Métodos Formales para Ingeniería de Software

Ma. Laura Cobo

Modelado de dinámica
Módulos en Alloy

Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur
Argentina

¿Qué podemos esperar de las operaciones?

Dado que los átomos son estáticos ….

¿Cómo expresamos una transición?

Puede modelarse a través de un predicado que establezca una relación entre dos estados

- El estado **anterior** a la transición y
- El estado **siguiente**

Requiere las restricciones necesarias: pre y post condiciones para cada transición, condiciones de marco “frame”

Solución: "nuevo" patrón

Trata a las acciones y operaciones en un estado global, que define el comportamiento de una máquina abstracta

```
sig State { … }

pred  init [s: State] { … }
// describe el estado inicial
pred  inv [s: State] { … }
// describe los invariantes que todo
estado debe verificar
pred  op1 [s, s1: State] { … }
…
…
pred  opN [s, s1: State] { … }
```

Máquina abstracta

Puede chequearse que las operaciones preservan invariantes

```
assert initVerifies { all s: State |
   init[s] => inv[s] }

// para cada operación

assert  opPreserves {
   all s,s1: State |
        inv[s] && op[s,s1] => inv[s1]
}
```

Ejemplo

Modelo

```
sig  Biblioteca { coleccion: set Libro}
sig  Libro { escritoPor: set Autor}
sig  Autor {}
```

Agregando la signatura que modela el estado

Modelo

```
sig  Biblioteca { states: set State}
sig  Libro { escritoPor: set Autor}
sig  Autor {}

sig  State { coleccion: set Libro}
```

Ejemplo

Modelo

```
sig  Biblioteca { states: set State}
sig  Libro { escritoPor: set Autor}
sig  Autor {}

sig  State { coleccion: set Libro}
```

Estado inicial

```
pred init [s: State]
  #s.coleccion=0
```

Ejemplo

Modelo

```
sig  Biblioteca { states: set State}
sig  Libro { escritoPor: set Autor}
sig  Autor {}

sig  State { coleccion: set Libro}
```

Ejemplo de operación sobre los estados

```
pred agregarAColeccion [s,s1: State, l:Libro]
  s1.coleccion = s.coleccion + l
```

Solución: "nuevo" patrón

Utilizar un patrón de trazas:

- Modela secuencias de ejecuciones sobre la máquina abstracta
- Crea un ordenamiento total sobre los estados
- Conecta estados sucesivos a través de operaciones
  - ✓ Todos los estados deben ser alcanzables

Para garantizar esto la signatura estado, debe garantizar ciertas características. Para facilitar el modelado Alloy utiliza módulos predefinidos.

Módulos en Alloy

- Alloy cuenta con un sistema de módulos que permiten la modularización y reuso de modelos.
- Un módulo define un modelo que puede ser incorporado como **submodelo** de otro
- Para facilitar el reuso, los módulos pueden ser ***paramétricos*** para **una o más signaturas**

Módulos en Alloy

Un ejemplo de módulo podría ser el siguiente:

```
module util/relation

-- r es una relación acíclica sobre el
conjunto S

pred acyclic[r: univ-> univ, S: set univ]
{all x:S | x !in x.^r}
```

Módulos en Alloy

```
module util/relation

-- r es una relación acíclica sobre el conjunto S

pred acyclic[r: univ-> univ, S: set univ] {all x:S |
    x !in x.^r}
```

O para asegurarse que entre las subcarpetas no este la carpeta que las contiene

```
module fileSystem
open util/relation as rel
sig Object {}
sig Folder extends Object{ subFolders: set
Folder}
fact {acyclic[subFolders,Folder]}
```

Declaración de módulos

```
module modulePathName
```

Es el encabezado o header del módulo

Declaración de módulos

```
module modulePathName
```

Es el encabezado o header del módulo

Un módulo puede **importar** otro módulo a través de la sentencia open

```
open modulePathName
```

La sentencia open puede interpretarse como inclusión textual

Declaración de módulos

```
module modulePathName
```

Es el encabezado o header del módulo

Un módulo puede **importar** otro módulo a través de la sentencia open

```
open modulePathName
```

Un módulo puede importar a otro que a su vez importa un tercer módulo, y así siguiendo
No se permiten ciclos en la estructura de importación

Definición

Todo módulo tiene un path que debe hacer match con el archivo correspondiente en el file system del sistema operativo

El path puede:

- Ser sólo el nombre de un archivo (sin la extensión als)
- Ser el path completo desde la raíz

La raíz del path en el encabezado del módulo a importar es la raíz para cada importación

Definición

```
module C/F/mod
    open D/lib1
    open C/E/H/lib2
    open C/E/G/lib3
```

El *path name* del módulo en el encabezamiento especifica el directorio raíz desde el que se importa cada archivo

¿ de acuerdo a lo especificado cómo podría ser la estructura de directorios que contiene los módulos importados?

Definición

Ejemplo

```
module biblioteca
   open lib/libro
```

Si el path de *bibilioteca.als es <p>* en el file system, entonces el analizador de Alloy buscará *libro.als* en *<p>/lib/*

Definición

Ejemplo

```
module biblioteca
   open lib/libro
```

Si el path de *bibilioteca.als es <p>* en el file system, entonces el analizador de Alloy buscará *libro.als* en *<p>/lib/*

Alloy cuenta con una librería de módulos predefinidos ¿qué acciones toma el analizador en caso de querer importar estos módulos?

Definición

Ejemplo

```
module biblioteca
   open lib/libro
```

Si el path de *bibilioteca.als es <p>* en el file system, entonces el analizador de Alloy buscará *libro.als* en *<p>/lib/*

Alloy cuenta con una librería de módulos predefinidos ¿qué acciones toma el analizador en caso de querer importar estos módulos?

Cualquier módulo importado será ***buscado primero*** *entre los predefinidos.*
Si esta búsqueda falla se aplica la búsqueda descripta previamente

Renombrado

El módulo, **debe** contar con un nombre corto cuando:

1. el path de importación incluye **/** (es decir: es un path, no sólo un nombre) y
2. se importa más de un módulo con algún predicado/función de igual parte pública

Esto se logra con la keyword **as**

```
open util/relation as rel
```

Colisión de nombres

Los módulos definen sus propios espacios de nombres

La colisión de nombres se evita utilizando **nombres calificados**.

```
module fileSystem
open util/relation as rel
sig Object {}
sig Folder extends Object{ subFolders: set
Folder}

fact {rel/acyclic[subFolders,Folder]}
```

Módulos Parametrizados

- Un modelo **m** puede **parametrizarse** mediante uno o más parámetros signatura, $[x_1, \ldots, x_n]$
- Cada importación debe instanciar cada parámetro con un nombre de signatura.
- El efecto de abrir $\mathbf{m}[S_1, \ldots, S_n]$ es que se obtiene una copia de **m** con cada parámetro signatura x_i reemplazado por la signatura S_i

Módulos Parametrizados: Ejemplo

```
module graph [Node] // 1 signature param
    open  util/relation as  rel
    pred  dag [r:  Node  ->  Node ] {
          acyclic [r,  Node ]
    }
```

```
module fileSystem
  open util/graph [Object] as g
  sig Object { }
  sig Folder extends Object {
          subFolders: set Folder
  }
  fact {g/dag [subFolders]}
```

El módulo predefinido "Ordering"

- Crea un ordenamiento lineal simple sobre los átomos en la signatura **S**

```
module util/ordering[S]
```

- También restringe a todos los átomos, permitidos por el scope, a existir.
  - Por ejemplo si el scope de la signatura `S` es `5` abrir `ordering[S]` forzará a `S` a tener `5` elementos, creando un orden lineal sobre esos `5` elementos

El módulo predefinido "Ordering"

```
module util/ordering[S]
    private one sig Ord {
        First, Last: S,
        Next, Prev: S -> lone S
    }
    fact {
    // todos los elementos de S están
    // totalmente ordenados
        S in Ord.First.*(Ord.Next)
    …
    }
```

El módulo predefinido "Ordering"

```
// las resticciones definen un orden total
     Ord.Prev = ~(Ord.Next)
     one  Ord.First
     one Ord.Last
     no Ord.First.Prev
     no Ord.Last.Next
```

Cuenta con todas les restricciones necesarias, como las funciones (primer elemento, ultimo elemento, siguiente, anterior, siguientes, anteriores) y predicados (está antes, está después …) necesarios

Volviendo a la solución de especificación de dinámica

Modelo

```
sig  Biblioteca { states: set State}
sig  Libro { escritoPor: set Autor}
sig  Autor {}

sig  State { coleccion: set Libro}
```

Ejemplo de operación sobre los estados

```
pred agregarAColeccion [s,s1: State, l:Libro]
  s1.coleccion = s.coleccion + l
```

Solución: "nuevo" patrón

Utilizar un patrón de trazas:

- Modela secuencias de ejecuciones sobre la máquina abstracta
- Crea un ordenamiento total sobre los estados
- Conecta estados sucesivos a través de operaciones
  - ✓ Todos los estados deben ser alcanzables

```
open util/ordering[State] as ord
…
fact traces {
   init [ord/first]
   all s:State - ord/last |
    let s1 = s.next |
        op1[s,s1] or … or opN[s,s1]
}
```

Chequeando propiedades "safety"

Este tipo de propiedades pueden chequearse con una aserción, dado que todos los estados son alcanzables.

```
…
pred safe [s:State] { … }

assert  allReachableSafe {
    all s:State | safe[s]
}
```

Chequeando propiedades "safety"

Este tipo de propiedades pueden chequearse con una aserción, dado que todos los estados son alcanzables.

```
…
pred safe [s:State] { … }

assert  allReachableSafe {
    all s:State | safe[s]
}
```

Se controlan propiedades que capturan los lemas:

- ***"lo que quiero que suceda en algún momento sucederá",***
- ***"lo que no quiero que suceda no sucederá"***

Máquina Abstracta: detalles

La máquina abstracta es una parte del modelo que **no puede ser modularizada**.
Esto se debe a la dependencia de comportamiento de la misma con respecto a la parte del modelo afectada

Máquina Abstracta: detalles

La máquina abstracta es una parte del modelo que **no puede ser modularizada**.
Esto se debe a la dependencia de comportamiento de la misma con respecto a la parte del modelo afectada

Modelo estático

```
sig Color { }
sig Light { color: Color }
```

Máquina Abstracta: detalles

La máquina abstracta es una parte del modelo que **no puede ser modularizada**.
Esto se debe a la dependencia de comportamiento de la misma con respecto a la parte del modelo afectada

Modelo estático

```
sig Color { }
sig Light { color: Color }
```

Modelo dinámico

```
sig Color { }
sig Light { }

sig State {color: Light -> one Color }
```

Solución "Nuevo" Patrón: opciones

El estado puede pensarse de manera global (primera columna) o local (última columna)

Estado global: signatura State

```
sig Color { }
sig Light { }
sig State {color: Light -> one Color }
```

Estado local: signatura Time

```
sig Time { }
sig Color { }

sig Light {color: Color one -> Time }
```