador se utiliza para filtrar relaciones a un dominio dado. Si s es un conjunto y r una relación, entonces $s<:r$ contiene las tuplas de r que comienzan con elementos que están en el conjunto s. Por ejemplo, dado el siguiente conjunto de nombres $Nombre = \{(N1), (N2)\}$ y la siguiente relación $libretaDir = \{(N0, DO), (N1, D1), (N1, D2), (N2, D3)\}$, una restricción de dominio sería:

  $$Nombre <: libretaDir = \{((N1, D1), (N1, D2), (N2, D3)\}$$

  es decir, se eliminan las tuplas cuyo primer átomo no esté en $Nombre$.

- restricción de rango: $:>$

  Este operador se utiliza para filtrar relaciones a un rango dado. Si s es un conjunto y r una relación, entonces $r:>s$ contiene las tuplas de r que terminan con elementos que están en el conjunto s. Por ejemplo, dado el conjunto $Dir = \{(D0), (D3)\}$ y la relación $libretaDir = \{(N0, DO), (N1, D1), (N1, D2), (N2, D3)\}$, una restricción de rango sería:

$$libretaDir :> Dir = \{(N0, DO), (N2, D3)\}$$

es decir, se eliminan las tuplas cuyo último átomo no esté en Dir.

- override: ++

  Esta operación es similar a la unión, con la salvedad de que algunas tuplas de la primer relación son "sobre-escritas" por tuplas de la segunda relación, ¿cuales? las que comparten el dominio. Es importante tener en cuenta que esta operación sólo es aplicable sobre relaciones de aridad mayor a uno (es decir, de aridad dos o más).

  Formalmente la operación se define como:

$$p{+}{+}q \equiv p \text{ - } (dominio(q) <: p) + q$$

  Es el conjunto p eliminando las tuplas que comparten dominio con q unido al conjunto q.

  Por ejemplo, dadas las siguientes relaciones:

$$viejaDir = \{(N0, DO), (N1, D1), (N1, D2)\}$$
$$nuevaDir = \{(N1, D4), (N3, D3)\}$$

  un posible override sería:

$$viejaDir{+}{+}nuevaDir = \overbrace{\{(N0, DO), (N1, D1), (N1, D2)\}}^{viejaDir} - \underbrace{\{(N1, D1), (N1, D2)\}}_{dominio(nuevaDir) <: viejaDir} + \underbrace{\{(N1, D4), (N3, D3)\}}_{nuevaDirreccion}$$

$$viejaDir{+}{+}nuevaDir = \{(N0, DO), (N1, D4), (N3, D3)\}$$

También están disponibles los operadores lógicos habituales:

- negación: **not** o !
- conjunción: **and** o &&
- disyunción: **or** o ||
- implicación lógica: **implies** o => (tengan presente que no es un condicional, la implicación lógica, $a => b$, es equivalente a **no** a **or** b)
- alternativa: **else**
- si y sólo si: <=>

Es importante prestar especial atención a la precedencia de los operadores.

1.3.2. Restricciones en Alloy

Una vez definido nuestro lenguaje lógico podemos escribir nuestras restricciones, pero nos falta la sintaxis para incorporarlas al modelo.

- **Hechos:** La primer forma es a través de *hechos* o facts. Un hecho es una restricción adicional sobre un conjunto o relación.

  Cuando el analizador de Alloy busque instancias, las mismas deben satisfacer todos los hechos que acompañen a nuestro modelo.

  Para especificar hechos se utiliza la palabra reservada **fact**.

  Teniendo en cuenta el modelo de la biblioteca, recodemos era:

  $$\begin{array}{l}\textbf{sig } \mathit{Biblioteca}\ \{\mathit{coleccion} : \mathit{Libro} \mathrel{->} \mathit{Autor}\}\\ \textbf{sig } \mathit{Libro}\ \{\mathit{escritoPor} : \textbf{set } \mathit{Autor}\}\\ \textbf{sig } \mathit{Autor}\ \{\}\\ \textbf{sig } \mathit{Novelista}, \mathit{Poeta}, \mathit{Periodista}\ \textbf{extends } \mathit{Autor}\ \{\}\end{array}$$

  Si queremos que en nuestro modelo no sea posible que haya *novelistas y poetas que escriban libros en común* la manera de lograr esto es agregando un hecho a nuestro modelo que evite esta situación. Podríamos lograrlo de la siguiente manera:

  $$\begin{array}{lll}\textbf{fact } \{ & \textbf{all } l : \mathit{Libro} \mid \textbf{no } a1, a2 : \mathit{Autor} \mid & a1 \textbf{ in } \mathit{Novelista} \textbf{ and}\\ & & a2 \textbf{ in } \mathit{Poeta} \textbf{ and}\\ & & a1 \textbf{ in } l.\mathit{escritoPor} \textbf{ and}\\ & & a2 \textbf{ in } l.\mathit{escritoPor}\\ \}\end{array}$$

- **Predicados:** Definen restricciones que el diseñador no desea guardar como hechos o plantean restricciones que pueden utilizarse en varios contextos. No afectarán a la búsqueda de instancias a menos que se utilicen en un hecho o como restricción de un comando de verificación.

  Un detalle interesante es que pueden parametrizarse. Para especificar predicados se utiliza la palabra reservada **pred**. Los argumentos de un predicado son especificados entre [ ], indicando el nombre de cada argumento y la expresión que determina el conjunto o relación a la que pertenecen.

  Por ejemplo, en el modelo de la biblioteca (1.3.2) podríamos querer tener un predicado que determine si un autor particular, identificado como a, escribió un dado libro, l. El predicado será verdadero en caso de que la tupla (l, a) esté en la relación *escritoPor* y será falso en caso contrario. Una versión de este predicado en Alloy sería:

  $$\textbf{pred } \mathit{esAutorDe}[a : \mathit{Autor}, l : \mathit{Libro}]\ \{\ a \textbf{ in } l.\mathit{escritoPor}\ \}$$

  Conseguimos entonces establecer una restricción nombrada con dos argumentos en este caso, aunque en general puede tener cero o más argumentos.

- **Funciones:** Tienen el mismo objetivo que los predicados sólo que a nivel expresión. Para especificar funciones se utiliza la palabra reservada **fun**. Los argumentos de una función son especificados de la misma forma que los argumentos de un predicado. Además, luego

de indicar los argumentos, se debe indicar la forma de las tuplas que definen el conjunto resultado. Esto puede ser directamente un conjunto, una relación o una expresión que los denote.

Utilizando el mismo ejemplo de la biblioteca (1.3.2), podemos querer definir una manera de *conseguir los libros que escribio un autor*. Se busca construir un conjunto de libros a partir de un autor, a. Una definición posibile sería:

$$\textbf{fun}\ escribio[a : Autor] : \textbf{set}\ Libro\ \{\ \{\ l : Libro \mid l\ \textbf{in}\ escritoPor.a\ \}\ \}$$

En este caso construimos explícitamente el conjunto en el cuerpo de la función, pero también podemos conseguir el conjunto que necesitamos apelando a operaciones de conjuntos. Así, si quisiéramos conseguir *los autores novelistas de un libro*, podríamos escribir:

$$\textbf{fun}\ autoresNovelistas[l : Libro] : \textbf{set}\ Novelista\ \{\ (l.escritoPor)\ \&\ Novelista\ \}$$

2. Alloy: el analizador

Como analizador, trabaja sobre las especificaciones escritas en el lenguaje Alloy. El analizador utiliza verificación limitada. Esta limitación esta asociada al número de objetos de cada clase. Esto significa que en cada análisis fija o limita los átomos a un número fijo por conjunto base (el límite se estable en cada análisis).

El analizador utiliza SAT-solvers para responder las consultas de verificación. Para poder utilizar esta estrategia convierte las consultas en fórmulas de lógica boolena e intenta encontrar una forma de satisfacerlas asignándoles valores de verdad (tarea que lleva a cabo el SAT-solver). Profundizaremos en esto en otro resumen.

2.1. Consiguiendo instancias

Para generar y visualizar instancias del modelo se utiliza el comando **run**. Su sintaxis es:

$$\textbf{run}\ \{restriccion\}\ scope$$

El scope limita el tamaño de las instancias a considerar, estableciendo una restricción a la cantidad máxima de átomos que puede tener cada signatura. El valor por defecto establecido por Alloy es 3, pero como mencionamos en la sintaxis, puede modificarse para cada comando run. Esta modificación puede ser global a todas las signaturas o puede establecerse para cada una de manera individual. En este último caso es necesario cubrir a todas las signaturas top-level. Algunos ejemplos:

run $\{\}$	comando run simple sin restricción adicional y con scope por defecto
run $\{\#Poeta > 1\}$ **for** 4	comando run con una restricción y modificación global del scope
run $\{$ **some** $Poeta\}$	comando run simple con restricción y scope por defecto.

El algoritmo que implementa el analizador trata siempre de buscar instancias pequeñas, lo más pequeñas que puede. Es por esta razón que muchas veces el run sin restricciones muestra instancias triviales o vacías.

2.2. Verificando propiedades

La verificación de propiedades del modelo se realiza a través de *aserciones*. Las *aserciones* expresan restricciones que se desea que el modelo verifique pero sin que sean limitaciones del mismo, por lo que no se incorporan de manera directa al modelo. La idea de una aserción es garantizar que una determinada propiedad es verificada por todas las instancias del modelo. Hacer una verificación instancia por instancia resulta inviable, por lo que la estrategia del analizador es hacer una prueba por el absurdo. De esta manera, si la aserción no se verifica, el analizador brinda un contra-ejemplo, es decir, una instancia del modelo donde la propiedad buscada no se satisface.

Para garantizar que el analizador pueda resolver la consulta, aquí también se establecen restricciones de scope. Por esta razón, el comentario del analizador al no encontrar contra-ejmplos es que la propiedad parece ser cierta. El analizador trabaja bajo la hipótesis de que si no aparecen contra-ejemplos con un scope dado (bastante bajo), la propiedad puede generalizarse.

Las aserciones se controlan a través del comando **check**, cuya sintaxis es:

check *A* **for** *scope*

Consideremos el siguiente modelo:

abstract sig *Object* {}
sig *Directorio* **extends** *Object* {}
sig *Archivo* **extends** *Object* {}
sig *Alias* **extends** *Archivo* {}

Observemos los comandos check, siendo *A* un predicado definido previamente o una restricción expresada en la lógica previamente descripta:

check *A* **for** 5 *Object*	válido ya que el scope cubre la única signatura top level
check *A* **for** 3 *Directorio*, 2 *Archivo*	válido ya que entre *Directorio* y *Archivo* cubren a *Object* (y los *Alias* son *Archivo*)
check *A* **for** 5 *Object*, 3 *Directorio*	válido ya que el scope cubre a *Object*
check *A* **for** 5 *Directorio*, 2 *Alias*, 5 *Archivo*	válido ya que, al igual que en el segundo check, *Directorio* y *Archivo* cubren a object
check *A* **for** 3 *Directorio*, 3 *Alias*	no es válido, ya que al no especificarse scope para *Object* ni para *Archivo*, la signatura *Object* no queda cubierta (pueden existir átomos de *Archivo* que sean *Object* y no sean *Directorio* ni *Alias*)

Se puede lograr el mismo efecto de la aserción buscando generar una instancia que satisfaga la negación de la propiedad planteada en la aserción (en cuyo caso el resultado esperado sería que no exista tal instancia). Si bien en términos prácticos se logra el mismo efecto, desde los aspectos de verificación resultaría más difícil visualizar qué comandos corresponden a la búsqueda de instancias de validación del modelo y cuáles están destinados a verificar propiedades.

Apéndice: Reglas de precedencia y asociatividad

Esta misma información puede encontrarse en la página 263 del libro de Jackson.

Los operadores en las expresiones se resuelven hacia la derecha, aplicando primero el de mayor precedencia. El orden de resolución sería:

- **operadores unarios**:· $\sim$, $\wedge$ y $*$;
- **dot join**: . ;
- **box join**:[];
- **operadores de restricción**: <: y :>
- **producto**: ->;
- **intersección**: &;
- **override**: ++;
- **cardinalidad**: #;
- **unión y diferencia**: + y −;
- **expresiones cuantificadas y multiplicidades**: *no*, *some*, *lone*, *one*, *set*;
- **operadores de comparación por negación**: ! y *not*;
- **operadores de comparación**: *in*, =, <, >, =<, >=.

Pueden notar que por la precedencia del box join, la expresión a.b[c] se traduce a (a.b)[c].

Los operadores lógicos tiene menos precedencia, respetando:

- **operadores de negación**: ! y *not*;
- **conjunción**: && y *and*;
- **implicación**: =>, *implies*, y *else*;
- **bi-implicación**: <=>, *iff*;
- **disyunción**:|| y *or*;
- **let y operadores de cuantificación**: *let*, *no*, *some*, *lone*, *one* y *sum*.

Todos los operadores binarios se asocian hacia la izquierda, con excepción de la implicación, que se asocia a derecha. Así, por ejemplo, $p \Rightarrow q \Rightarrow r$ se traduce como $p \Rightarrow (q \Rightarrow r)$, y $a.b.c$ se traduce como $(a.b).c$

En una implicación con cláusula *else*, esta se asocia a la implicación más cercana. De esta manera, una restricción como $p \Rightarrow q \Rightarrow r$ **else** s se interpreta como $p \Rightarrow (q \Rightarrow r$ **else** $s)$.