CIS 771: Software Specifications

Lecture: Alloy Logic (part E)

Outline

- Declarations
- Set multiplicities
- Relational multiplicities
- Declaration constraints
- Nested multiplicities
- Cardinality constraints

Declarations

A *declaration* introduces a name for a relation whose value is a subset of the value of the *bounding expression* appearing after the ":"

```
relation-name : expression
```

Examples

Note: *the bounding expression is usually formed with unary relations and the arrow operator, but any expression can be used.*

- `address : Name -> Addr`
  - maps names to addresses (representing a single address book)
- `addr: Book -> Name -> Addr`
  - maps books to names to addresses (representing a collection of address books)
- `address: Name -> (Name + Addr)`
  - maps names to names and addresses (representing a multilevel address book)

Declarations

A *declaration* introduces a name for a relation whose value is a subset of the value of the *bounding expression* appearing after the ":"

```
relation-name : expression
```

Examples -- with more complicated expressions

- `address : (Alias + Group)->(Addr + Alias + Group)`
  - maps aliases and groups to addresses, aliases, and groups
- `address : (Alias->Group) + (Group->(Addr + Alias + Group)`
  - has the same "type" as the declaration above
    - aliases, groups in domain; addresses, aliases, groups in co-domain
  - more precise than the version above because it constrains aliases to only map to groups
  - illustrates how declaration expressions can combine relations (e.g., via the union operator) as well as sets

Set Multiplicities

If the *bounding expression* of a declaration denotes a set (is unary), it can be prefixed by a multiplicity keyword *m* which constrains the size of set `x` according to *m*.

`x : m e`

Bounding expression with multiplicity m.

Multiplicity Keywords -- similar to those used for quantification

- **`set`** any number
- **`one`** exactly one
- **`lone`** zero or one
- **`some`** one or more

For set-value bounded expressions, omitting the multiplicity keyword is the same as writing **`one`**.

Set Multiplicities

If the *bounding expression* of a declaration denotes a set (is unary), it can be prefixed by a multiplicity keyword *m* which constrains the size of set `x` according to *m*.

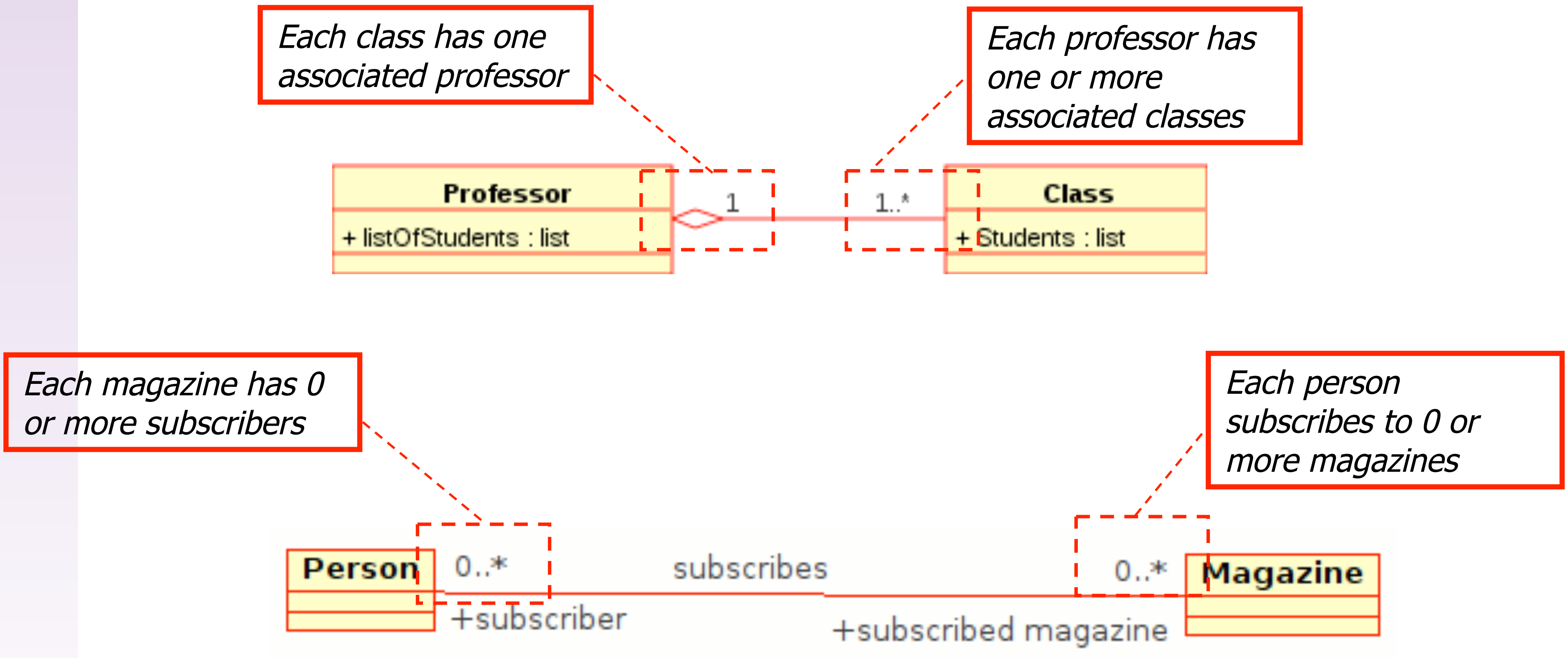
`x : m e`

Examples

- `RecentlyUsed: `**`set`**` Name`
  - says that `RecentlyUsed` is a subset of the set `Name`
- `senderAddress: Addr`
  - says that `senderAddress` is a scaler in the set `Addr`
- `senderName: `**`lone`**` Name`
  - says that `senderName` is an option: either a scaler in the set `Name`, or empty
- `receiverAddresses: `**`some`**` Addr`
  - says that `receiverAddresses` is a non-empty subset of `Addr`.

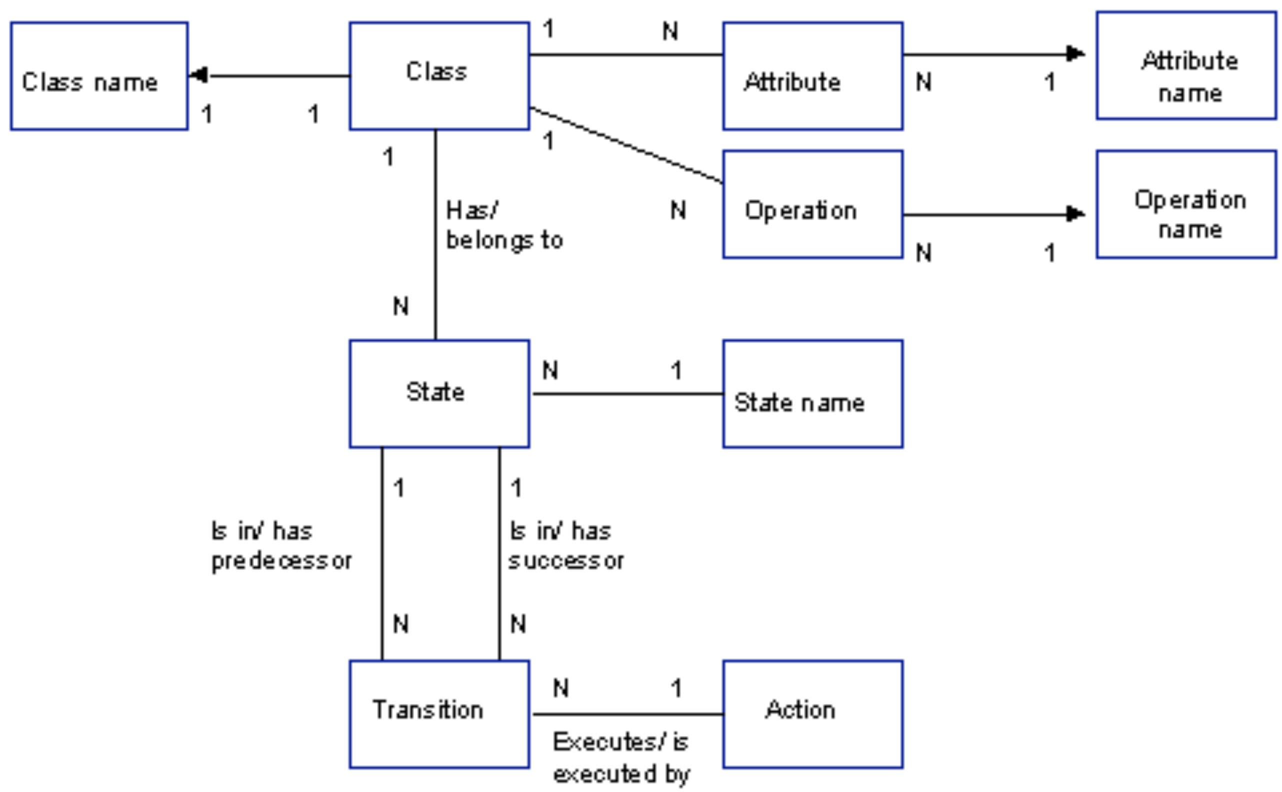
Relational Multiplicities

Examples of relational multiplicities in UML…



Relational Multiplicities

Examples of relational multiplicities in Entity-Relation models

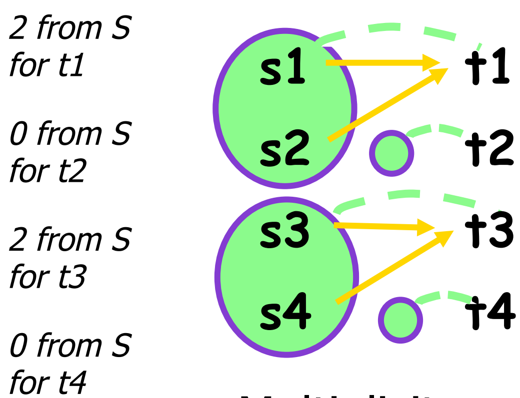


Relational Multiplicities in Alloy

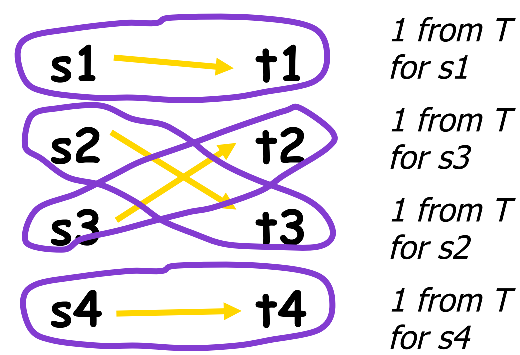
Multiplicities in Alloy can be placed on the elements of a relation declaration…

r : S *m* -> *n* T

- A multiplicity *m* on the left (domain) tells you the size of the set associated with each range set element.
- A multiplicity *n* on the right (range) tells you the size of the set associated with each domain set element.



Multiplicity: set S

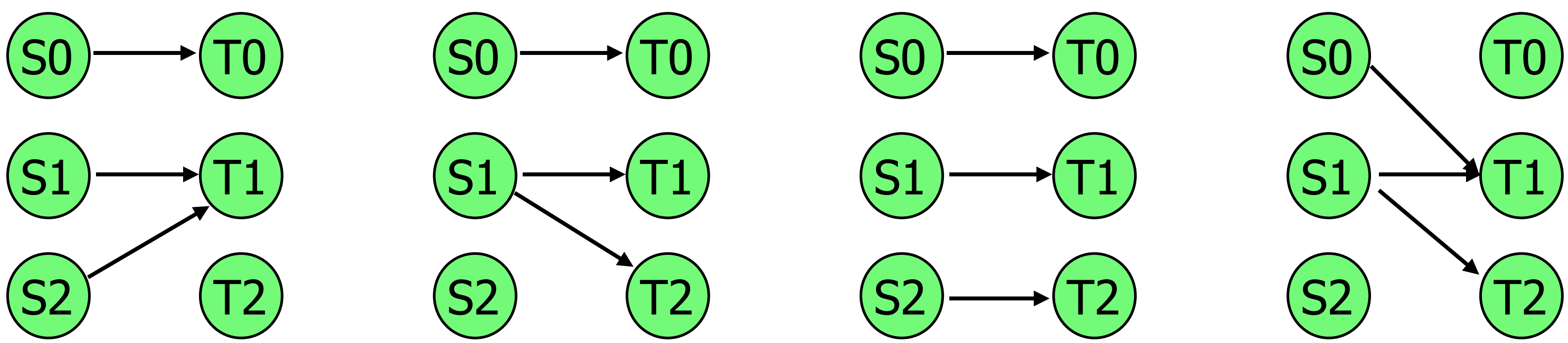


Multiplicity: one T

Functions and Injections

Examples

S -> one T | S one -> T | S one -> one T | S -> T



functional, but not injective

injective, but not functional

functional, and injective

neither functional, nor injective

Note: *due to the "at most one" in the definitions below, it would be valid to replace the instances of* one *above with* lone

- A binary relation that maps each atom to at most one other atom is said to be *functional*, and is called a *function.*
- A binary relation that maps at most one atom to each atom is *injective*.

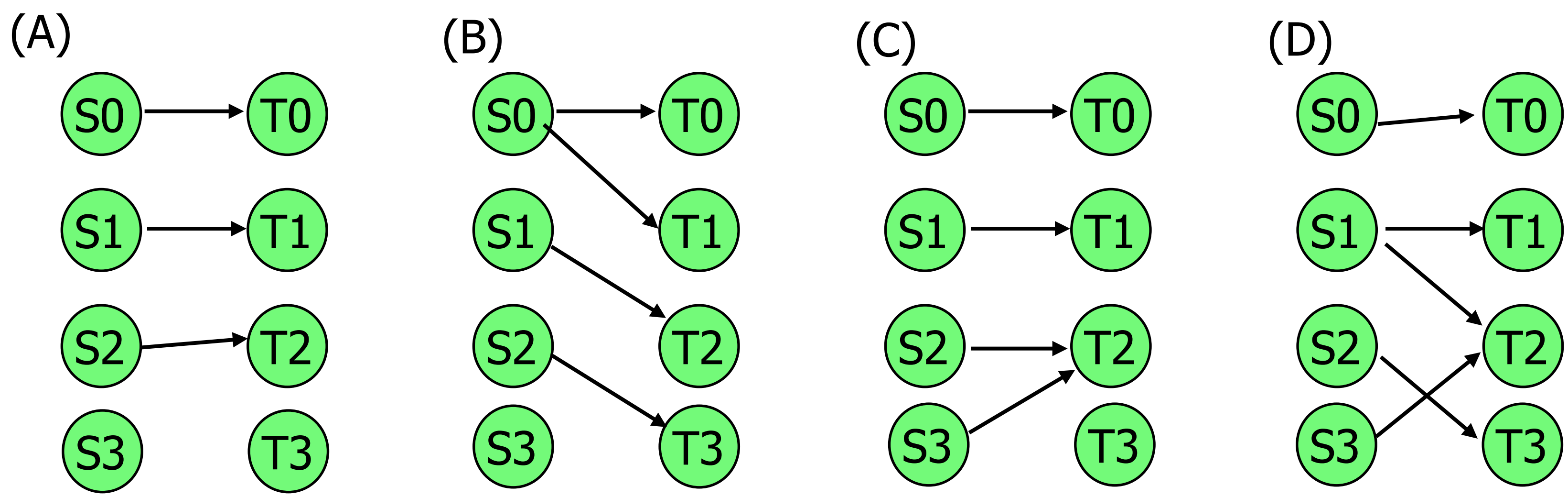
Common Relations via Multiplicities

Examples

- r: A -> one B
  - A (total) function with domain A, range B
- r: A one -> B
  - An injective relation
- r: A -> lone B
  - A (partial) function
- r: A one -> one B
  - An injective (total) function
- r: A some -> some B
  - A surjective relation

For You To Do

- For each of the relations, give the most precise declaration with multiplicities.



- For each of the declarations below, draw two relations with the same domain / range as the relations above such that the first relation satisfies the multiplicities in the declaration while the second one violates it.
  - r: S some -> lone T
  - r: S set -> one T
  - r: S lone -> lone T
  - r: S some -> some T

Multiplicities are a Shorthand

Multiplicities are just a shorthand, and can be replaced by standard constraints…

r : A *m* -> *n* B

can be written as…

```
all a: A | n a.r
all b: B | m r.b
```

Example

```
members: Group lone -> some Addr
```

can be written as…

```
all g: Group | some g.members
all a: Addr  | lone members.a
```

Generalizing to Tuples

In the schema below, *A* and *B* can be arbitrary expression, and don't have to be relation names.

r : *A m* -> *n B*

...in such a case, this says that r maps m tuples in A to each tuple in B, and maps each tuple in A to n tuples in B.

Example

```
addr: (Book -> Name) -> lone Addr
```

...says that that relation `addr` *associates at most one address with each address book / name pair.*

✓
```
{(B0, N0, A1)
 (B0, N1, A2)
 (B1, N0, A3)}
```

✗
```
{(B0, N0, A1)
 (B0, N0, A2)
 (B1, N0, A3)}
```

Declaration Constraints

Declaration syntax can also be used to impose constraints on relations that have already be declared, or on arbitrary expressions…

Original declaration…

```
address: (Group + Alias) -> Addr
```

…imposing additional constraints somewhere later in the model…

```
(Alias <: address): Alias -> lone Addr
```

…says that each alias maps to at most one address

Declaration constraints, like any other formula, can be combined with logical operators, placed inside the body of quantifications, etc.

```
all b: Book | b.addr: Name lone -> Addr
```

…says that each address book is injective (maps at most one name to an address)

Nested Multiplicities

Multiplicities can be nested

A declaration of the form…

r: A -> (B *m* -> *n* C)

…means that for each tuple in A, the corresponding tuples in B -> C form a relation with the given multiplicity. In the case that A is a set, the multiplicity constraint is equivalent to…

```
all a: A | a.r : B m -> n C
```

Example

```
addr: Book -> (Name lone -> Addr)
```

…says that, for any book, each address is associated with at most one name, and is equivalent to…

```
all b: Book | b.addr: Name lone -> Addr
```

…whereas…

```
addr: (Book -> Name)lone -> Addr
```

…says that each address is associated with at most book/name combination. The first (but not the second) allows an address to appear in more than one book

For You To Do

- For each of the relation declarations below, give two relation instances such that the first relation satisfies the declaration while the second relation violates the declaration…
  - `addr: (Book -> Name) -> some Addr`
  - `addr: lone Book -> (Name -> Addr)`
  - `addr: one Book -> (one Name -> some Addr)`
- For each of the relation declarations above, write a declaration that does not include multiplicities but instead is accompanied by explicit constraint that achieve the same effect as the declarations above
- Given the declaration below, write a declaration constraint that enforces the additional property that each group maps to one or more addresses
  - `address: (Group + Alias) -> Addr`

Cardinality Constraints

The operator # applied to a relation gives the number of tuples it contains, as an integer value.

```
Address: (Group + Alias) -> Addr
  …
all g: Group | #g.address > 1
```

…says that every group has more than one address associated with it

```
addr: Book -> Name -> Addr
 …
all a: Addr | #addr.a <= 5
```

…says that the number of book/name pairs associated with each address is less than or equal to 5

Sum Expression

The expression below denotes the integer obtained by summing the values of the integer expression *ie* for all values of the scalar drawn from the set *e*.

```
sum x: e | ie
```

Example

```
addr: Book -> Name -> Addr

sum b : { b:Book | #b.addr > 5 } | #b.addr
```

…gives the total number of entries across all books that have great than five entries

Acknowledgements

- The material in this lecture is based on Sections 3.6 and 3.7 from…
  - *Software Abstractions: Logic, Language, and Analysis,* Daniel Jackson, MIT Press, 2006.