Ingeniería del Software II
8 - El lenguaje de especificación Alloy

El lenguaje de especificación Alloy

- Alloy es un lenguaje liviano (lightweight) de modelado para la especificación y diseño de software.
- Puede analizarse de manera automática con el Alloy Analyzer.
- Provee un visualizador que ayuda a comprender las soluciones y los contraejemplos que el analizador encuentre.
- Alloy = lógica de primer orden + álgebra de relaciones.
  - el lenguaje subyacente de Alloy es la lógica relacional.
- A diferencia del álgebra de relaciones, Alloy posee relaciones de cualquier aridad.

Sintaxis

Signaturas

- Las signaturas son la base del lenguaje Alloy.
- Permiten denotar dominios.
- Ejemplo: Especificación de un sistema de memorias.
  - Primero necesitamos dos dominios:
    1. direcciones (de memoria);
    2. datos (a almacenarse en direcciones de memoria).
  - En Alloy:

```
sig Addr {}
sig Data {}
```

Sintaxis

Signaturas

- Las signaturas son la base del lenguaje Alloy.
- Permiten denotar dominios.
- Ejemplo: Especificación de un sistema de memorias.
  - Primero necesitamos dos dominios:
    1. direcciones (de memoria);
    2. datos (a almacenarse en direcciones de memoria).
  - En Alloy:

Alternativamente

```
sig Addr, Data {}
```

Sintaxis

Signaturas

- Las signaturas son la base del lenguaje Alloy.
- Permiten denotar dominios.
- Ejemplo: Especificación de un sistema d
  - Primero necesitamos dos domi
    1. direcciones (de memoria);
    2. datos (a almacenarse en d).
  - En Alloy:

Estas signaturas son básicas, ya que no tienen estructura interna.

Alternativamente

```
sig Addr, Data {}
```

Sintaxis

Signaturas Complejas

- Las signaturas pueden tener estructuras complejas.
- Pueden incluir "**campos**":

```
sig Memory {
    addrs: set Addr
    map: Addr -> Data
}
```

- El ejemplo describe una memoria que
  1. posee un espacio de direcciones (**addrs**); y
  2. cada dirección de memoria puede albergar un dato (**map**).

Sintaxis

Signaturas Complejas

- Las signaturas pueden tener
- Pueden incluir "**campos**":

```
sig Memory {
    addrs: set Addr
    map: Addr -> Data
}
```

"**Addr -> Data**" denota una relación binaria. Es lo que usualmente denotamos con **Addr×Data**.

- El ejemplo describe una memoria que
  1. posee un espacio de direcciones (**addrs**); y
  2. cada dirección de memoria puede albergar un dato (**map**).

Sintaxis

Signaturas Complejas

- Las signaturas pueden tener
- Pueden incluir "**campos**":

```
sig Memory {
    addrs: set Addr
    map: Addr -> Data
}
```

¡¡Pero entonces puede ser inconsistente!!

- El ejemplo describe una memoria que
  1. posee un espacio de direcciones (**addrs**); y
  2. cada dirección de memoria puede albergar un dato (**map**).

Sintaxis

Signaturas Complejas

- Las signaturas pueden tener
- Pueden incluir "**campos**":

```
sig Memory {
    addrs: set Addr
    map: Addr -> Data
}
```

- El ejemplo describe un

1. posee un espacio
2. cada dirección de
   (**map**).

"A denota
que

¡¡Pero entonces puede ser inconsistente!!

Nada asegura que **map** sea una función, ni que su dominio sea exactamente las direcciones que la memoria puede direccionar (i.e. que **map** sea total en **addrs**).

Sintaxis

Signaturas Complejas

- Las signaturas pueden tener estructuras complejas.
- Pueden incluir "campos" y **propiedades**:

```
sig Memory {
     addrs: set Addr
     map: Addr -> Data
    }{
     ~map.map in iden
     map.Data = addrs
    }
```

Sintaxis

Signaturas Complejas

- Las signaturas pueden tener estructuras complejas.
- Pueden incluir "campos" y **propiedades**:

```
sig Memory {
    addrs: set Addr
    map: Addr -> Data
}{
    ~map.map in iden
    map.Data = addrs
}
```

`map` es una función
(lo vimos la clase pasada)

`map` es total.

Sintaxis

Signaturas Complejas

- Las signaturas pueden tener estructuras complejas.
- Pueden incluir "campos" y **propiedades**:

```
sig Memory {
    addrs: set Addr
    map: Addr -> Data
}{
    ~map.map in iden
    map.Data = addrs
}
```

`map` es una función
(lo vimos la clase pasada)

`map` es total.

Notar el uso del 'join' (.)
entre una relación binaria y un
conjunto ("relación unaria")

Sintaxis

Signaturas Complejas

- Las signaturas pueden te
- Pueden incluir "campos" y

¡Pero esto lo podemos escribir más facil!

```
sig Memory {
    addrs: set Addr
    map: Addr -> Data
}{
    ~map.map in iden
    map.Data = addrs
}
```

`map` es una función (lo vimos la clase pasada)

`map` es total.

Notar el uso del 'join' (.) entre una relación binaria y un conjunto ("relación unaria")

Sintaxis

Signaturas Complejas

- Las signaturas pueden tener estructuras complejas.
- Pueden incluir "campos" y propiedades.
- Alloy provee un lenguaje rico para definir multiplicidades de conjuntos y relaciones:

```
sig Memory {
    addrs: set Addr
    map: addrs -> one Data
  }
```

Sintaxis

Extensión de signaturas

- Pueden definirse tipos más expecíficos de signaturas extendiendo otras signaturas existentes
- Ejemplo: un sistema de memoria de una computadora consiste de la memoria principal (`MainMemory`) y la memoria cache (`Cache`).

```
sig MainMemory extends Memory { }

sig Cache extends Memory {
    dirty: set addrs
  }
```

La memoria cache, además, debe llevar un registro de los datos que fueron modificados pero aún no copiados en la memoria principal.

Sintaxis

Completando el sistema del ejemplo

- El sistema completo podría quedar definido como:

```
sig System {
    cache: Cache
    main: MainMemory
   }
```

- La intención de esta signatura es describir un sistema (de memoria) como una estructura compuesta por una memoria principal (campo de tipo **`MainMemory`**) y una cache (campo de tipo **`Cache`**).

Sintaxis

Operaciones en un modelo Alloy

- Las signaturas permiten describir los dominios y sus estructuras.
- Además se pueden especificar predicados sobre estos dominios.
- Los predicados permiten, en particular, definir operaciones que relacionan el estado previo a la aplicación de la operación con el estado posterior correspondiente.

```
pred Write [m,m': Memory, d: Data, a: Addr] {
    m'.map = m.map ++ (a -> d)
   }
```

- `Write` modifica una memoria sobreescribiendo el valor asociado a una dirección `a` con el dato `d`.

Sintaxis

Operaciones en un modelo Alloy

- Las signaturas permiten describir
- pecifica
- niten, en parti
- el estado previo a la icación de la operación con el estado posterior correspondiente.

Este predicado no está directamente asociado a ninguna signatura.

Las variables "primadas" corresponden a estados posteriores a la ejecución de la operación (es una convención).

```
pred Write [m,m': Memory, d: Data, a: Addr] {
    m'.map = m.map ++ (a -> d)
  }
```

- `Write` modifica una memoria sobreescribiendo el valor asociado a una dirección `a` con el dato `d`.

Sintaxis

Hechos (facts)

- Las especificaciones de la estructura de un sistema que se pueden lograr usando sólo signaturas son limitadas.
- Usualmente, las signaturas deben complementarse con restricciones estructurales adicionales que permitan describir propiedades o restricciones generales.
- Por ejemplo, podríamos agregar una restricción que asegure que la memoria cache no direcciona fuera de la memoria principal:

```
fact {
    all s:System | s.cache.addrs in s.main.addrs
   }
```

Sintaxis

Hechos (facts) (cont.)

Escribir:

```
sig Memory {
    addrs: set Addr
    map: Addr -> Data
   }{
    ~map.map in iden
    map.Data = addrs
   }
```

Es lo mismo que escribir:

```
sig Memory {
    addrs: set Addr
    map: Addr -> Data
   }

fact {
    all m:Memory | ~(m.map).(m.map) in iden
    all m:Memory | m.map.Data = m.addrs
   }
```

Sintaxis

Hechos (facts) (cont.)

Escribir:

```
sig Memory {
    addrs: set Addr
    map: Addr -> Data
}{
    ~map.map in iden
    map.Data = addrs
}
```

Es lo mismo que escribir:

```
sig Memory {
    addrs: set Addr
    map: Addr -> Data
}

fact {
    all m:Memory | ~(m.map).(m.map) in iden
    all m:Memory | m.map.Data = m.addrs
}
```

...pero esto es más complicado!

Sintaxis

Un `fact` de signatura es, de hecho, un invariante de la signatura.

Hechos (facts) (cont.)

Escribir:

```
sig Memory {
    addrs: set Addr
    map: Addr -> Data
}{
    ~map.map in iden
    map.Data = addrs
}
```

Es lo mismo que escribir:

```
sig Memory {
    addrs: set Addr
    map: Addr -> Data
}

fact {
    all m:Memory | ~(m.map).(m.map) in iden
    all m:Memory | m.map.Data = m.addrs
}
```

...pero esto es más complicado!

Sintaxis

Aserciones

- Todos los elementos de Alloy descritos hasta el momento forman parte de la descripción de los modelos.
- Las aserciones, en cambio, son enunciados que queremos comprobar si son o no propiedades del sistema especificado.
- Por ejemplo, podríamos querer saber si en el sistema de memorias se cumple la siguiente propiedad :

```
assert {
    all s: System |
        all a: s.cache.addrs-s.cache.dirty |
            s.cache.map[a] = s.main.map[a]
}
```

¿Qué quiere decir?

Sintaxis

Hechos, Predicados y Aserciones

- Un hecho define una restricción que es siempre verdadera.
- Por lo tanto, un hecho limita el dominio de la especificación.
- Un predicado sólo será evaluado cuando se invoque.
- Un predicado puede invocarse desde un hecho, una aserción, una función u otro predicado.
- Una aserción es una restricción que se espera que se deduzca de los hechos del modelo.
- Por lo tanto, Alloy Analyzer verifica la validez de una aserción en un modelo.
- Si una aserción no es válida, o bien ésta expuso un error del modelo, o bien la aserción no estuvo bien formulada.

Características de Alloy

- El lenguaje es mínimo, no incluye ni siquiera tipos básicos.
- El lenguaje es relacional, con una sintaxis simple y elegante basada en operadores relacionales tales como composición, unión, complemento, etc.
- El lenguaje es expresivo, permitiendo cuantificación similar a la cuantificación sobre individuos, pero con una semántica relacional, y clausura transitiva.
- El lenguaje es fácil de usar, con construcciones que recuerdan elementos de orientación a objetos, tales como extensión de signaturas y la notación de punto para acceso a servicios (pero el lenguaje no es orientado a objetos).

Características de

En realidad, tiene `Int` "built in" y bibliotecas (ej: orden total)

- El lenguaje es mínimo, no incluye ni siquiera tipos básicos.
- El lenguaje es relacional, con una sintaxis simple y elegante basada en operadores relacionales tales como composición, unión, complemento, etc.
- El lenguaje es expresivo, permitiendo cuantificación similar a la cuantificación sobre individuos, pero con una semántica relacional, y clausura transitiva.
- El lenguaje es fácil de usar, con construcciones que recuerdan elementos de orientación a objetos, tales como extensión de signaturas y la notación de punto para acceso a servicios (pero el lenguaje no es orientado a objetos).

Características de Alloy

- el lenguaje hace que las especificaciones sean analizables automáticamente, usando técnicas de SAT solving (para validación, pues el lenguaje es de primer orden)
- Alloy trabaja bajo la "small scope hypothesis" (hipótesis de entorno pequeño):

> La mayoría de los errores tienen contraejemplos pequeños.

- i.e. los sistemas que fallan en una gran instancia, casi seguro falla también en una instancia pequeña con propiedades similares, aún si las pequeñas instancias no ocurren en la práctica.

Alloy como Constraint Solver

- Alloy es una herramienta versátil, que puede utilizarse con diferentes finalidades.
- Ya hemos utilizado Alloy para realizar una especificación y verificar propiedades en ella.
- Alloy también puede utilizarse como constraint solver, i.e., para la resolución de problemas con restricciones.

Alloy como Constraint Solver

Ejemplo: El problema del granjero que debe cruzar un río

- Un granjero se encuentra a orillas de un río llevando un zorro, una gallina y una bolsa de maíz.
- Necesita cruzar el río con el zorro, la gallina y la bolsa de maíz, pero sólo cuenta con un pequeño bote en el que puede llevar a lo sumo una carga. (Por supuesto que el único que sabe remar es el granjero.)
- El granjero no debe dejar solo al zorro con la gallina porque se la puede comer. Por las mismas razones, tampoco puede dejar sola a la gallina junto al maíz.
- ¿Cómo puede hacer para cruzar toda su carga sin que alguno de los animales se haga un festín?
- Podemos utilizar Alloy y el Alloy Analyzer para resolver este problema.

Alloy como Constraint Solver

- Primero modelamos los "objetos" que son el dominio del problema mediante signaturas.

```
abstract sig Object {
    eats: set Object
   }
one sig Farmer, Fox, Chicken, Grain
    extends Object {}
```

Allo r

- Primero m dominio del problema m

La signatura `Object` es "abstracta". esto significa que no contendrá elementos a menos que los declaremos explícitamente mediante extensiones.

```
abstract sig Object {
    eats: set Object
   }
one sig Farmer, Fox, Chicken, Grain
    extends Object {}
```

La multiplicidad `one` garantiza que cada una de estas signaturas contendrá exactamente un elemento.

Alloy como Constraint Solver

- La especificación del dominio del problema necesita complementarse con un hecho que explicite quién puede comer qué.

```
fact eating {
     eats = Fox -> Chicken + Chicken -> Grain
    }
```

Alloy como Co

- La especificación del domi
  complementarse con un hec
  comer qué.

Observar que el "campo" **eats** de la signatura **Object** es en realidad una relación **Object -> Object** encubierta

```
fact eating {
     eats = Fox -> Chicken + Chicken -> Grain
   }
```

Alloy como Co

Observar que el "campo" **eats** de la signatura **Object** es en realidad una relación **Object -> Object** encubierta

- La especificación del domi… complementarse con un hec… de comer qué.

```
fact eating {
     eats = Fox -> Chicken + Chicken -> Grain
   }
```

- Esta es una forma simple de expresarlo, pero no la única. Otro forma:

```
fact eating {
     Fox.eats = Chicken
     Chicken.eats = Grain
     no Grain.eats
     no Farmer.eats
   }
```

Alloy como Constraint Solver

Cambios de estado

- Los posibles traslados del bote lo podemos modelar como cambios de estado.
- Para ello necesitamos indicar cómo está compuesto el estado.
- En nuestro caso debemos indicar qué objetos hay en cada lado del río

```
sig State {
    near: set Object,
    far: set Object,
   }
```

Alloy como Constraint Solver

Cambios de estado (cont.)

- Para poder identificar la secuencia de los movimientos vamos a necestiar introducir "trazas".
- Una traza es una secuencia ordenada de estados.
- Alloy provee una definición de orden en un módulo predefinido

```
open util/ordering[State]
```

- Podemos caracterizar el estado inicial de la siguiente manera:

```
fact initialState {
    let s0 = first[] |
        s0.near = Object && no s0.far
   }
```

Alloy como Constraint Solver

Cambios de estado (cont.)

- Para poder identificar la secuencia de los movimientos vamos a necestiar introducir "trazas".
- Una traza es una secuencia orden
- Alloy provee una definición de o predefinido

Notar que el módulo **ordering** es paramétrico.

```
open util/ordering[State]
```

- Podemos caracterizar el estado inicial de la siguiente manera:

```
fact initialState {
    let s0 = first[] |
        s0.near = Object && no s0.far
  }
```

Alloy como Constraint Solver

Cambios de estado (cont.)

- Para poder identificar la secuencia de los movimientos vamos a necestiar introducir "trazas".
- Una traza es una secuencia orden
- Alloy provee una definición de o predefinido

```
open util/ordering[State]
```

- Podemos caracterizar el esta

```
fact initialState {
    let s0 = first[] |
        s0.near = Object && no s0.far
  }
```

Notar que el módulo **ordering** es paramétrico.

first es una función del módulo **ordering**. También lo son **next** y **last**.

Alloy como Constraint Solver

Cambios de estado (cont.)

- Debemos especificar ahora como se modifica el estado en un paso, i.e., la transición.
- Para ello utilizaremos un predicado auxiliar que especifica el cruce del río.

```
pred crossRiver [from,from',to,to': set Object ] {
   ( from' = from - Farmer &&
     to' = to - to.eats + Farmer )
   ||
   ( some item: from - Farmer |
       from' = from - Farmer - item &&
       to' = to - to.eats + Farmer + item
    )
   }
```

Alloy como Constraint Solver

Cambios de estado (cont.)

- Usando `crossRiver` podemos definir las transiciones del sistema como sigue.

```
fact stateTransition {
  all s: State, s': next[s] |
    ( Farmer in s.near =>
        crossRiver[s.near,s'.near,s.far,s'.far] )
    &&
    ( Farmer in s.far =>
        crossRiver[s.far,s'.far,s.near,s'.near ] )
  }
```

Alloy como Constraint Solver

Cambios de estado (cont.)

- Usando `crossRiver` poder
  del sistema como sigue.

Notar que este `fact` está definiendo la relación de orden a través de la función `next`.

```
fact stateTransition {
  all s: State, s': next[s] |
    ( Farmer in s.near =>
        crossRiver[s.near,s'.near,s.far,s'.far] )
    &&
    ( Farmer in s.far =>
        crossRiver[s.far,s'.far,s.near,s'.near ] )
  }
```

Alloy como Constraint Solver

Cambios de estado (cont.)

- Usando un predicado que especifique cual es el estado final deseado, podemos utilizar Alloy para que busque la solución:

```
pred solvePuzzle[] {
  last[].far = Object
}

run solvePuzzle for 8 State
```

Alloy como Constraint Solver

Cambios de estado (cont.)

- Usando un predicado que especif… …do final deseado, podemos utiliz… la solución:

Notar el uso de `last` para identificar cómo debería ser el estado final.

```
pred solvePuzzle[] {
  last[].far = Object
}

run solvePuzzle for 8 State
```

Modelos de Ejecuciones en Alloy

- Alloy es un lenguaje adecuado para la especificación de aspectos estáticos de software.
- Sin embargo, presenta una alternativa para describir (o analizar) aspectos dinámicos del mismo, es decir, propiedades de ejecuciones.
- Aún así, la idea presentada anteriormente sobre cómo modelar trazas puede aprovecharse para modelar y analizar propiedades dinámicas de sistemas.

Modelos de Ejecuciones en Alloy

La técnica para modelar ejecuciones y sus propiedades en Alloy consta de los siguientes pasos:

1. identificar la parte de la especificación que constituye el estado del sistema (usualmente será un conjunto de signaturas).
2. utilizar esto para definir la signatura del estado.
3. definir una traza como un orden total sobre estados.
4. indicar cuáles son las propiedades que se asumen del estado inicial con un hecho que predique sobre el primer elemento de las trazas.
5. indicar cómo se pasa de un estado a otro mediante hechos, que relacionen pares consecutivos en las trazas (define una transición)
6. usar aserciones para especificar las propiedades requeridas de las ejecuciones (como propiedades de los estados involucrados en las trazas).

Modelos de Ejecuciones en Alloy

Ejecuciones de Operaciones de Memorias

Consideremos un modelo simple de memorias:

```
sig Memory{
    data: Addr -> lone Data
   }

pred init [ m: Memory ] { no m.data }

pred write [ m,m': Memory, a: Addr, d: Data ] {
    m'.data = m.data ++ a -> d
   }

pred read [ m: Memory, a: Addr, d: Data ] {
    let d' = m.data[a] | some d' implies d = d'
}
```

Modelos de Ejecuciones en Alloy

Ejecuciones de Operaciones de Memorias (cont.)

Obviamente, Memory representa nuestro estado, así que la definición de trazas será la siguiente:

```
open util/ordering [ State ]

sig State {
    mem:Memory
  }

fact Traces {
    init[first[].mem]
    all s:State, s':next[s] |
      (some a:Addr, d:Data | read[s.mem,a,d] and s = s')
      or
      (some a:Addr, d:Data | write[s.mem,s'.mem,a,d])
  }
```

Modelos de Ejecuciones en Alloy

Ejecuciones de Operaciones de Memorias (cont.)

Finalmente podemos expresar propiedades usando aserciones o predicados, de acuerdo a lo que busquemos:

```
pred freshDir [ m: Memory ] {
     some a: Addr | no (m.data.Data & a)
    }

pred freshDirInLast {
     freshDir [last[].mem]
    }

run freshDirInLast for 5
```

Uso de Alloy para verificar refinamientos

- En la práctica, muchas veces se trabaja con diferentes modelos de un mismo problema (o sistema).
- Estos difieren en el grado de abstracción empleado:
  - los más abstractos corresponden a la especificación,
  - los más concretos agregan más detalles de implementación
- Por ejemplo, el modelo de memorias con cache, puede verse como una versión más detallada, más concreta, más compleja, del modelo simple de memorias de las transparencias anteriores.
- Es decir, las memorias con cache son un refinamiento de las memorias simples.
- Sería interesante y útil poder comparar los diferentes modelos.
- Esto permitiría, por ejemplo, garantizar que las operaciones sobre las memorias más complejas se corresponden (i.e., respetan la especificación) de las memorias más abstractas.

Uso de Alloy para verificar refinamientos

- Tal comparción de modelos no puede hacerse directamente, pues estos definen espacios de estados diferentes.
- Una forma de relacionar modelos en diferentes niveles de abstracción es mediante funciones de abstracción.
- Una función de abstracción es una función que relaciona elementos de un modelo concreto con elementos de uno abstracto, mapeando cada estado concreto al estado abstracto que éste representa.
- En el ejemplo de los modelos de memoria cache y memoria abstracta, el map de memorias que una memoria con cache representa es la memoria principal "pisada" con el contenido de la cache:

```
fun alpha [c: CacheSystem]: Memory {
    { m: Memory | m.data = c.main.map ++ c.cache.map }
  }
```

Uso de Alloy para verificar refinamientos

Con esta definición Alloy de la función de abstracción podemos:

- comprobar que no hay "pérdida" en la representación (la función de abstracción es suryectiva)
- comprobar que las operaciones concretas "implementan" las operaciones abstractas correspondientes:

```
assert WriteOK {
    all c, c': CacheSystem,
        a: Addr, d: Data, m, m': Memory |
           ( cache/write[c,c',a,d] and
             m = alpha[c] and m' = alpha[c'] )
           =>
              amemory/write[m,m',a,d]
   }
```

Análisis de Especificaciones en Alloy

El análisis nos ayuda a construir especificaciones de 3 maneras distintas:

- Fomenta la exploración a través de ejemplos concretos que confirman nuestra intuición y sugieren nuevos escenarios.
- Supervisa, ayudándonos a verificar sobre las versiones preliminares que lo que escribamos signifique lo que pretendemos significar.
- Revela defectos de manera temprana que de otra manera se hubieran revelado mucho más tarde (inclusive, ya entregado y en funcionamiento)

Análisis de Especificaciones en Alloy

Hemos visto que las especificaciones Alloy son analizables automáticamente usando técnicas de SAT solving.

Esencialmente, el uso de la técnica requiere:

- la caracterización de la propiedad a validar usando predicados o aserciones;
- la provisión de cotas adecuadas para los dominios de la especificación; y
- la interpretación de los resultados para la depuración de las especificaciones.

Análisis de Especificaciones en Alloy

Análisis de restricciones con Alloy Analyzer

- Para el caso de análisis de predicados:
  - La restricción se construye como la conjución del predicado bajo estudio con todos los hechos del modelo (los explícitos y los implícitos en las declaraciones).
  - Si es satisfactible se genera un ejemplo, i.e., una instancia (o modelo) que satisface el predicado.
- Para el caso de análisis de aserciones:
  - La restricción se construye como la conjución de la negación de la aserción bajo estudio con todos lo hechos del modelo.
  - Si no es satisfactible se genera un contraejemplo, i.e., una instancia (o modelo) que hace que la aserción sea falsa.

Análisis de Especificaciones en Alloy

Cotas adecuadas en la búsqueda de instancias

- El éxito en el uso del Alloy Analyzer para validar propiedades de especificaciones depende en gran medida de la elección de cotas adecuadas para la búsqueda de instancias de aserciones o predicados.
- Al elegir cotas debemos ser cuidadosos:

  cotas incorrectas pueden llevarnos a conclusiones erróneas.

Análisis de Especificaciones en Alloy

Sugerencias para la elección de cotas

- Asegurarse de que la cota de cada signatura es lo suficientemente grande como para permitir la existencia de constantes definidas en el modelo.

  Por ejemplo:

```
some disj start, end: Node | ...
```

  requiere como mínimo **dos** elementos en **Node**.

Análisis de Especificaciones en Alloy

Sugerencias para la elección de cotas (cont.)

- Si los elementos relevantes de una signatura se declaran como variables, no hace falta considerar cotas superiores al número de variables usadas.

  Por ejemplo:

```
sig Name, Address {}
sig Alias, Group extends Name {}
sig Book {
    addr: Name -> Address
   }
pred add [ b,b': Book, n: Name, a: Address ] {
    b'.addr = b.addr + n->a
   }
run add for 3 but 2 Book
```

Análisis de Especificaciones en Alloy

Sugerencias para la elección de cotas (cont.)

- Si los elementos relevantes de una signatura se declaran como variables, no hace falta considerar cotas superiores al número de variables usadas.

  Por ejemplo:

```
sig Name, Address {}
sig Alias, Group extends Name {}
sig Book {
    addr: Name -> Address
   }
pred add [ b,b': Book, n: Name, a: Address ] {
    b'.addr = b.addr + n->a
   }
run add for 3 but 2 Book
```

Sólo se necesitan 2 **Books** para visualizar el predicado **add**

Análisis de Especificaciones en Alloy

Sugerencias para la elección de cotas (cont.)

- Por otro lado, el análisis del siguiente assert

```
sig Node {
    adj: Node
   }
assert allConectedGraphsAreClique {
    all n, n': Node |
        n' in n.*adj => (n' in n.adj or n' = n )
   }
check allConectedGraphsAreClique for 3
```

no debería limitarse a 2 **Nodes** porque las variables **n** y **n'** están cuantificadas universalmente sobre todos los **Nodes**

Análisis de Especificaciones en Alloy

Sugerencias para la elección de cotas (cont.)

- Por otro lado, el análisis del siguiente assert

```
sig Node {
     adj: Node
    }
assert allConectedGraphsAreClique {
     all n, n': Node |
          n' in n.*adj => (n' in n.adj or n' = n )
    }
check allConectedGraphsAreClique for 3
```

Observar que el assert es válido si la cota es 2

no debería limitarse a 2 **Nodes** porque las variables **n** y **n'** están cuantificadas universalmente sobre todos los **Nodes**

Análisis de Especificaciones en Alloy

Cuantificadores universales no acotados

- Un problema que puede dar lugar a contraejemplos no intuitivos es el que surge al querer representar, mediante una signatura, todos los valores posibles de una estructura compuesta.
- Esto va, en un sentido, en contra de la semántica de Alloy, que interpreta a las signaturas sólo como un conjunto de elementos.
- Esta contradicción sólo se manifiesta cuando se utiliza cuantificación universal de cierta forma (concretamente, de manera no acotada).

Análisis de Especificaciones en Alloy

Cuantificadores universales no acotados (cont.)

Consideremos el siguiente ejemplo:

```
sig Element {}

sig Set {
    elements: set Element
   }

assert Closed {
    all s0, s1: Set | some s2: Set |
        s2.elements = s0.elements + s1.elements
   }

check Closed
```

Análisis de Especificaciones en Alloy

Cuantificadores universales no acotados (cont.)

Esta aserción nos puede dar contraejemplos tales como el siguiente:

```
Set = { (S0), (S1) }
Element = { (E0), (E1) }
s0 = { (S0) }
s1 ={ (S1) }
elements = { (S0, E0), (S1, E1) }
```

¿Dónde está el problema?

Análisis de Especificaciones en Alloy

Cuantificadores universales no acotados (cont.)

El analizador encontró un contraejemplo que no "completó" la signatura `Set` con valores suficientes: no existen instancias de `Set` con el conjunto vacío, ni con los elementos `E0` y `E1`.

Dado que queremos que haya tantas instancias de Set como subconjuntos de Element, podemos forzar esto mediante axiomas de generación.

```
fact SetGenerator {
    some s: Set | no s.elements
    all s: Set, e: Element |
       some s': Set | s'.elements = s.elements + e
  }
```

Análisis de Especificaciones en Alloy

Cuantificadores universales no acotados (cont.)

El analizador encontró un contraejem[illegible] la signatura `Set` con valores su[illegible] `Set` con el conjunto vacío

Dado que queremos que ha[illegible] subconjuntos de Element, n[illegible] de generación.

¿Qué sucederá si analizamos la aserción anterior, teniendo en cuenta el axioma de generación?

El resultado es el esperado:
"No counterexample found. Assertion may be valid."

```
fact SetGenerator {
    some s: Set | no s.elements
    all s: Set, e: Element |
       some s': Set | s'.elements = s.elements + e
  }
```

Análisis de Especificaciones en Alloy

Cuantificadores universales no acotados (cont.)

Notar: las interpretaciones que el AlloyAnalyzer explorará son **finitas**

Por lo tanto, si el axioma de generación fuerza infinitos átomos, la especificación asociada se hará inconsistente en el análisis.

```
abstract sig List {}
one sig EmptyList extends List {}
sig NonEmptyList extends List {
    element: Element,
    rest: List
  }
fact ListGenerator {
    all l: List, e: Element |
        some l': List | l'.rest = l  and  l'.element = e
  }
```

Análisis de Especificaciones en Alloy

Cuantificadores universales no acotados (cont.)

Sobre el ejemplo anterior:

```
assert ObviouslyNotValid {
    all l: List |
        all l': List | l'.rest = l or l = l'
   }
check ObviouslyNotValid
```

??!!

Alloy Analyzer no encuentra contraejemplo para esta aserción

Es consecuencia del axioma de generación forzando infinitas listas

Análisis de Especificaciones en Alloy

Cuantificadores universales no acotados (cont.)

Sobre el ejemplo anterior:

¿Cuántas instancias de lista se muestra al correr lo siguiente?

```
pred show [] {}
run show
```

¿Y en el siguiente caso?

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

Análisis de Especificaciones en Alloy

Cuantificadores universales no acotados (cont.)

Sobre el ejemplo anterior:

¿Cuántas instancias de lista se muestra al correr lo siguiente?

```
pred show [] {}
run show
```

1

¿Por qué?

¿Y en el siguiente caso?

0

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

Análisis de Especificaciones en Alloy

Cuantificadores universales no acotados (cont.)

- Cabe aclarar que la necesidad del axioma de generación surge como consecuencia de la cuantificación universal no acotada.
- Existen ciertas formas en las aserciones para las cuales está garantizado que no es necesario introducir axiomas de generación.
- Éstas son las expresiones con cuantificación universal acotada.
- Afortunadamente, son las más comunes en la práctica.