

# Métodos Formales para Ingeniería de Software

## Modelado de dinámica Módulos en Alloy

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

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 hablamos de las operaciones? ¿de los efectos?**

Alloy no considera la noción de tiempo o de estado mutable.

Resulta necesario modelar estas nociones explícitamente

## ¿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”

## ¿Qué podemos esperar de las operaciones?

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”

Para ello es necesario contar una signatura que nos permita definir la noción de estado y de máquina abstracta.

Vinculando esta maquinaria con el modelo estático del problema corriente.



## 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] }  
check initVerifies  
  
// para cada operación  
  
assert opPreserves {  
    all s,s1: State |  
        inv[s] && op[s,s1] => inv[s1]  
}  
check opPreserves
```

## 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 firmas

## 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}
```



## 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]}
```

## 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 restricciones 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 las 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] and
  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]  
}  
  
check allReachableSafe
```



## 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]  
}  
check allReachableSafe
```

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 }
```