Métodos Formales Livianos

Marcelo Frias

Departamento de Computación

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Nazareno Aguirre

Departamento de Computación

Facultad de Ciencias Exactas, Físico-Químicas y Naturales

Universidad Nacional de Río Cuarto

Clase 2: Álgebra Relacional - Lógica Relacional - Alloy

1 SAT Solving en Lógica de Primer Orden

Si bien el problema de satisfactibilidad de fórmulas **no es decidible** en lógica de primer orden, es útil aplicar una técnica similar a la aplicada para lógica proposicional.

El número de interpretaciones posibles para fórmulas en lógica de primer orden no puede, en general, reducirse a un conjunto finito, pues las variables pueden tomar valores arbitrarios de sus dominios, y los dominios pueden ser infinitos.

Sin embargo, si ponemos una **cota** k en el **número máximo de elementos en el dominio de una interpretación**, podemos examinar exhaustivamente **todas las interpretaciones posibles de "tamaño" a lo sumo** k. Si empleando este procedimiento **encontramos un modelo** de la fórmula en cuestión, entonces dicha fórmula **es satisfactible**. **Sino**, sólo podemos garantizar que **no existen modelos de la fórmula de "tamaño" menor o igual que** k

1.1 Un Ejemplo

Consideremos la fórmula siguiente:

$$\forall x : p(x, c) \wedge q(f(x, x))$$

Tenemos una función binaria (f), una constante (c), y predicados de aridades uno y dos (q y p, respectivamente). Cuántas interpretaciones posibles de “tamaño” 5 existen?

2 Poder Expresivo de la Lógica de Primer Orden

La lógica de primer orden es un formalismo muy expresivo, que permite especificar gran cantidad de propiedades (en el contexto de especificaciones sistemas de software).

Algunas propiedades, como por ejemplo aquellas relacionadas con minimalidad, no pueden expresarse en lógica de primer orden.

3 Álgebra Relacional

Antes de comenzar con el estudio de la lógica relacional (una lógica más expresiva que la lógica de primer orden), revisemos la definición de algunos operadores relacionales clásicos.

Sean R y S relaciones binarias sobre cierto dominio A.

- $R \cup S$ es la *unión* de las relaciones binarias R y S
- $R \cap S$ es la *intersección* de las relaciones binarias R y S
- $R; S$ es la *composición* de las relaciones binarias R y S, definida de la siguiente manera:

$$R; S = \{(a, b) | \exists c \in A : (a, c) \in R \wedge (c, b) \in S\}$$

- $\overline{R}$ es el *complemento* de la relación R (con respecto a A)

- $\check{R}$ es la *conversa* de la relación R, definida de la siguiente manera:

$$\check{R} = \{(b, a) | (a, b) \in R\}$$

- R^* es la *clausura transitiva* de la relación R, definida como la *mínima relación transitiva que contiene a* R

Estas operaciones poseen propiedades algebraicas: $\emptyset; R = \emptyset$, $(R; S); T = R; (S; T), \ldots$

4 Alloy: Un Lenguaje de Primer Orden basado en Relaciones

Alloy es un lenguaje de especificaciones de primer orden basado en el uso de relaciones. Alloy está orientado a la especificación de propiedades estructurales de sistemas.

Alloy intenta resolver la complejidad en la expresión de algunas propiedades estructurales comunes en lógica de primer orden utilizando operadores relacionales, definidos en la *lógica relacional*, el formalismo subyacente a Alloy.

5 Sintaxis de Alloy

5.1 Signaturas

Las signaturas son la base del lenguaje Alloy. Éstas permiten, entre otras cosas, denotar dominios.

Consideremos, como un ejemplo inicial, la especificación de un sistema de memorias. En principio, parece necesario contar con los dominios de las direcciones (de memoria) y de datos (a almacenarse en direcciones de memoria). Estos dominios se pueden especificar en Alloy de la siguiente manera:

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

5.2 Signaturas Complejas

Las signaturas anteriores son básicas, ya que no tienen estructura interna. Se pueden definir estructuras más complejas usando signaturas. Por ejemplo:

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

Esta signatura indica, intuitivamente, que una memoria consta de dos “campos”: un conjunto `addrs` de direcciones (subconjunto de `Addr`) y una función total de `addrs` en `Data`.

5.3 Extensión de Signaturas

Pueden definirse tipos más específicos de signaturas extendiendo otras signaturas existentes. Por ejemplo:

```
sig MainMemory extends Memory { }

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

Aquí, se define `MainMemory` como un tipo particular de `Memory`, sin estructura adicional; `Cache`, por otra parte, se define como un tipo particular de `Memory` en el cual un subconjunto de su espacio de direcciones se reconoce como “sucio”.

Para completar el ejemplo, podemos considerar una signatura que defina sistemas:

```
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`).

5.4 Operaciones en un Modelo Alloy

Las signaturas permiten describir dominios y sus estructuras. Siguiendo el estilo de Z, se pueden especificar operaciones mediante *expresiones* que relacionan el estado previo a la aplicación de una operación y el estado posterior correspondiente:

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

La función `Write` transforma una memoria, sobreescribiendo el valor asociado a una dirección `a` con `d`.

Esta función, como los esquemas para operaciones en Z, no está directamente asociada a ninguna signatura. Las variables “primadas” corresponden a estados posteriores a la ejecución de la operación (es una convención).

5.5 Hechos (*facts*)

Por supuesto, las especificaciones acerca de la estructura de un sistema que uno puede lograr mediante el uso exclusivo de signaturas son limitadas. Generalmente, se necesita complementar las signaturas con restricciones estructurales adicionales.

Por ejemplo, podríamos complementar la especificación anterior con un *hecho* que las memorias principales no son caches y viceversa:

```
fact {no (MainMemory & Cache)}
```

5.6 Aserciones

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

```
assert {
   all s : System |
          no s.cache.dirty => s.cache.map in s.main.map
  }
```

6 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),
- el lenguaje hace que las especificaciones sean *analizables* automáticamente, usando técnicas de SAT solving (para validación, pues el lenguaje es de primer orden)

7 Actividades

1. Averigüe acerca de la semántica de los distintos operadores de la lógica relacional y la semántica de las construcciones de Alloy.
2. Cuál es la semántica del operador ‘·’?
3. Instale el Alloy Analyzer.
4. Estudie la documentación provista en “Micromodels of Software: Lightweight Modelling and Analysis with Alloy” (D. Jackson 2002).
5. Complete el modelo de memorias con cache y compruebe, usando el Alloy Analyzer, si la aserción vista en estas notas es consecuencia del modelo obtenido.
6. Modele grafos dirigidos y algunas de sus operaciones en Alloy. Modele con una aserción la aciclicidad de grafos.