Navigation in Object Graphs

Mitch Wand

and

Karl Lieberherr

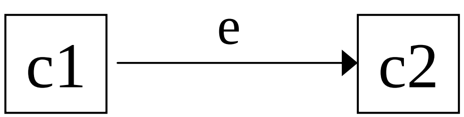
Searching for Reachable Objects

- Task: Given an object o1 of class c1 in an object graph, find all objects of type c2 that are reachable from o1.
- Assumptions: we know the class structure that describes the object graph, but we know nothing else about the object graph except the class of the current object.

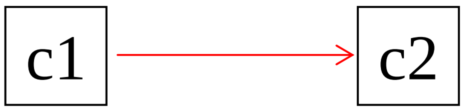
Search using meta information

- we could visit the entire object but that
  - would be wasteful or
  - might lead to wrong results

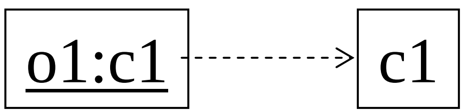
Classes and Objects: Basic Notations



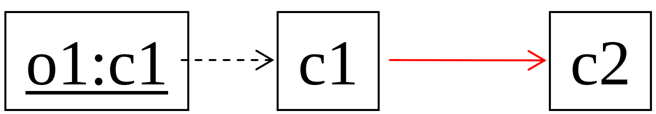
Class c1 has a part e of type c2



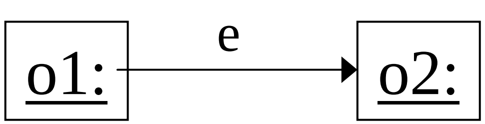
Class c1 inherits from class c2



Object o1 is of class c1

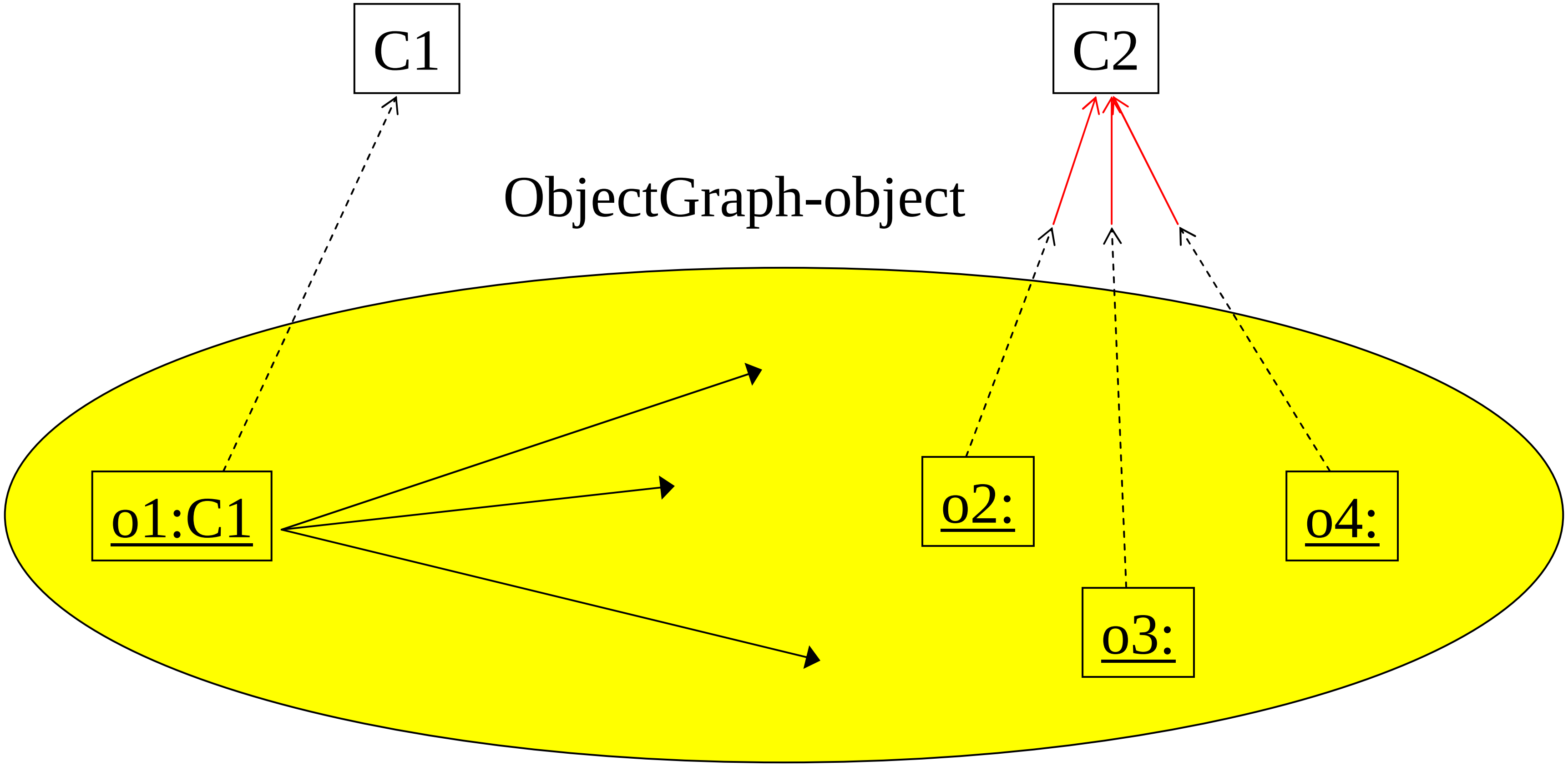


Object o1 is of type c2
(i.e., its class is a subclass of c2)



Object o1 has a part e which is object o2

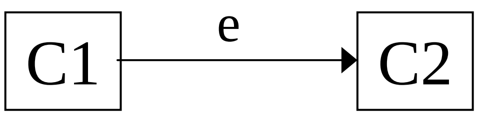
Finding the first step for the search



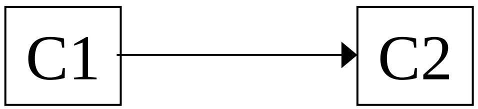
Which arrows might lead
to an object of type C2?

Traversal Strategy:
(// C1 C2)

Relations between Classes

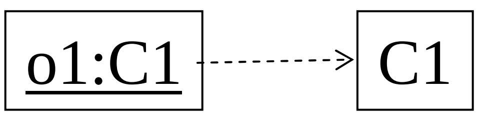


e(C1,C2)

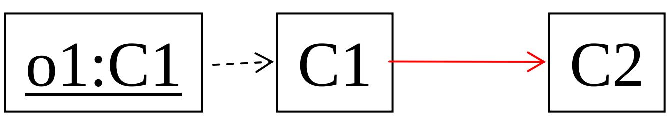


C(C1,C2)   (that is, e(C1,C2) for some e)

<=(C1,C2), =>(C2,C1)



Class(o1) = C1



Object o1 is of type C2:
Class(o1) <= C2

Relations between Objects

$\boxed{\underline{\text{o1:}}} \xrightarrow{e} \boxed{\underline{\text{o2:}}}$ e(o1,o2)

$\boxed{\underline{\text{o1:}}} \longrightarrow \boxed{\underline{\text{o2:}}}$ O(o1,o2) (that is, e(o1,o2) for some e)

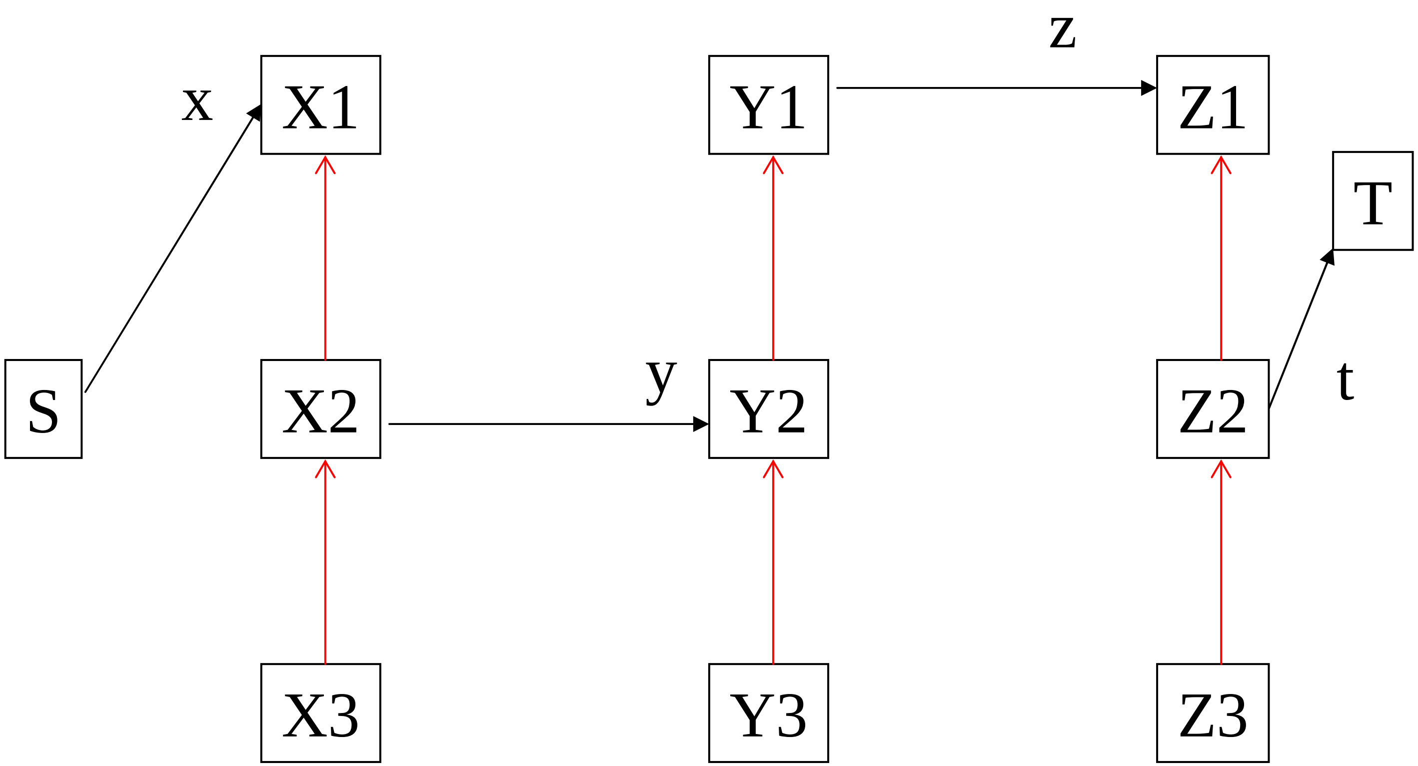
Operations on Relations

- R.S = {(x,z) | exists y s.t. R(x,y) and S(y,z)}
- R* = reflexive, transitive closure of R

Write graph in terms of relations

Set: {S,T,X1,X2,X3,Y1,Y2,Y3,Z1,Z2,Z3}

Relations are sets of pairs, ordering is irrelevant.



Relations:

x,y,z,t,<=,=>

x(S,X1)

y(X2,Y2)

z(Y1,Z1)

t(Z2,T)

<=(Y2,Y1)

<=(X2,X1)

<=(X3,X2)

…

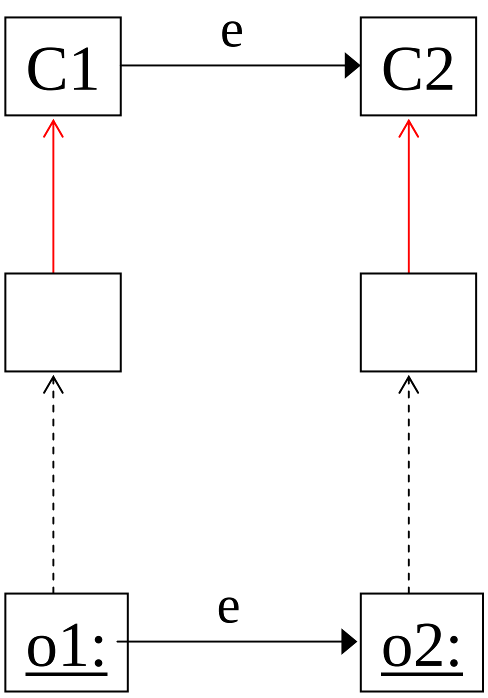
=>(X2,X3)

=>(X1,X2)

=>(Z1,Z2)

…

Possible edges in the object graph

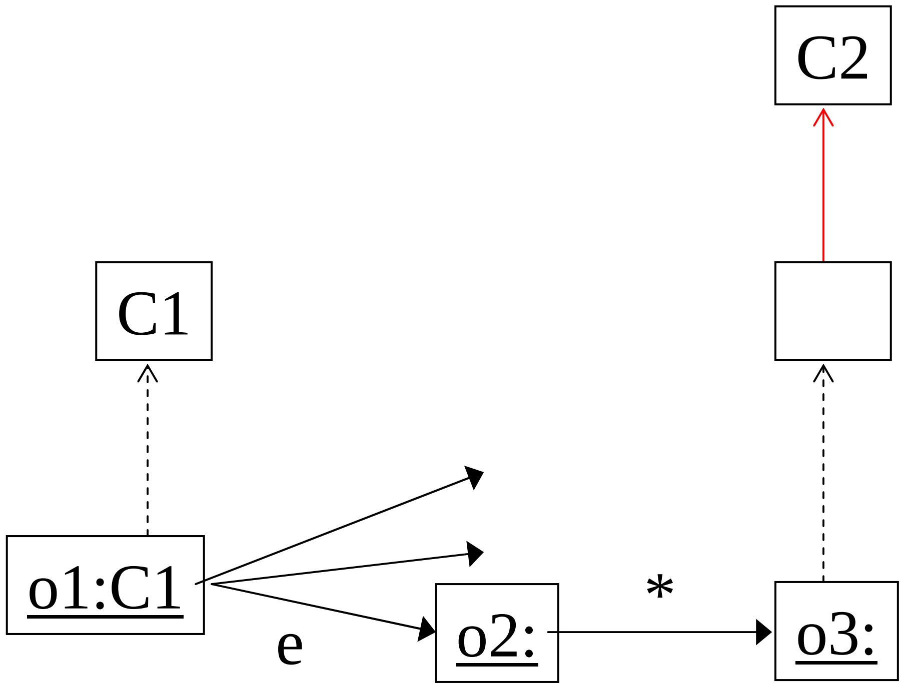


e(o1, o2) implies
class(o1) (<= .e .=>) class(o2)
in the class graph

"up, over, and down"

O(o1, o2) implies
class(o1) (<= .C .=>) class(o2)
in the class graph

Which edges to follow to C2?



From o1 of class C1, follow edge e iff *there is some object graph* O and some o2, o3 s.t.

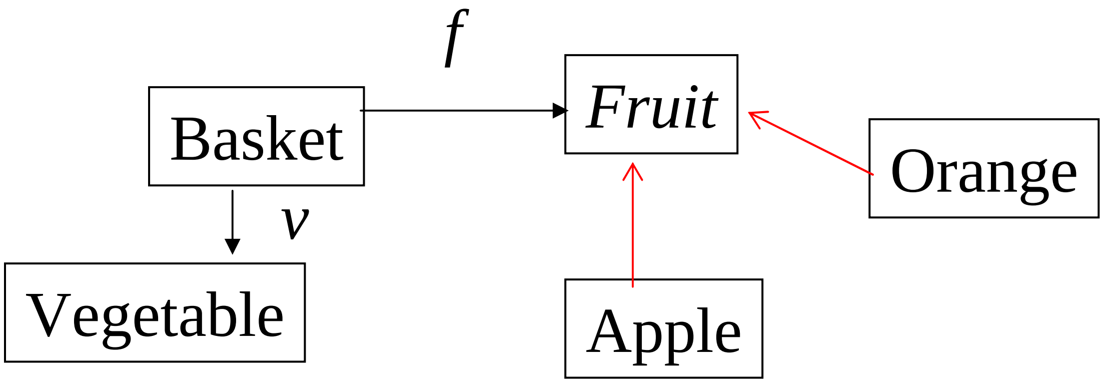
(1) e(o1,o2),

(2) O*(o2,o3), and

(3) class(o3) <= C2

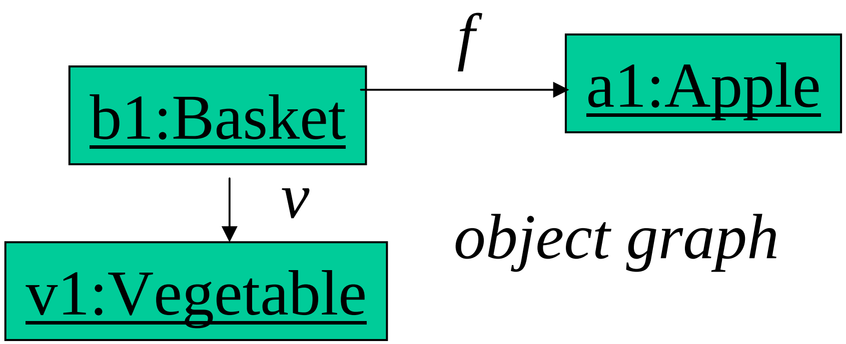
The existential quantifier "there is some object graph" represents our lack of knowledge about the rest of the object graph from Basket to Orange = (from-to Basket Orange) = (// Basket Orange)

Example

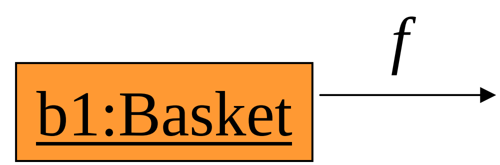


class graph

premature termination



object graph



object graph slice

(// Basket Orange)

Example

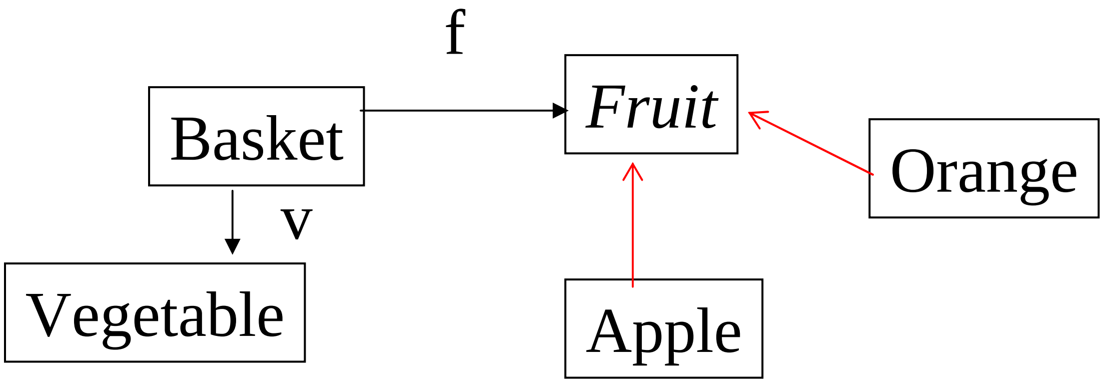
mapping:

o1 b1

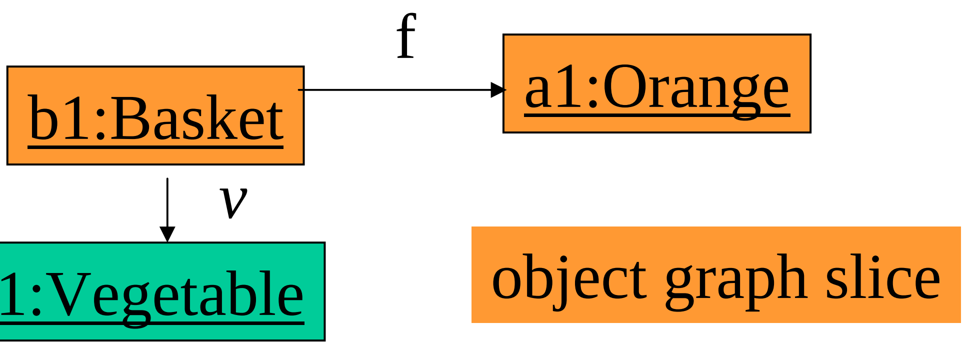
o2 a1

o3 a1

e f



class graph

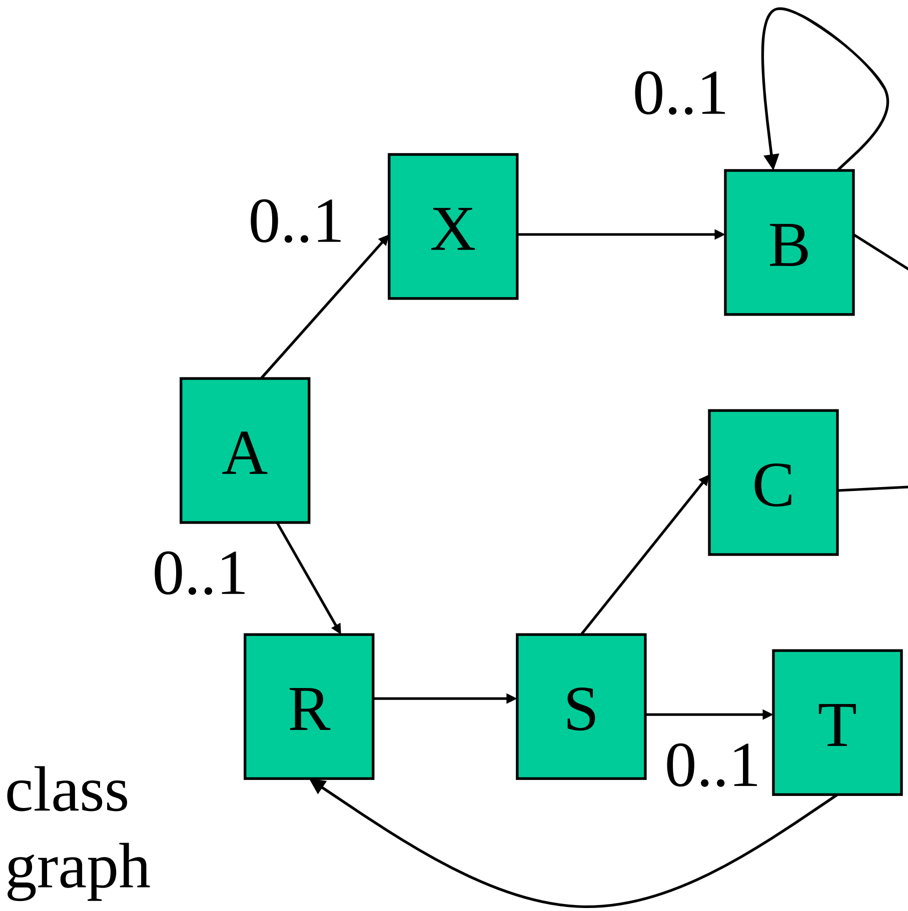


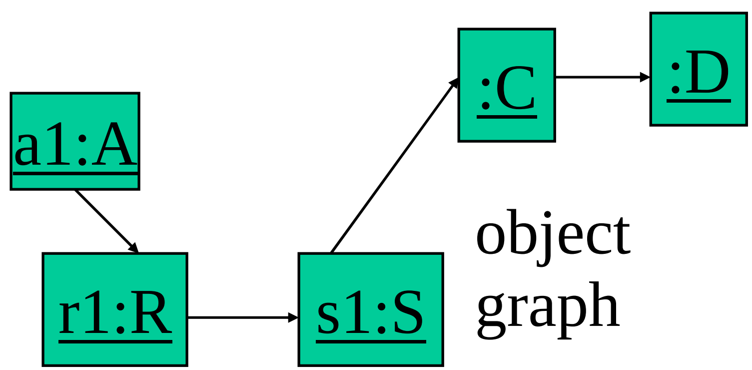
object graph

Example B1

strategy

```
from A via T to D=
(join (// A T)
      (// T D))
```

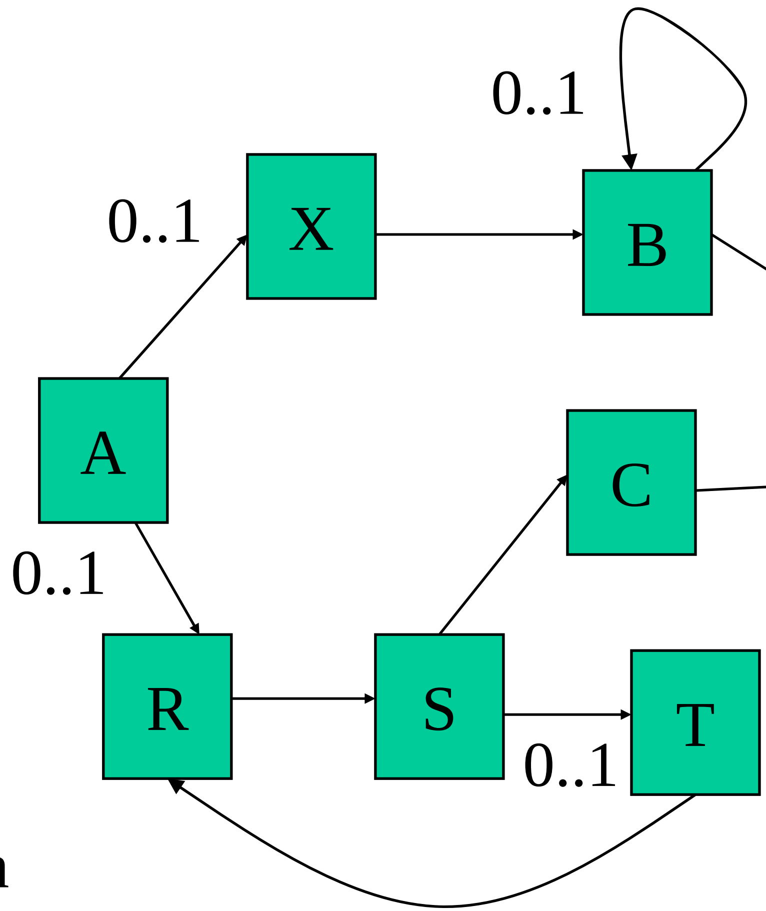




Example B1

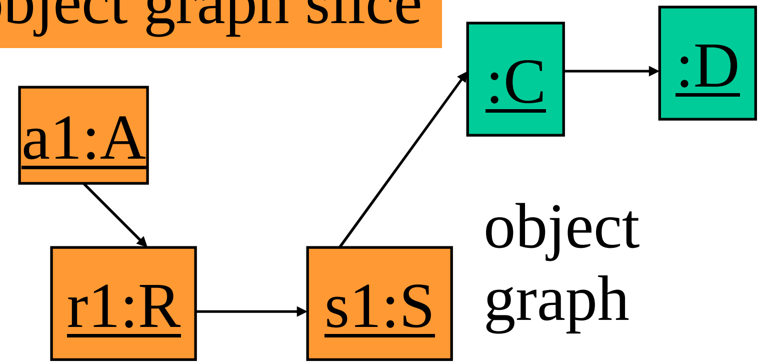
strategy

```
from A via T to D=
(join (// A T)
      (// T D))
```



class graph

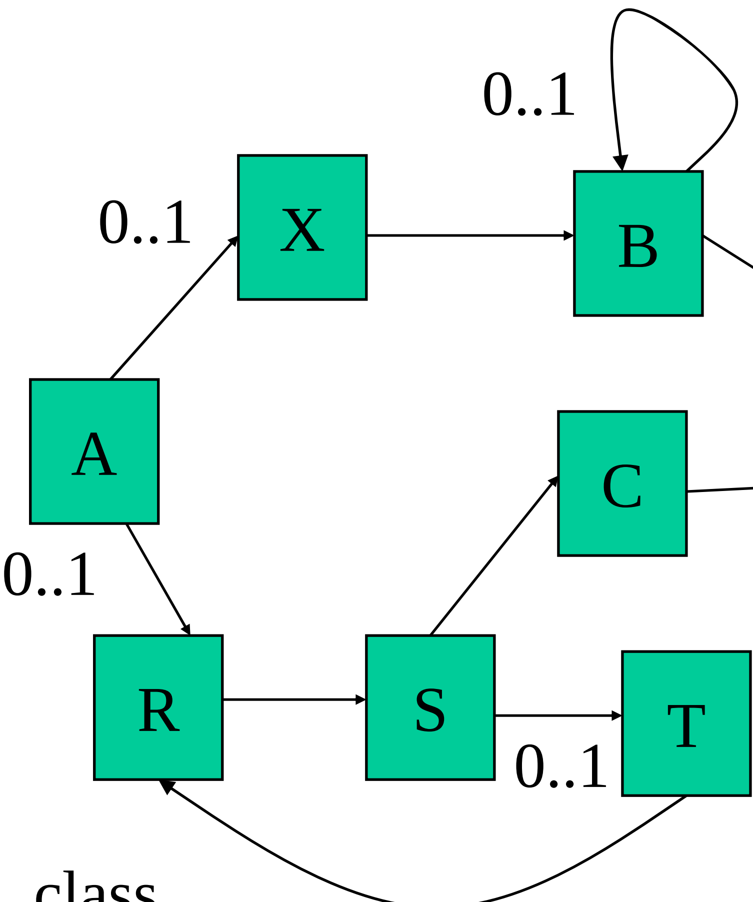
object graph slice



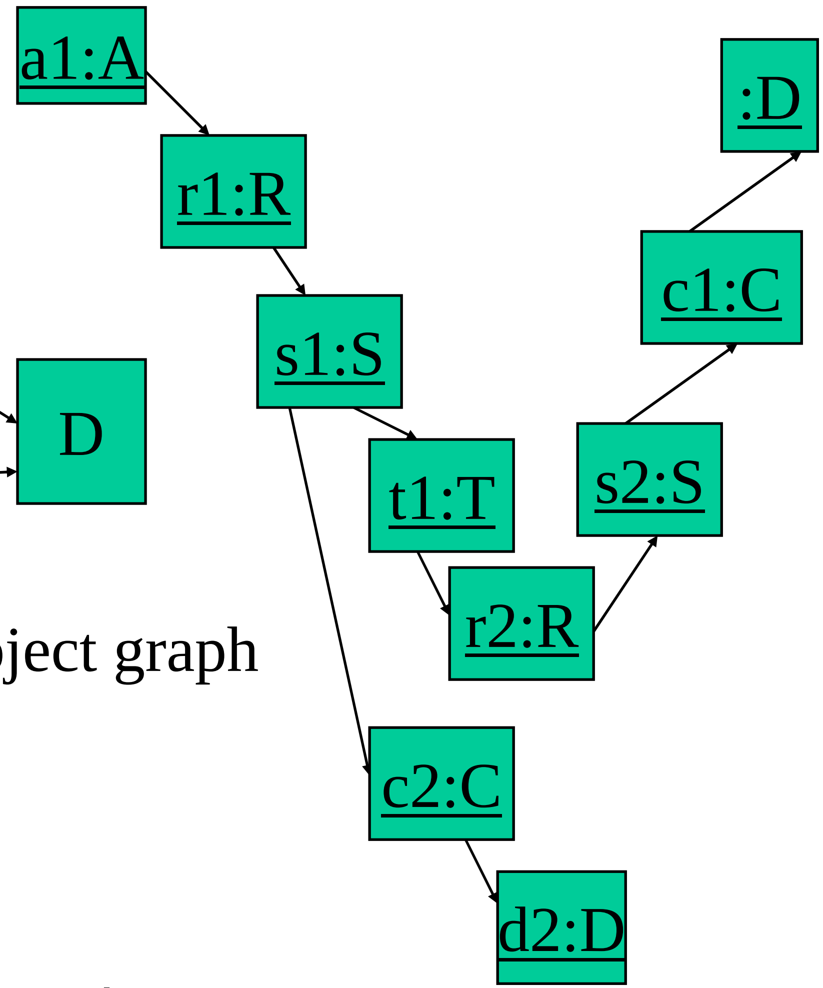
object graph

Example B2

strategy
```
from A via T to D=
 (join (// A T)
       (// T D))
```



class graph

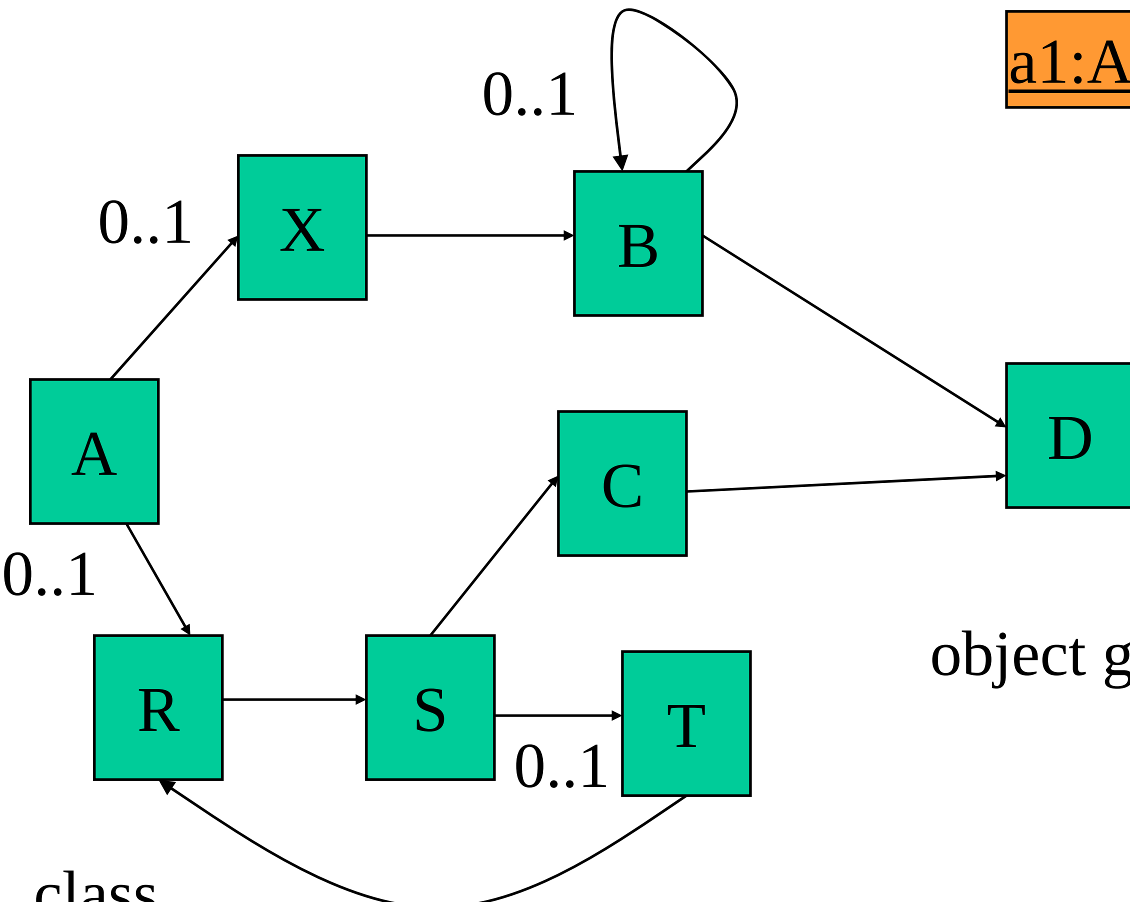


object graph

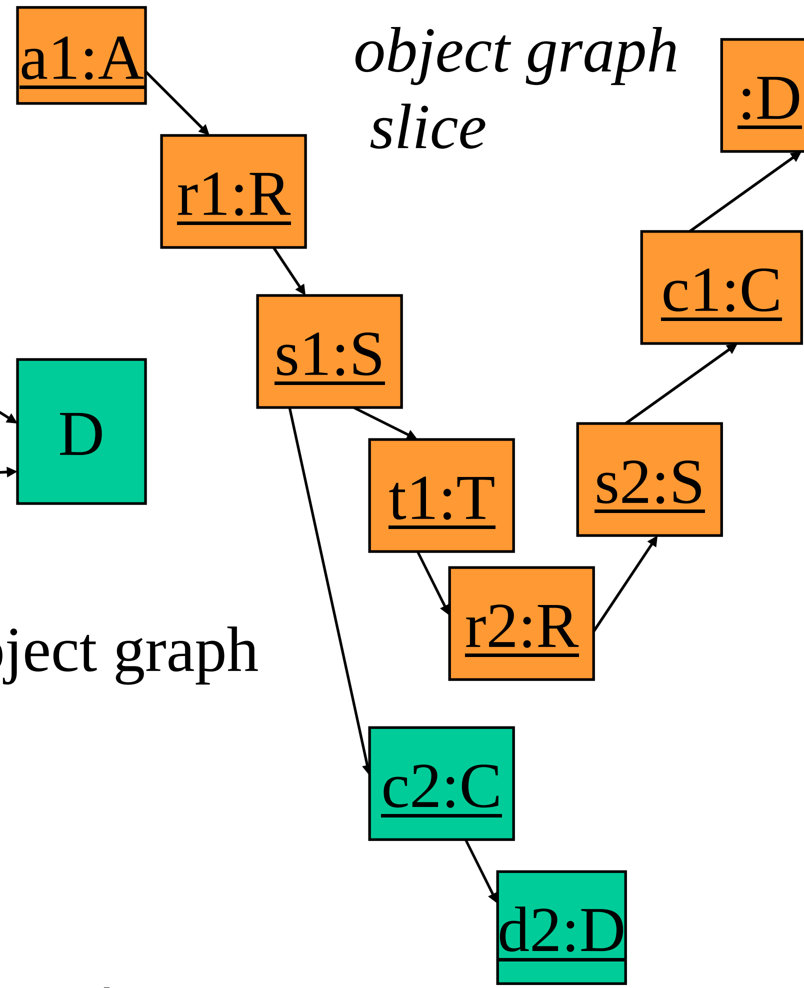
Example B2

strategy

```
from A via T to D=
(join (// A T)
      (// T D))
```



class graph



object graph

Lack of Knowledge

- Objects of a given class may be very different.
- We want to go down edges without looking ahead!
- We don't want to go down edges that are guaranteed to be unsuccessful (never reaching a target object).

Object graph conforms to class graph

- The object graph O must follow the rules of the class graph: the object graph cannot contain more information than the class graph allows.

For all edges e(o1,o2) in the object graph:
e(o1, o2) implies
class(o1) (<= .e .=>) class(o2)
in the class graph

From dynamic to static characterization

From o1 of class c1, follow edge e iff there is some object graph O and some o2, o3 s.t.

1. e(o1,o2),
2. O*(o2,o3), and
3. class(o3) <= c2

From o1 of class c1, follow edge e iff there are classes c’, c’’ s.t.

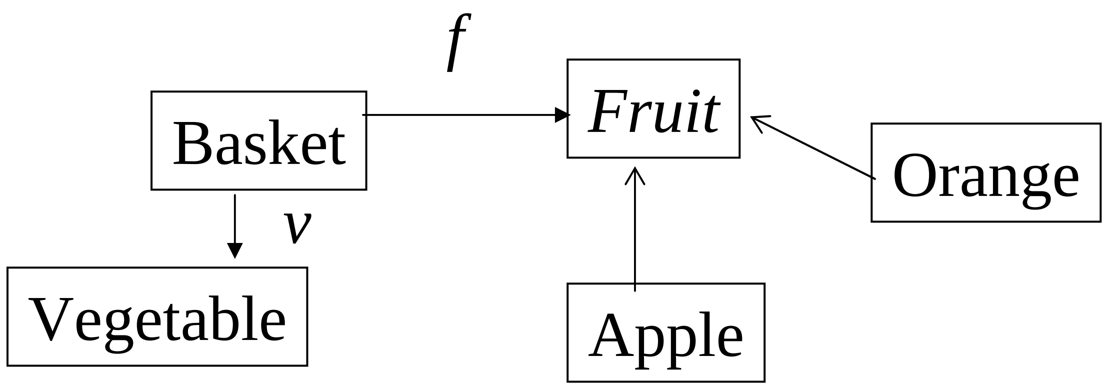
1. c1 <=.e.=> c’
2. c’ (<=.C.=>)* c’’ and
3. c’’ <= c2

Let c’ be class(o2), c’’ be class(o3)

from Basket to Orange

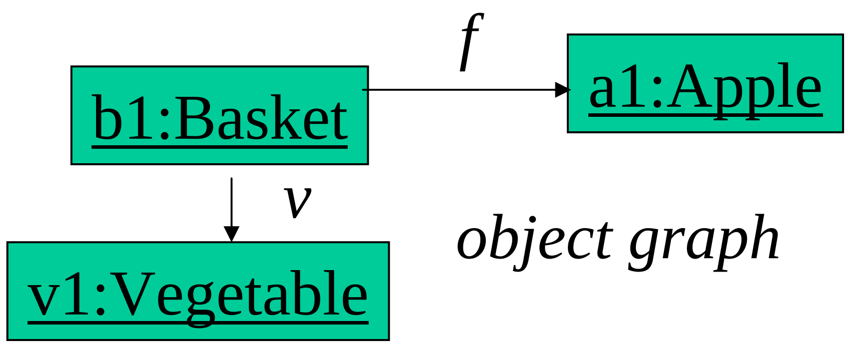
static characterization

Example

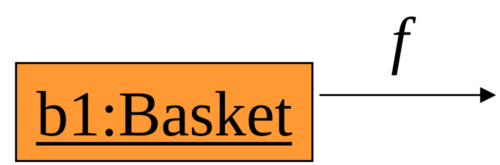


class graph

premature termination



object graph



object graph slice

from Basket to Orange

static characterization

mapping:

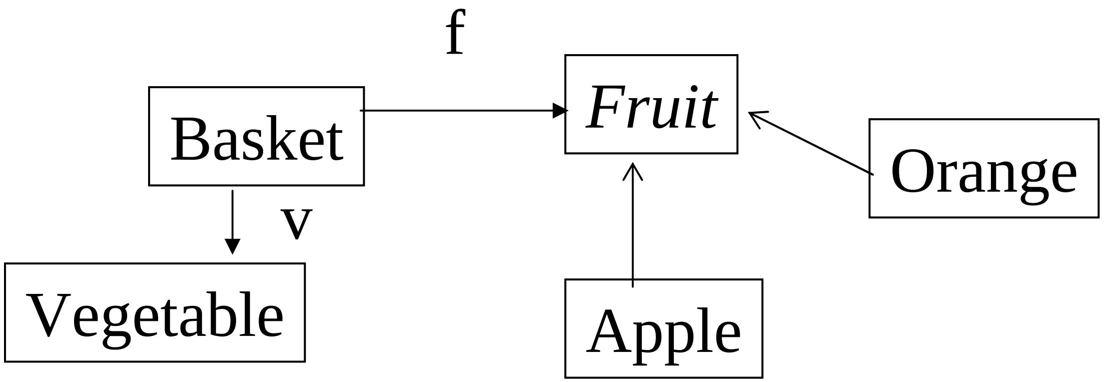
c1 Basket

c’ Orange

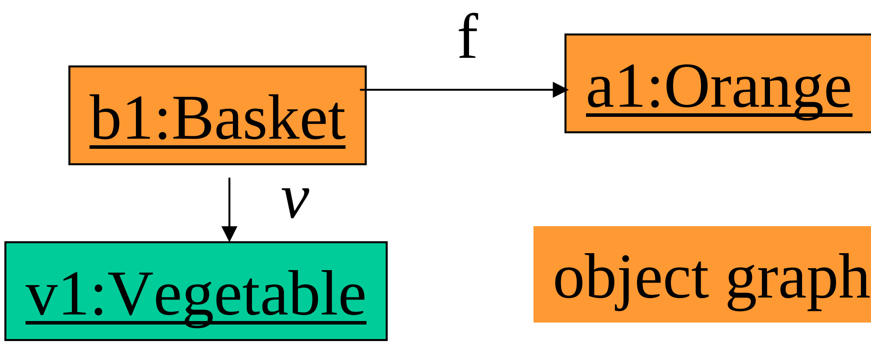
c’’ Orange

e f

Example



class graph



object graph

object graph slice

from x1 to x2

Relational Formulation

From object o of class x1, to get to x2, follow edges in the set
POSS(x1,x2)={e | x1 <=.e.=>.(<=.C.=>)*.<= x2 }

Can easily compute these sets for every x1, x2
via transitive-closure algorithms.
POSS = abbreviation for: following these edges it
is still possible to reach a x2-object for some x1-
object rooted at o.

from x1 to x2

Relational Formulation

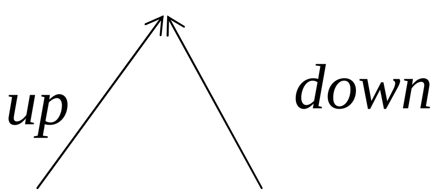
From object o of class x1, to get to x2, follow edges in the set
POSS(x1,x2)={e | x1 <=.e.=>.(<=.C.=>)*.<= x2 }

Simplification for class graphs obtained from hw class graphs:
POSS(x1,x2)={e | x1 e.=>.(C.=>)*.<= x2 } (flat)
up-over-and-down becomes over-and-down
=> means only one edge
<= means only one edge

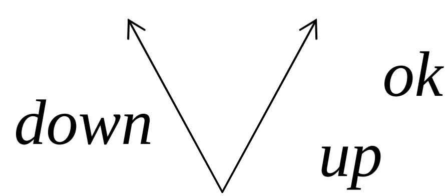
Negative formulation

Positive Formulation:
From object o of class x1, to get to x2, follow edges e in the set
POSS(x1,x2)={e | x1 <=.e.=>.(<=.C.=>)*.<= x2 }

Build paths anyway you like but don't follow => (down) immediately after <= (up) and the first has-a edge must have label e.



Edge kinds:
Is-a (down, up)
Has-a



Forbidden

ok

Followed-by relationships

Followed by	Is-a: down =>	Is-a: up <=	Has-a e, C
Is-a: down	yes	yes	yes
Is-a: up	NO	yes	yes
Has-a	yes	yes	yes

Linking the terminologies

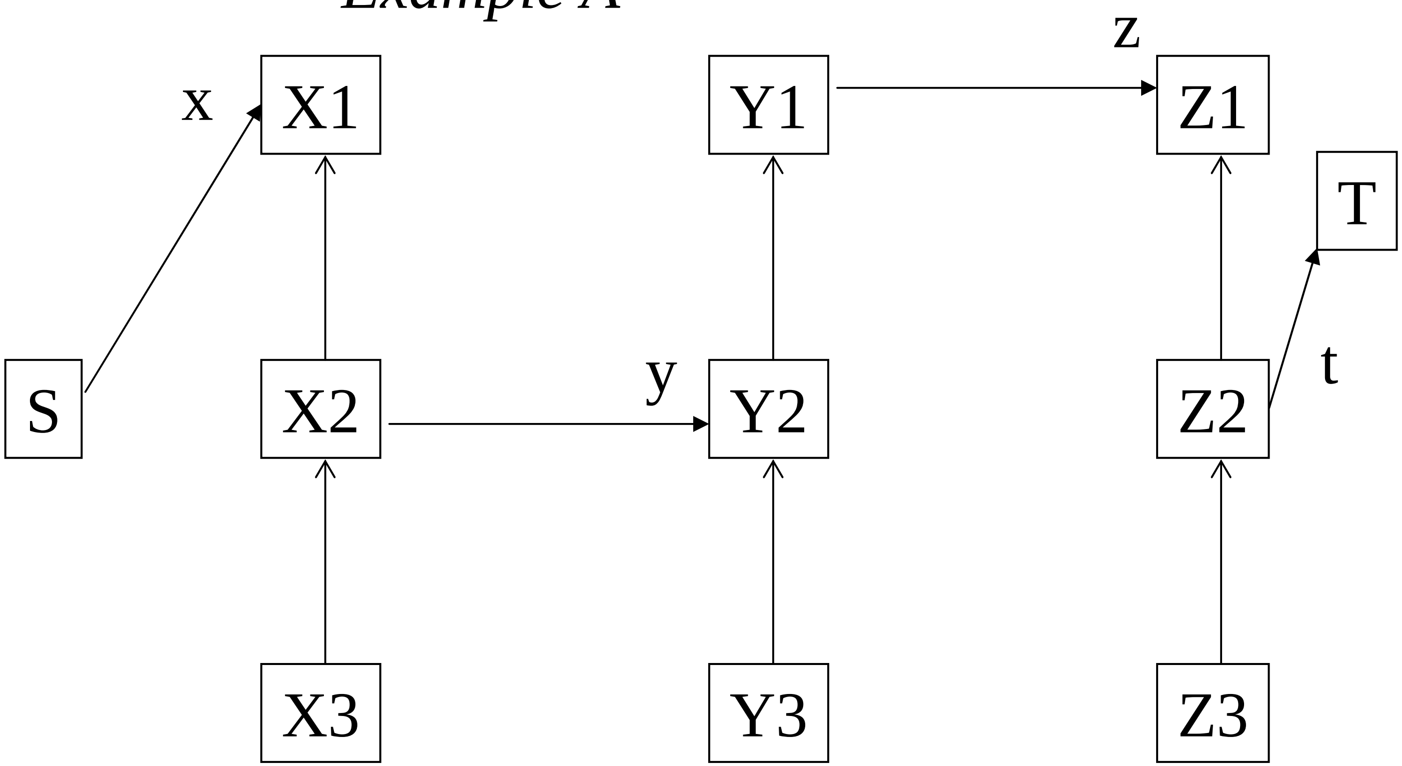
program			theory
`->A,b,B`	Has-a	over	e, C
`=> A,B`	Is-a in reverse	down	=>
`:> A,B`	Is-a	up	<=

Generalizations

- More complex strategies
- “from c1 through c2 to c3”
  - Use “waypoint navigation”; get to a c2 object, then search for a c3 object.
- More complex strategy graphs also doable in this framework

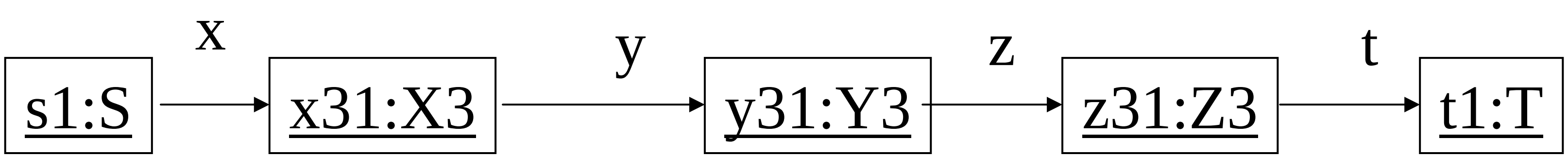
Examples

Example A



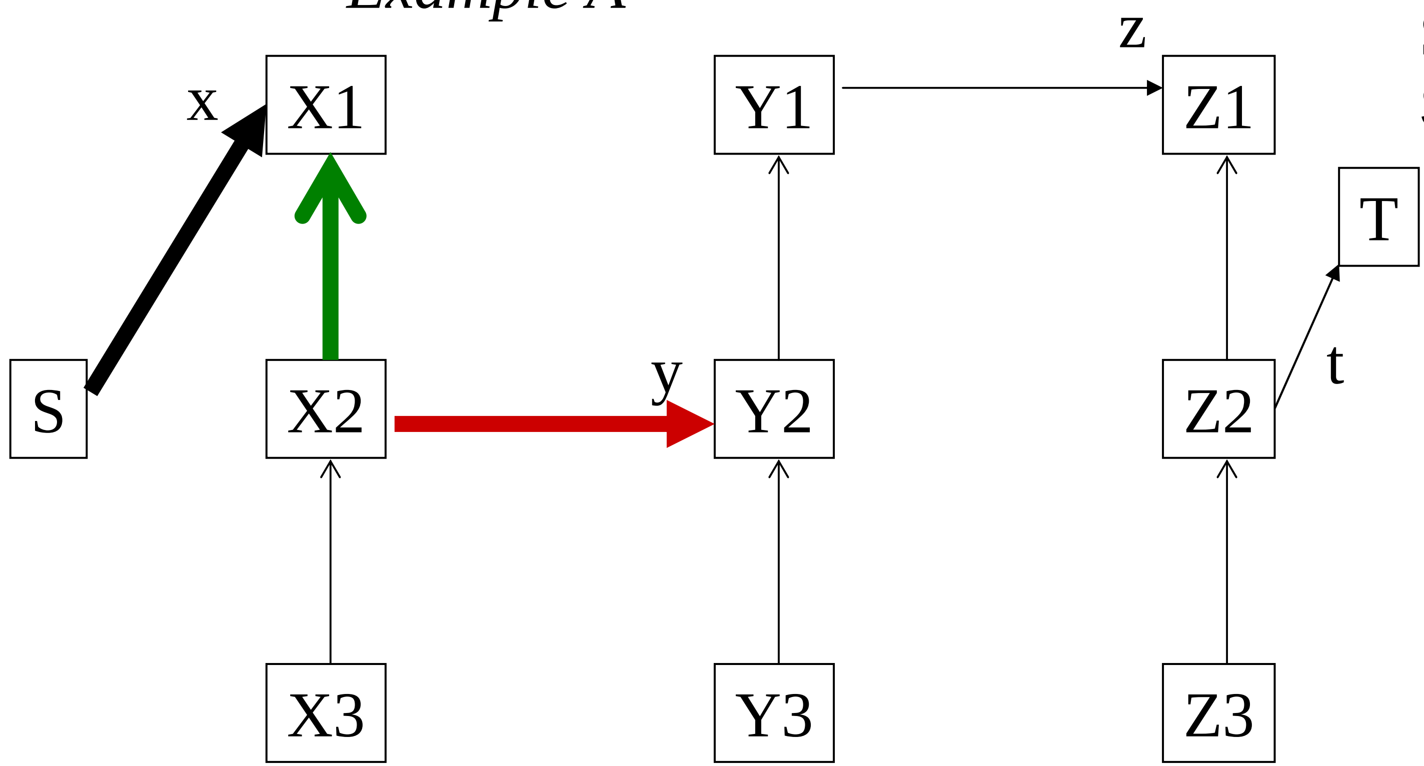
Strategy
S -> T

Relations:
x(S,X1)
=>(X1,X2)
y(X2,Y2)
<=(Y2,Y1)
z(Y1,Z1)
=>(Z1,Z2)
t(Z2,T)
<=(X2,X1)
<=(X3,X2)
=>(X2,X3)
…



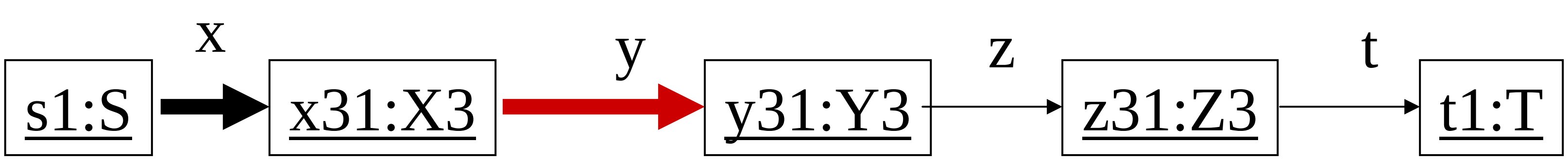
go down x iff S and T are in relation: <=.x. =>.(<=.C.=>)*.<=

Example A



Strategy
S -> T

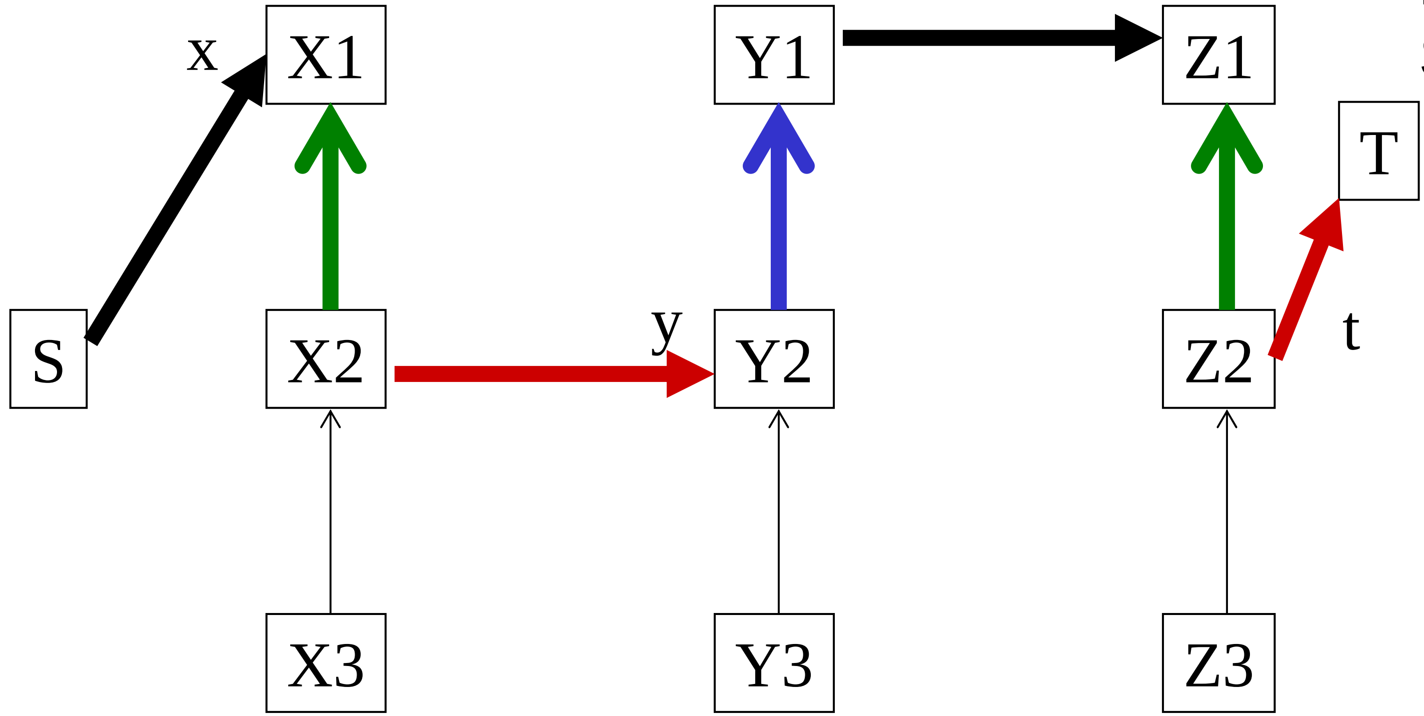
Relations:
x(S,X1)
=>(X1,X2)
y(X2,Y2)
<=(Y2,Y1)
z(Y1,Z1)
=>(Z1,Z2)
t(Z2,T)
<=(X2,X1)
…



<=,=>
not
used

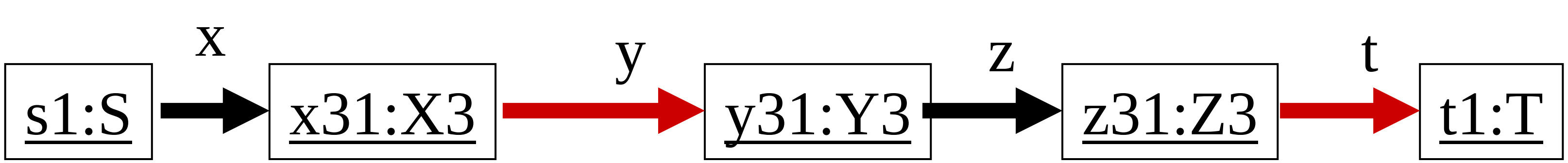
go down x iff **S** <=**.x.** =>**.(**<=**.**C**.**=>**)*.<= T**

Example A



Strategy
S -> T

Relations:
x(S,X1)
=>(X1,X2)
y(X2,Y2)
<=(Y2,Y1)
z(Y1,Z1)
=>(Z1,Z2)
t(Z2,T)
<=(X2,X1)
…

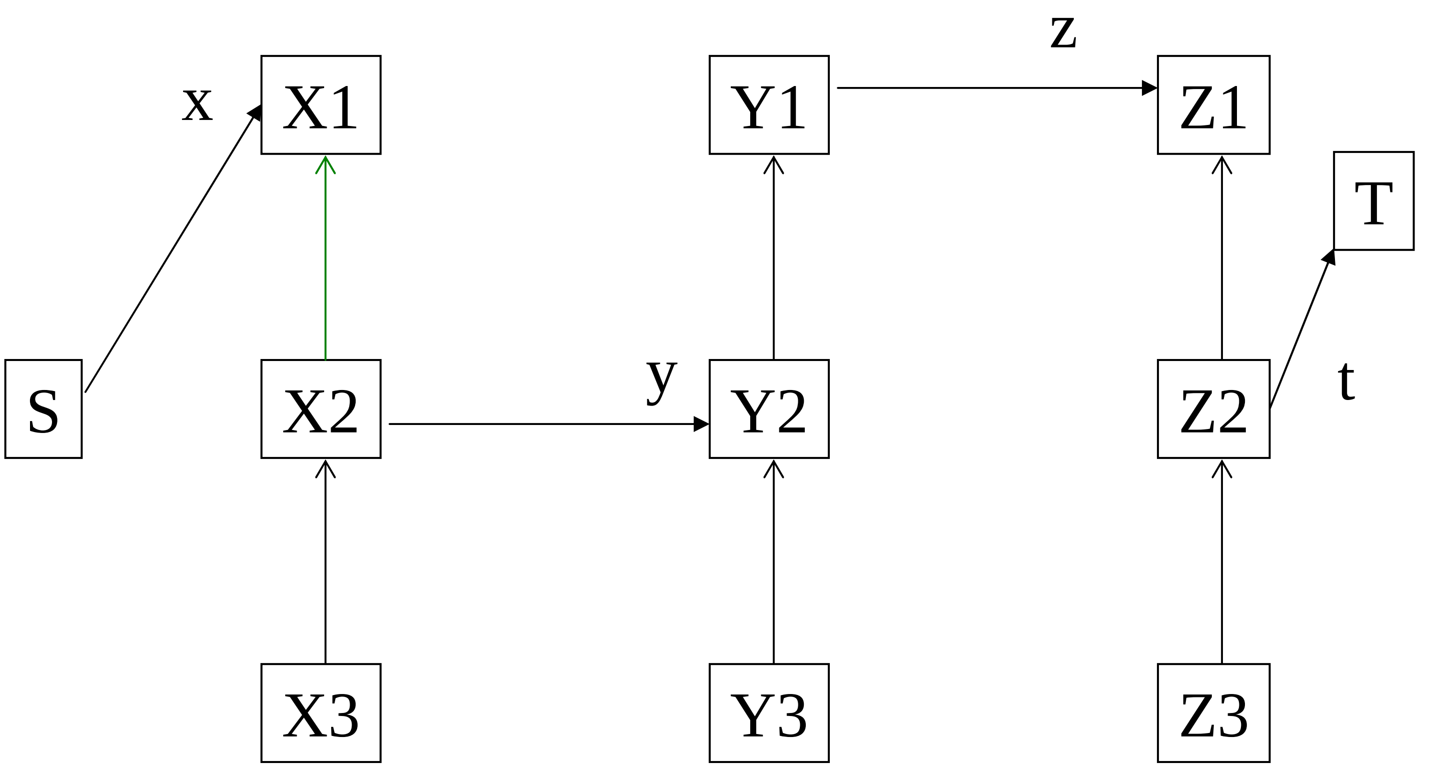


go down x iff **S** <=**.x.** =>**.(**<=**.C.**=>.<=.**C**.=>.<=.**C**=>**).**<= **T**

<=,=>
not
used

Just in terms of relations

Relations are sets of pairs, ordering is irrelevant.

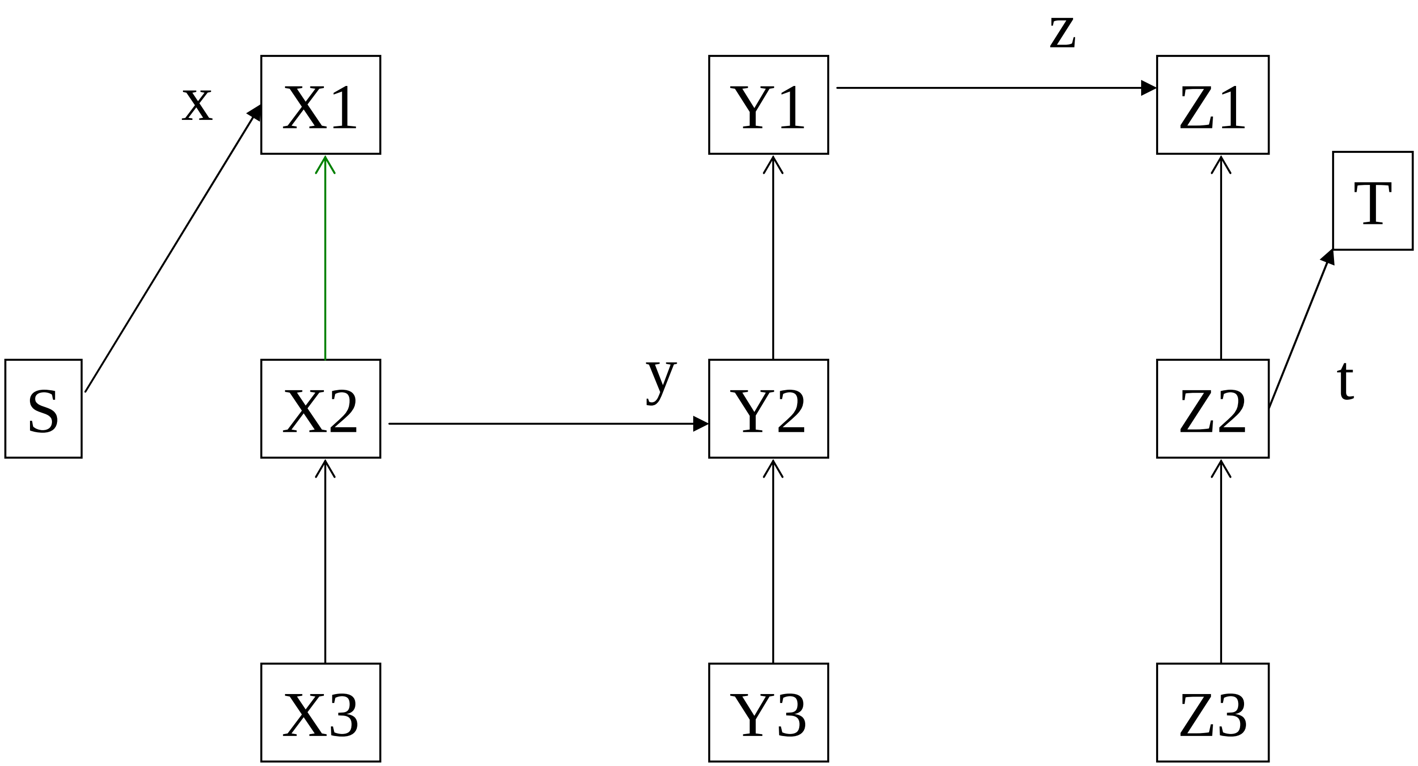


Set: {S,T,X1,X2,X3,Y1,Y2,Y3,Z1,Z2,Z3}

Write graph in terms of relations

Set: {S,T,X1,X2,X3,Y1,Y2,Y3,Z1,Z2,Z3}

Relations are sets of pairs, ordering is irrelevant.



Relations:
x,y,z,t,<=,=>
x(S,X1)
y(X2,Y2)
z(Y1,Z1)
t(Z2,T)
<=(Y2,Y1)
<=(X2,X1)
<=(X3,X2)
…
=>(X2,X3)
=>(X1,X2)
=>(Z1,Z2)
…

Set: {S,T,X1,X2,X3,Y1,Y2,Y3,Z1,Z2,Z3}

Compose relations

Relations are sets of pairs, ordering is irrelevant.

Relations: x,y,z,t,<=,=> *inheritance in reverse*

inheritance

x(S,X1)
y(X2,Y2)
z(Y1,Z1)
t(Z2,T)
<=(Y2,Y1)
<=(X2,X1)
<=(X3,X2)
…
=>(X2,X3)
=>(X1,X2)
=>(Z1,Z2)
…

Are S and T in the relation: **<=.x. =>.(<=.C.=><=.C.=>.<=.C=>).<=**

Set: {S,T,X1,X2,X3,Y1,Y2,Y3,Z1,Z2,Z3}

Just in terms of relations

Relations are sets of pairs, ordering is irrelevant.

Relations: x,y,z,t,<=,=>
x(S,X1)
y(X2,Y2)
z(Y1,Z1)
t(Z2,T)
<=(Y2,Y1)
<=(X2,X1)
<=(X3,X2)
…
=>(X2,X3)
=>(X1,X2)
=>(Z1,Z2)
…

The order in which we consume the pairs

Relations:
x(S,X1)
=>(X1,X2)
y(X2,Y2)
<=(Y2,Y1)
z(Y1,Z1)
=>(Z1,Z2)
t(Z2,T)

go down x iff **S <=.x. =>.(<=.C.=><=.C.=>.<=.C=>).<= T**

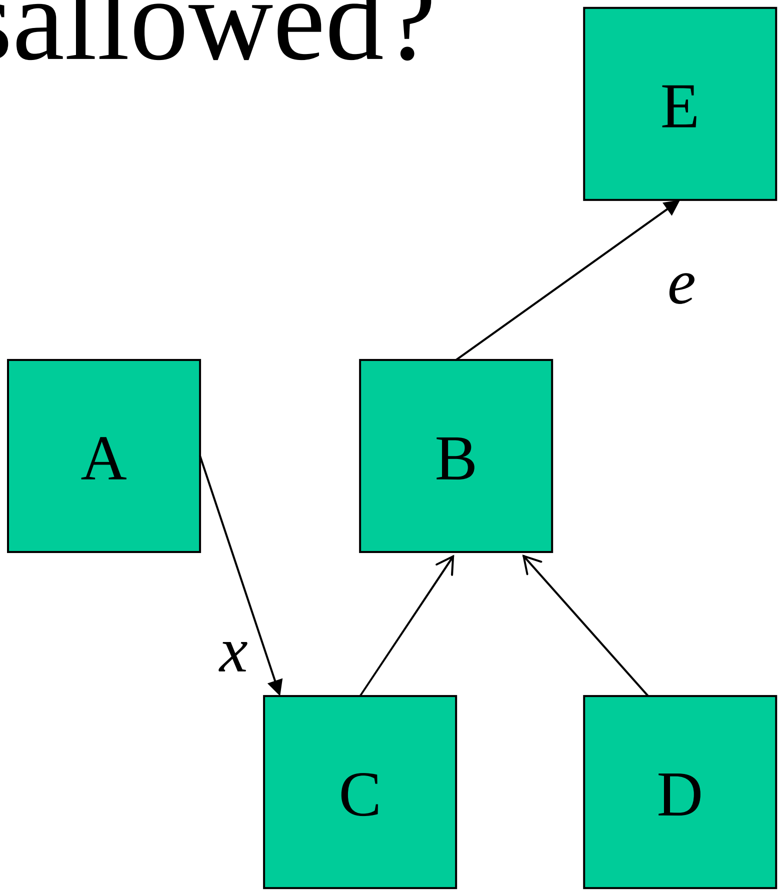
How big is the relation?

How many pairs does the relation contain in this example?

<=.x. =>.(<=.C.=><=.C.=>.<=.C=>).<=

What is disallowed?

Where can you go from A?



go down x iff S and T are in relation: <=.x. =>.(<=.C.=>)*.<=

class dictionary

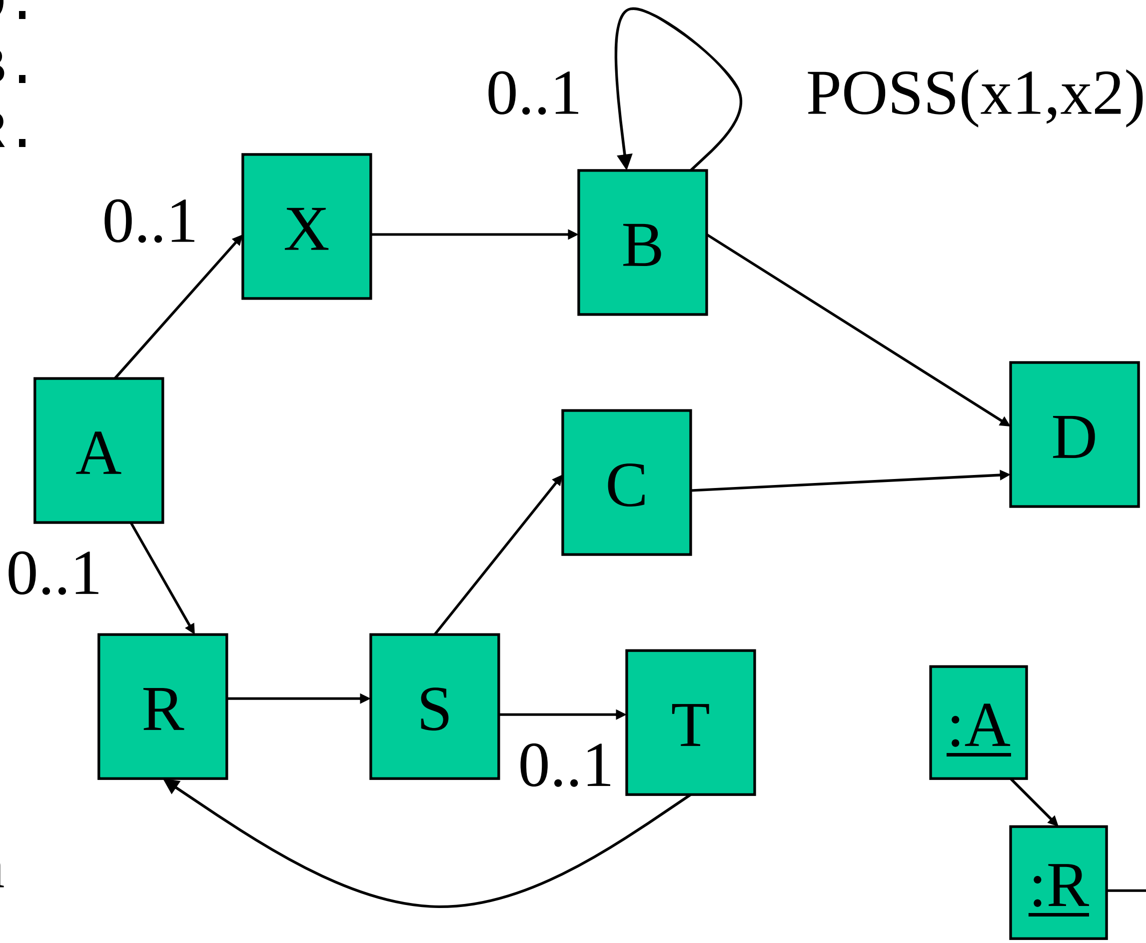
```
A = ["x" X] ["r" R].
B = ["b" B] D.
R = S.
S = ["t" T] C
C = D.
X = B.
T = R.
D = .
```

Example B

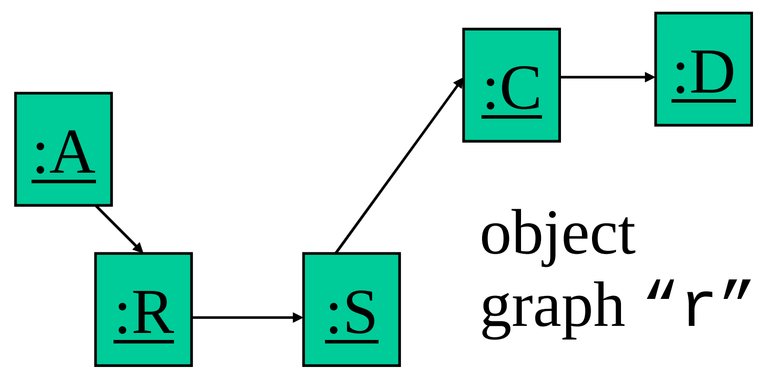
strategy

```
A -> T
T -> D
```

POSS(x1,x2)={e | x1 e.C* x2 }



class graph



object graph "r"

class dictionary

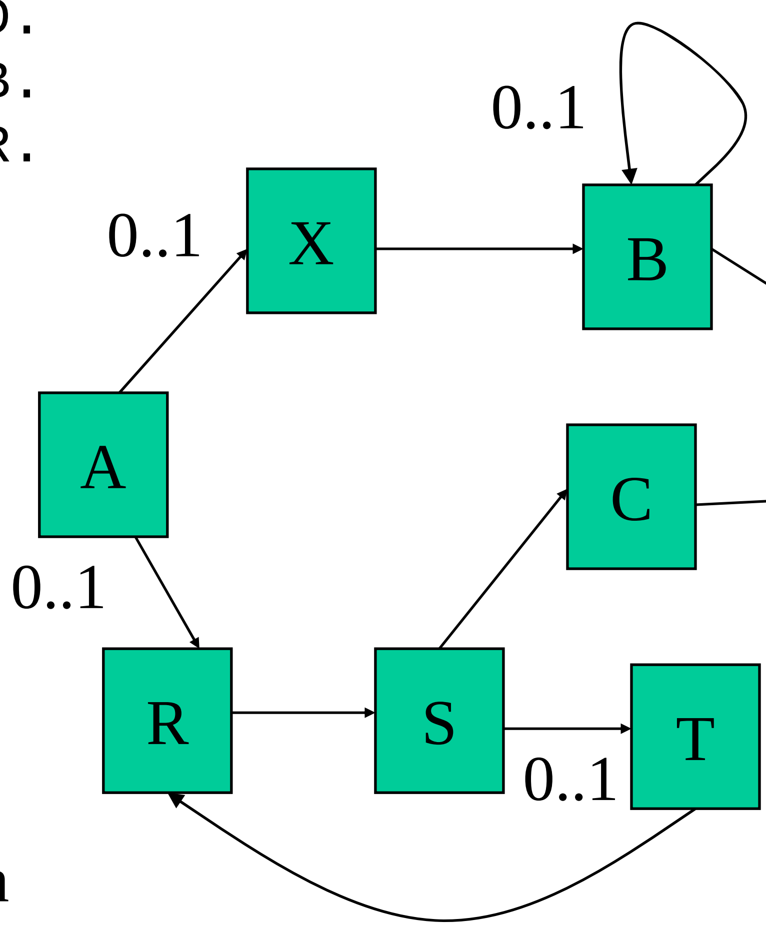
```
A = ["x" X] ["r" R].
B = ["b" B] D.
R = S.
S = ["t" T] C
C = D.
X = B.
T = R.
D = .
```

Example B1

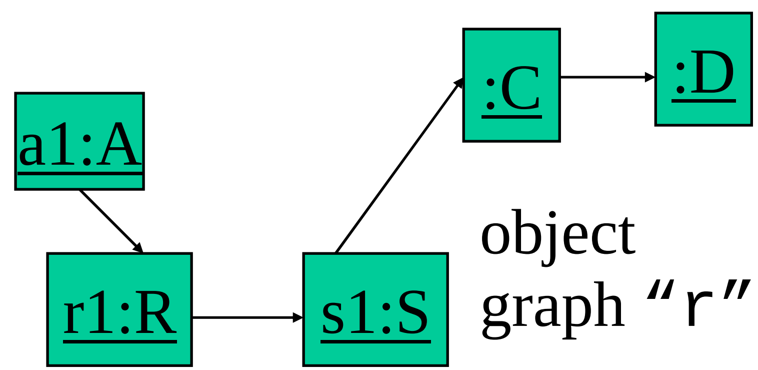
strategy

```
A -> T
T -> D
```

POSS(A,T) = 1 edge
POSS(R,T) = 1 edge
POSS(S,T) = 0 edges



class graph



object graph "r"

POSS(c1,c2)={e | c1 e.C* c2 }

Example B1

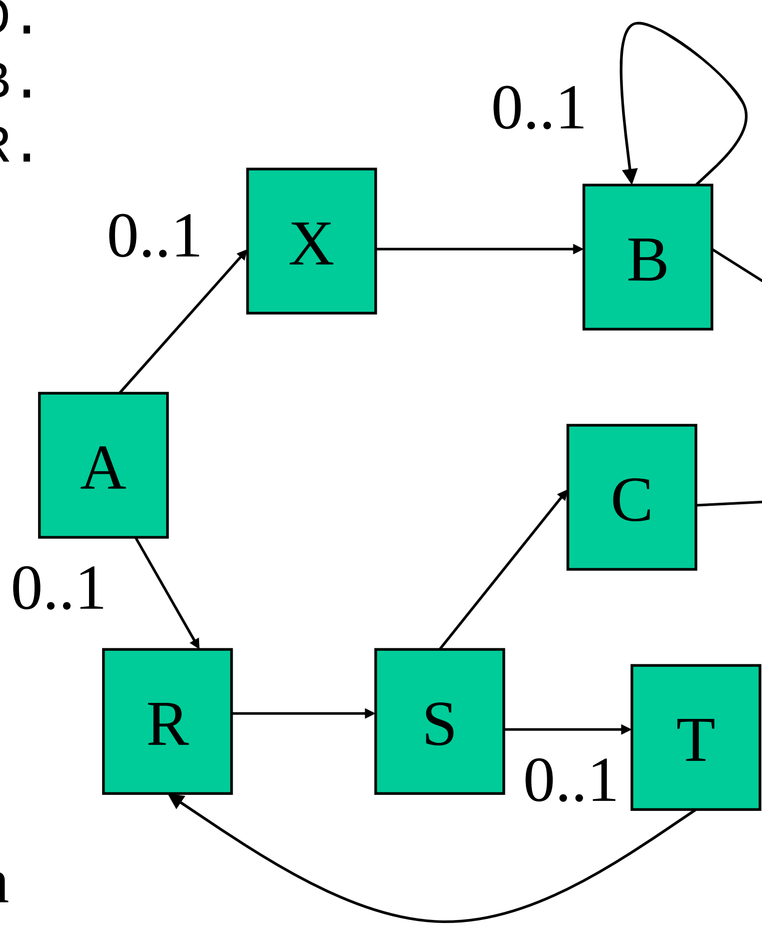
class dictionary

```
A = ["x" X] ["r" R].
B = ["b" B] D.
R = S.
S = ["t" T] C
C = D.
X = B.
T = R.
D = .
```

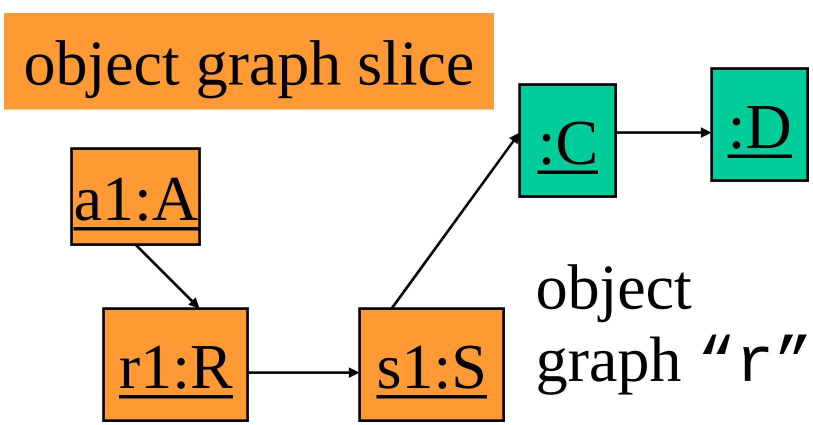
strategy

```
A -> T
T -> D
```

POSS(A,T) = 1 edge
POSS(R,T) = 1 edge
POSS(S,T) = 0 edges



class graph



object graph "r"

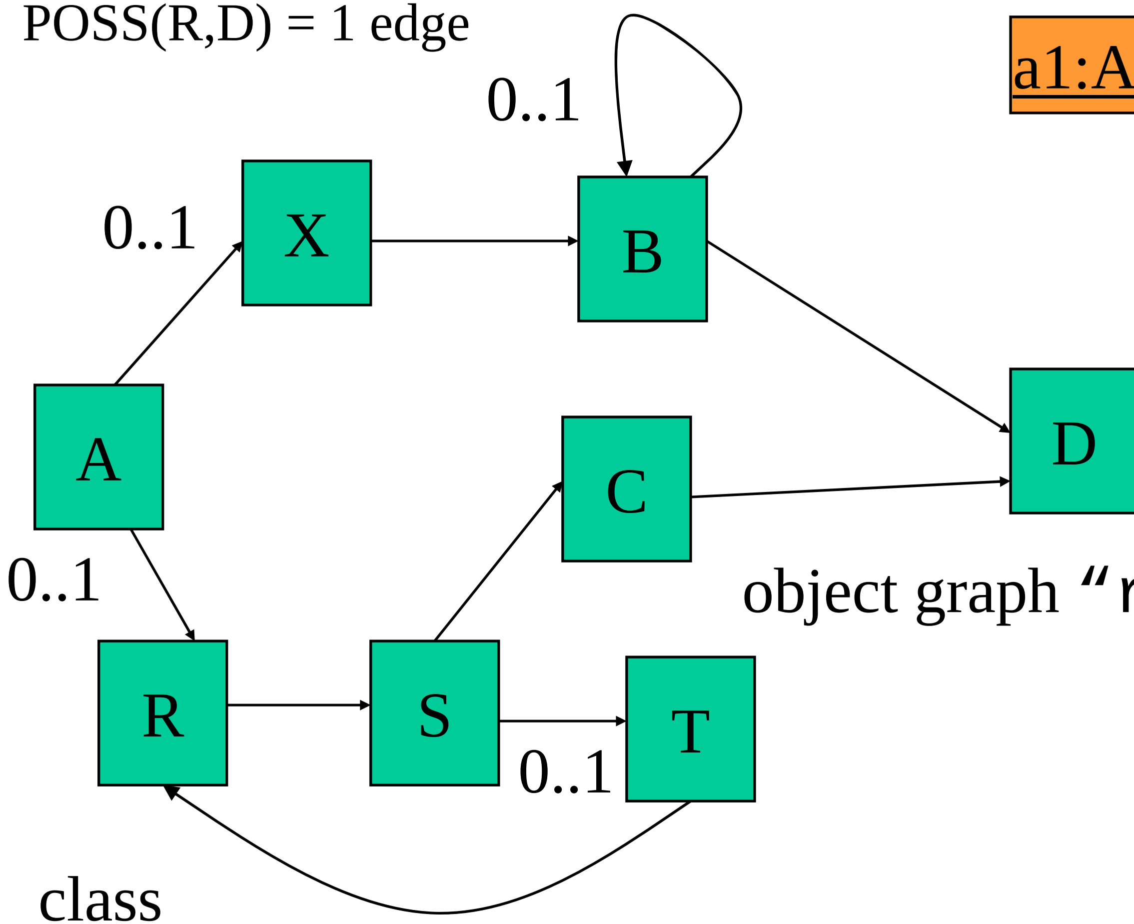
POSS(c1,c2)={e | c1 e C* c2 }

POSS(A,T) = 1 edge
POSS(R,T) = 1 edge
POSS(S,T) = 1 edge
POSS(T,D) = 1 edge
POSS(R,D) = 1 edge

Example B2

strategy

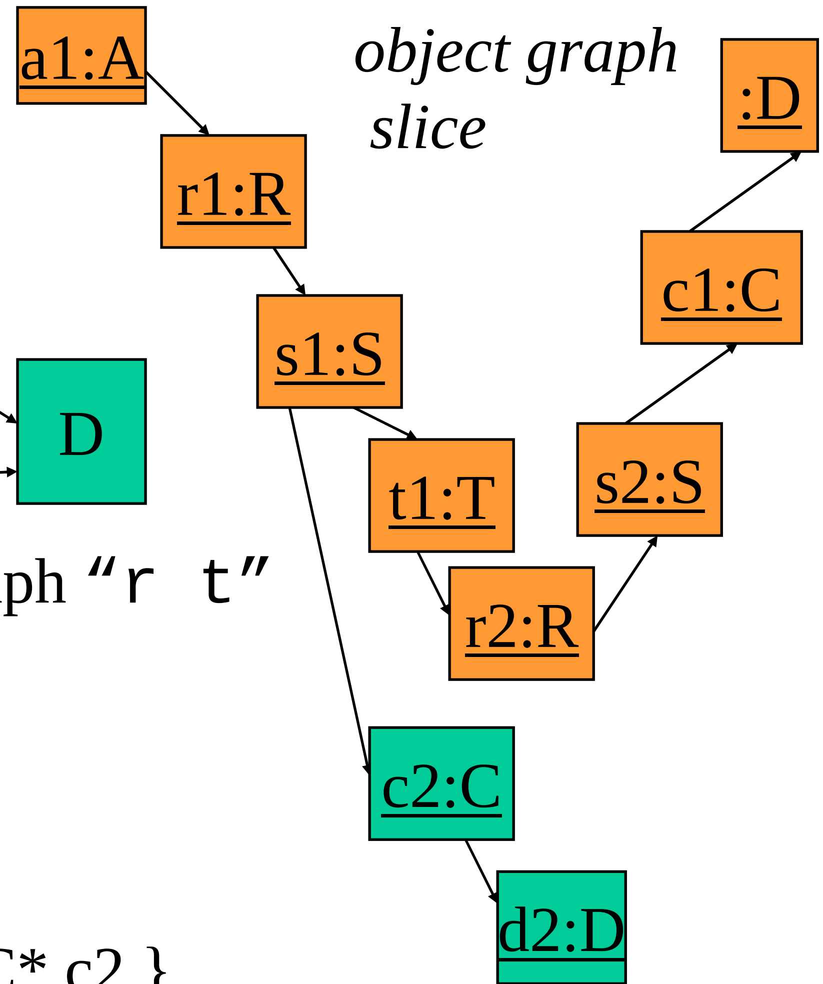
```
A -> T
T -> D
```



object graph "r t"

class graph

object graph slice



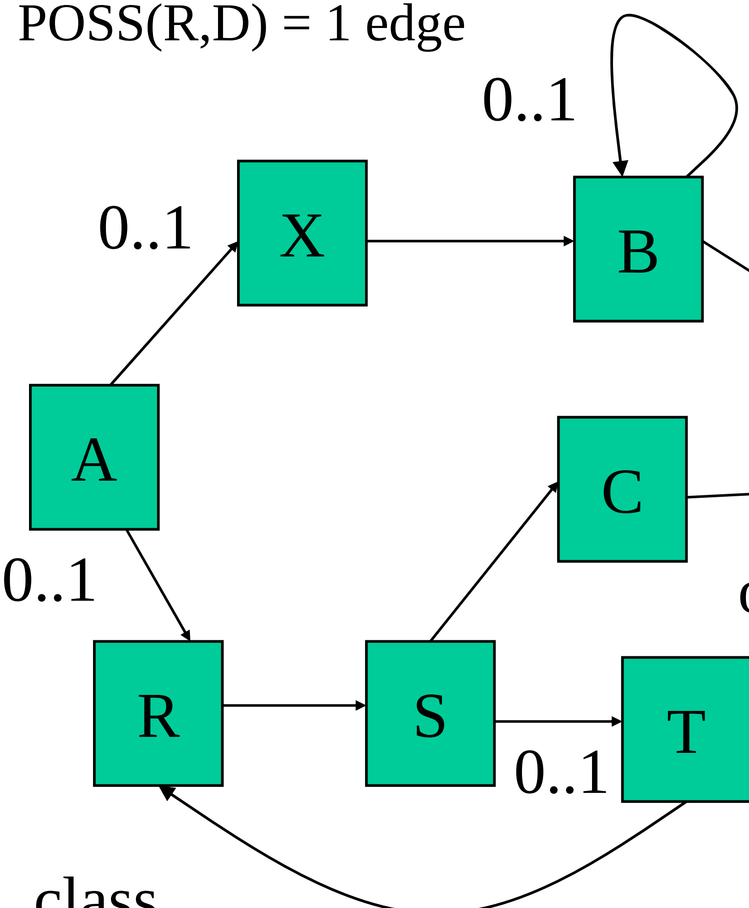
POSS(c1,c2)={e | c1 e.C* c2 }

POSS(A,T) = 1 edge
POSS(R,T) = 1 edge
POSS(S,T) = 1 edge
POSS(T,D) = 1 edge
POSS(R,D) = 1 edge

Example B2

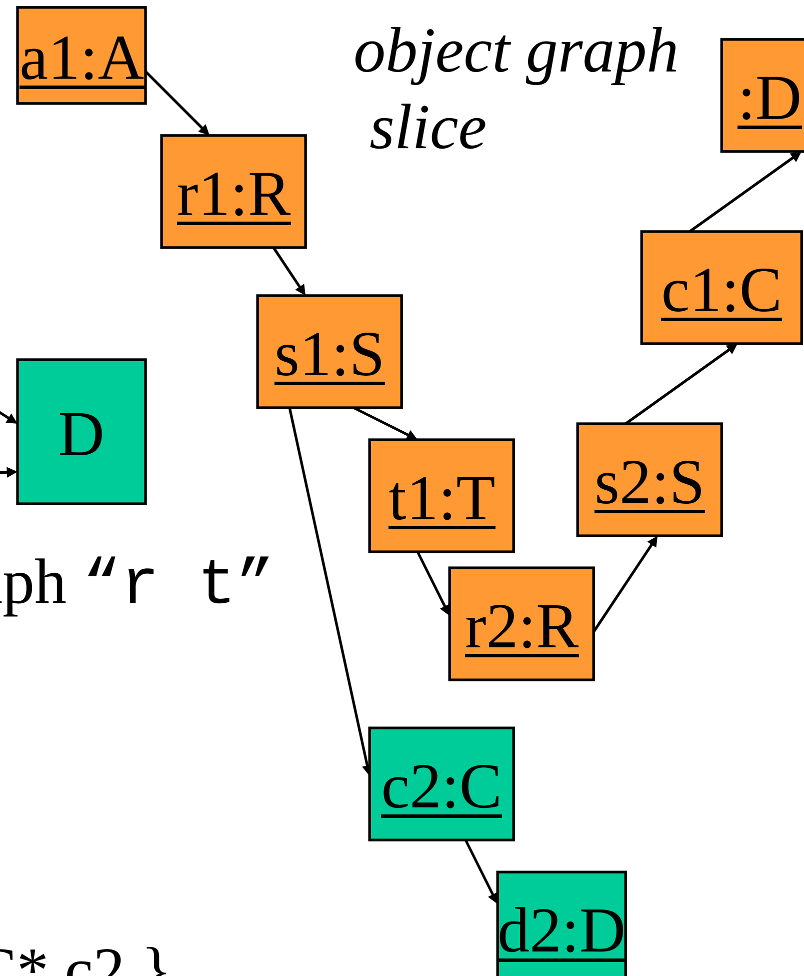
strategy

```
A -> T
T -> D
```



class graph

object graph "r t"



object graph slice

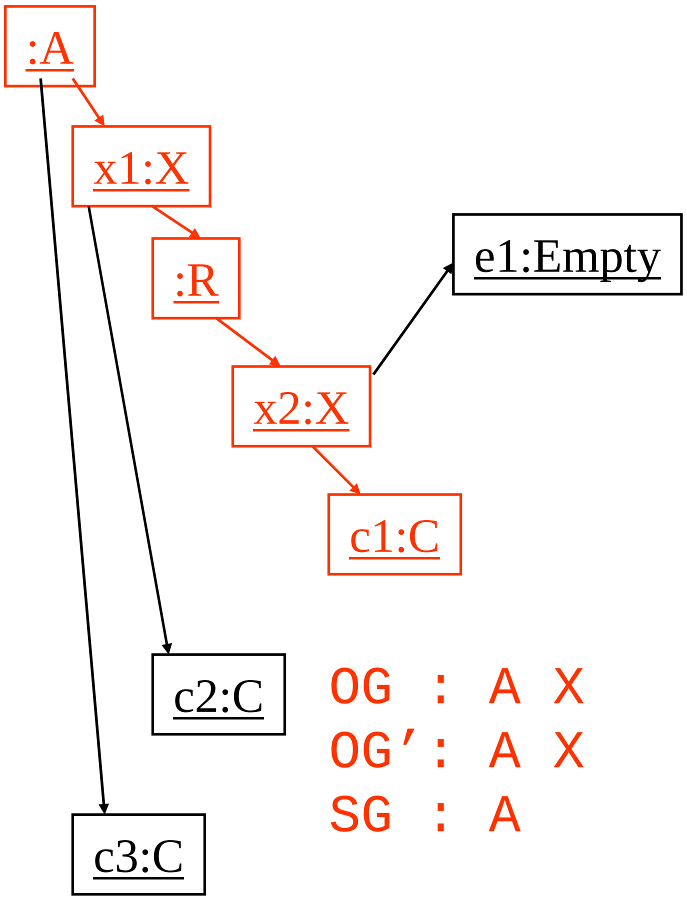
POSS(c1,c2)={e | c1 e.C* c2 }

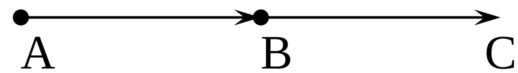
Only node paths shown for space reasons

Example C

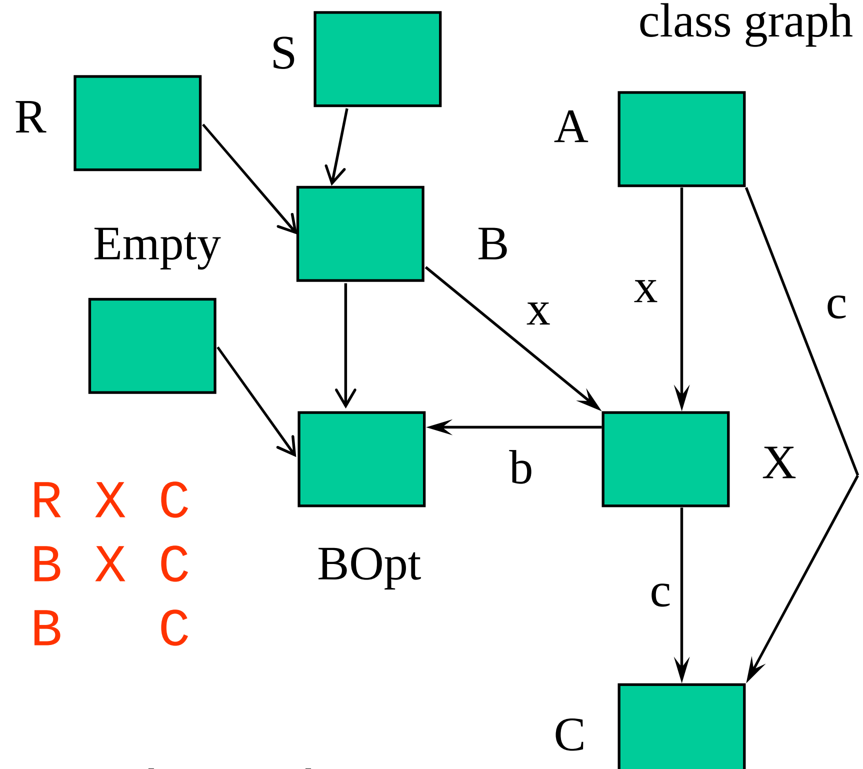
```
strategy SG:
{A -> B
 B -> C}
```

Object graph






```
OG : A X   R X C
OG': A X   B X C
SG : A     B   C
```

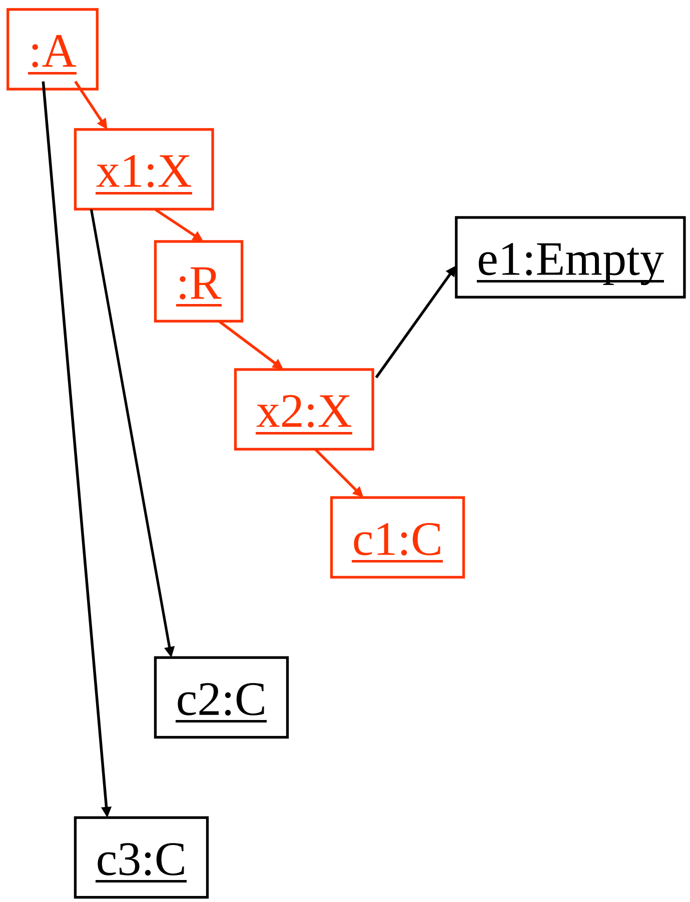


Write Java code that does the traversal

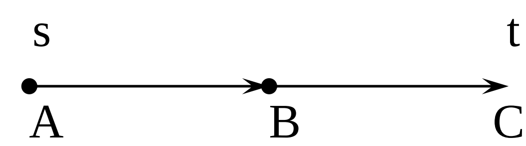
Example C

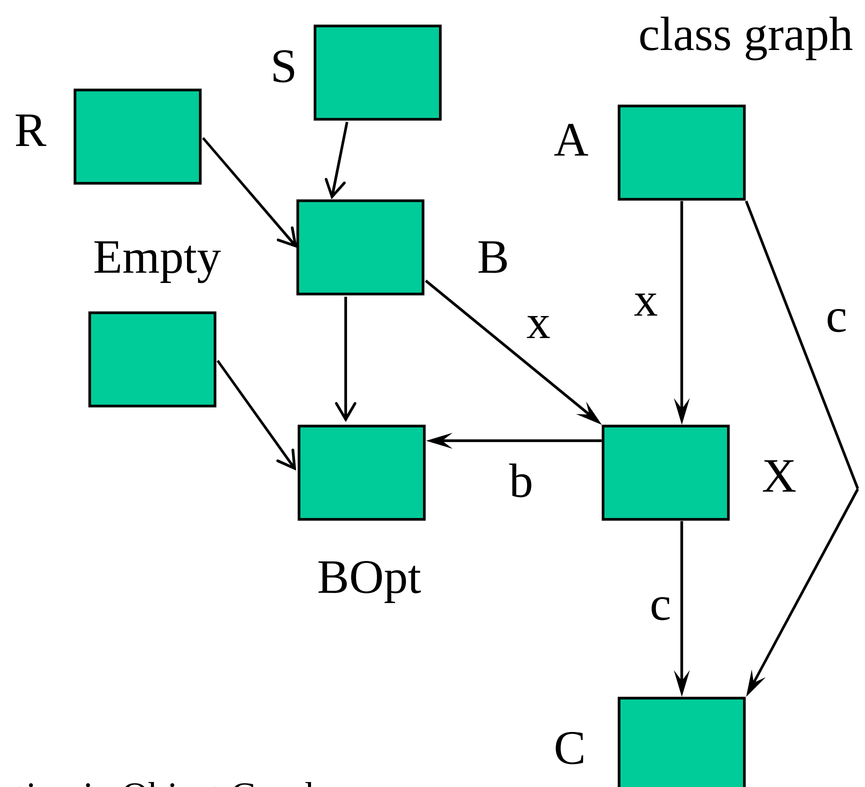
```
strategy:
{A -> B
 B -> C}
```

Object graph



Strategy

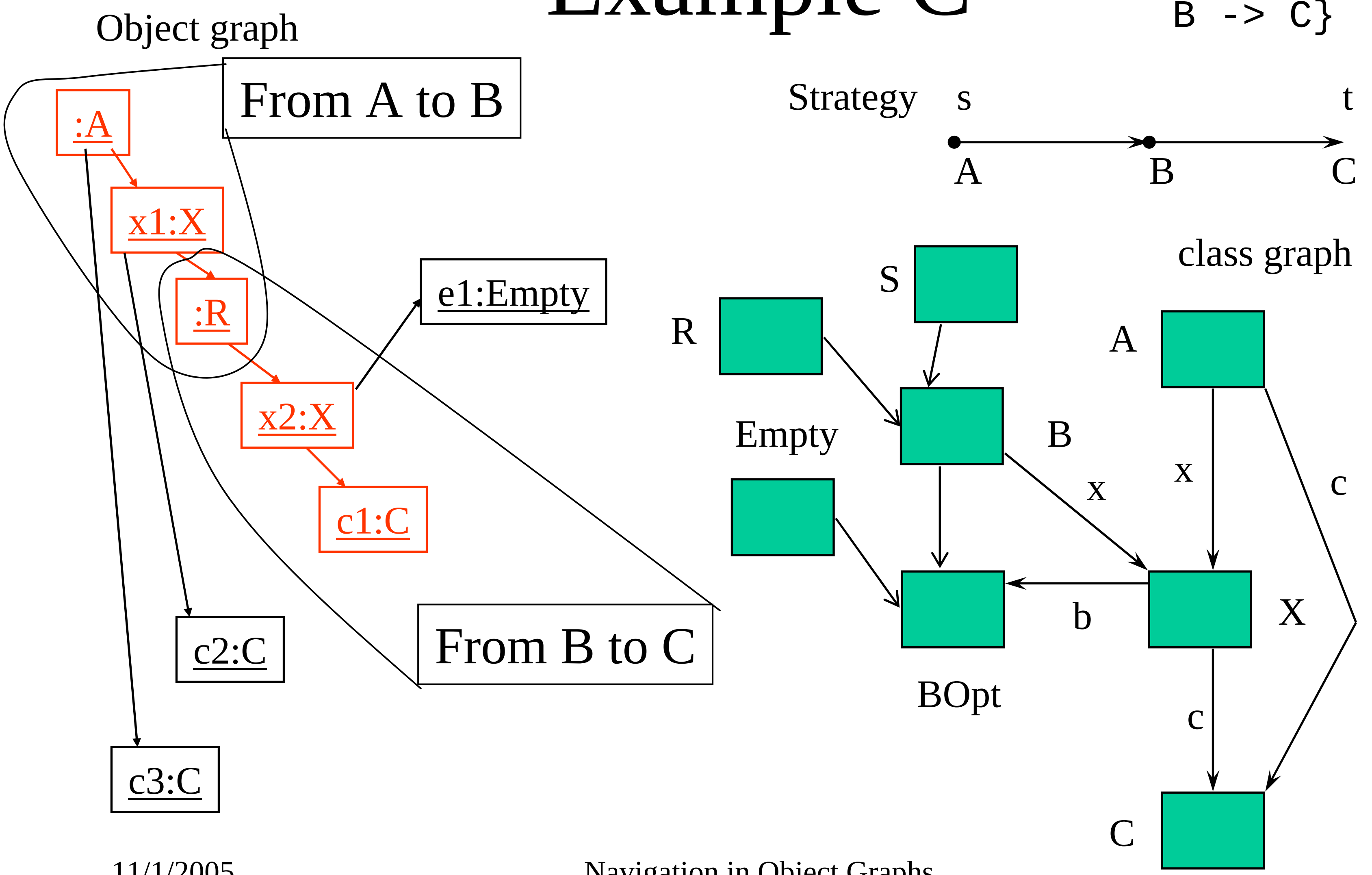




Not the complete story

Example C

```
strategy:
{A -> B
 B -> C}
```

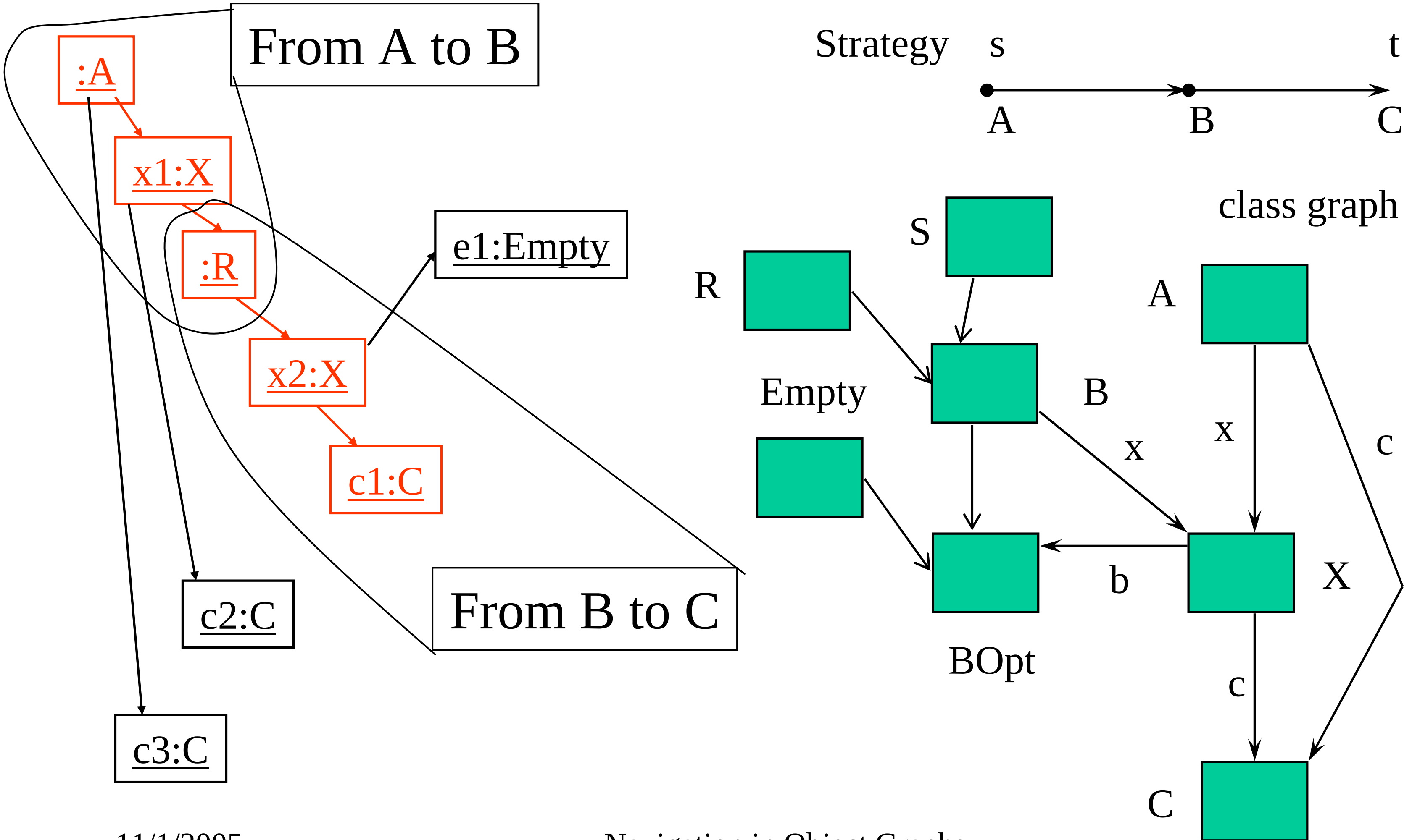


Not the complete story: traversal must look for further B

Example C

```
strategy:
{A -> B
 B -> C}
```

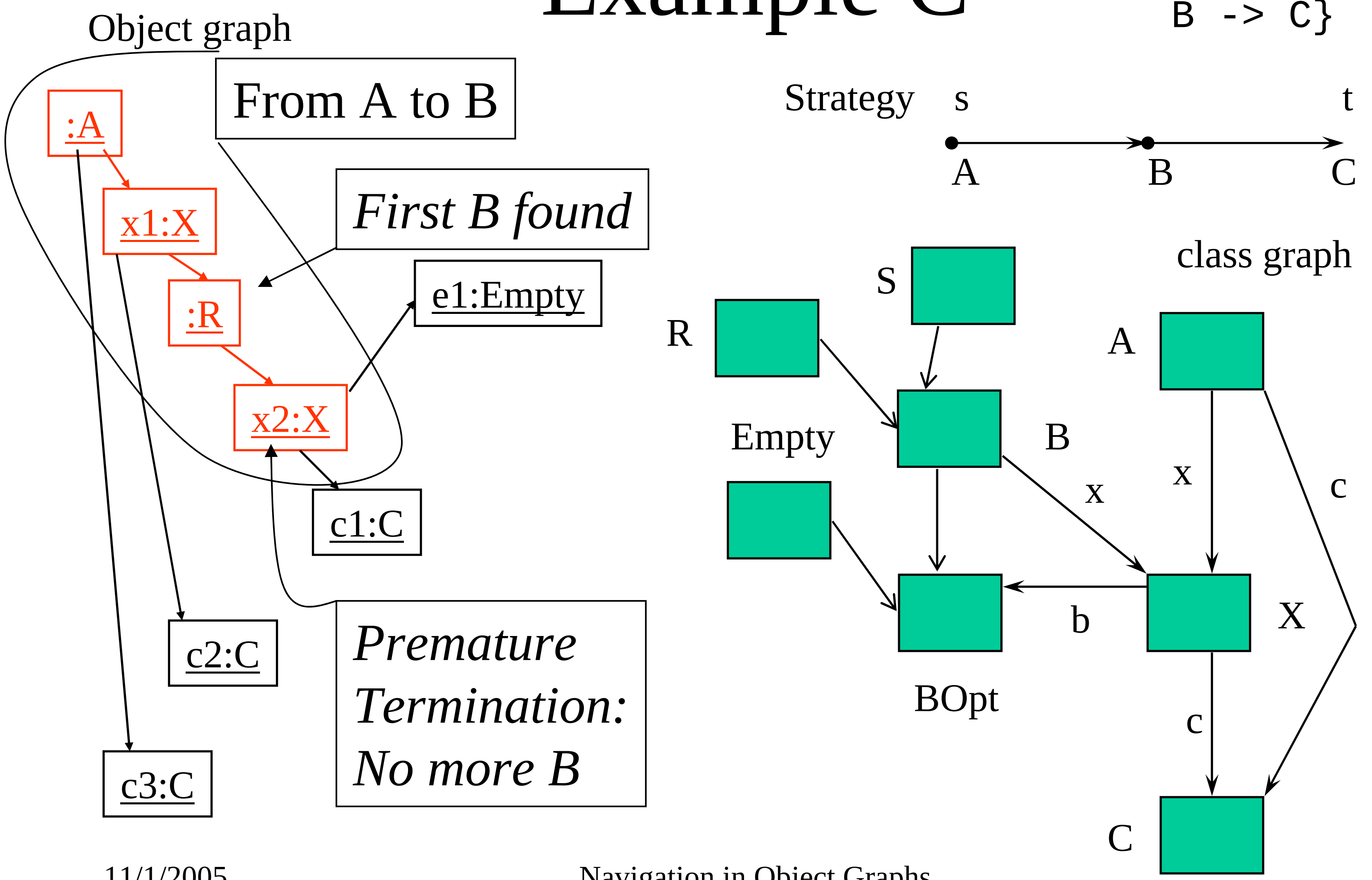
Object graph



Not the complete story: traversal must look for further B

Example C

```
strategy:
{A -> B
 B -> C}
```

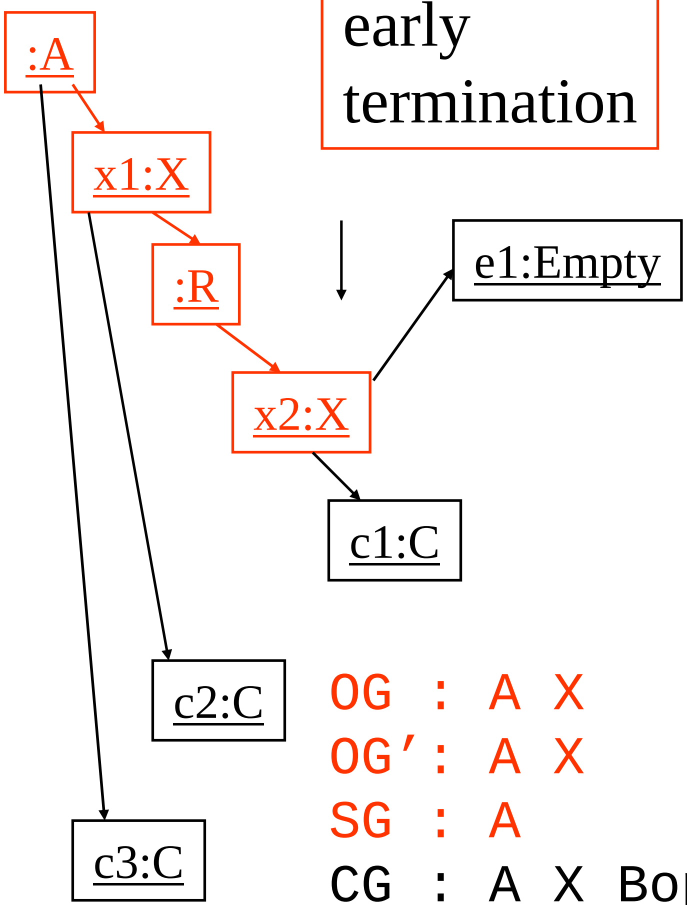


Only node paths shown for space reasons

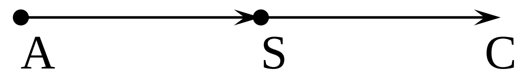
Example C1

```
strategy SG:
{A -> S
 S -> C}
```

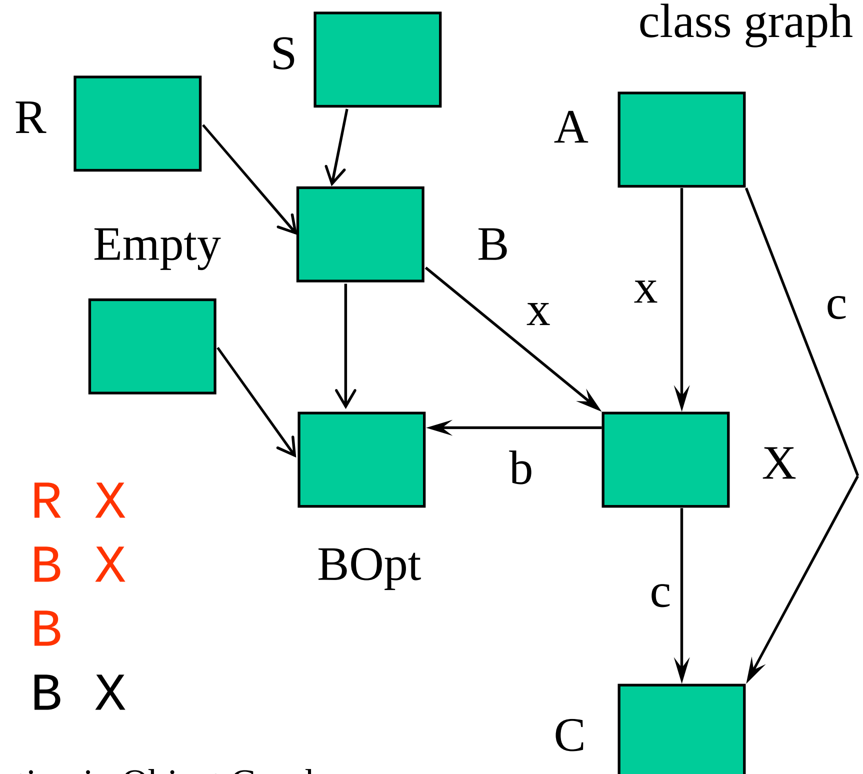
Object graph



Strategy



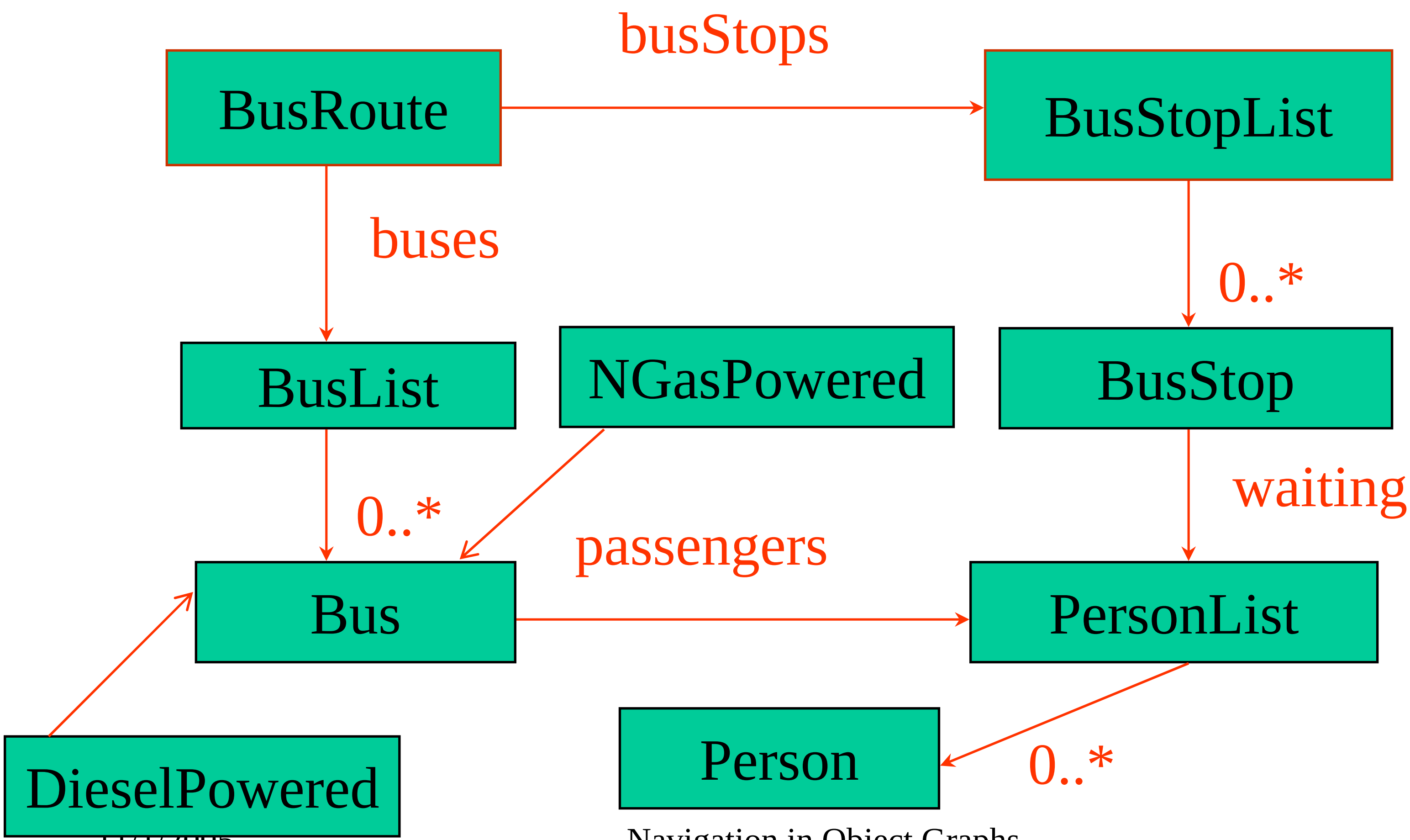
class graph




```
OG : A X        R X
OG': A X        B X
SG : A          B
CG : A X Bopt B X
```

`S = from` BusRoute `through` Bus `to` Person

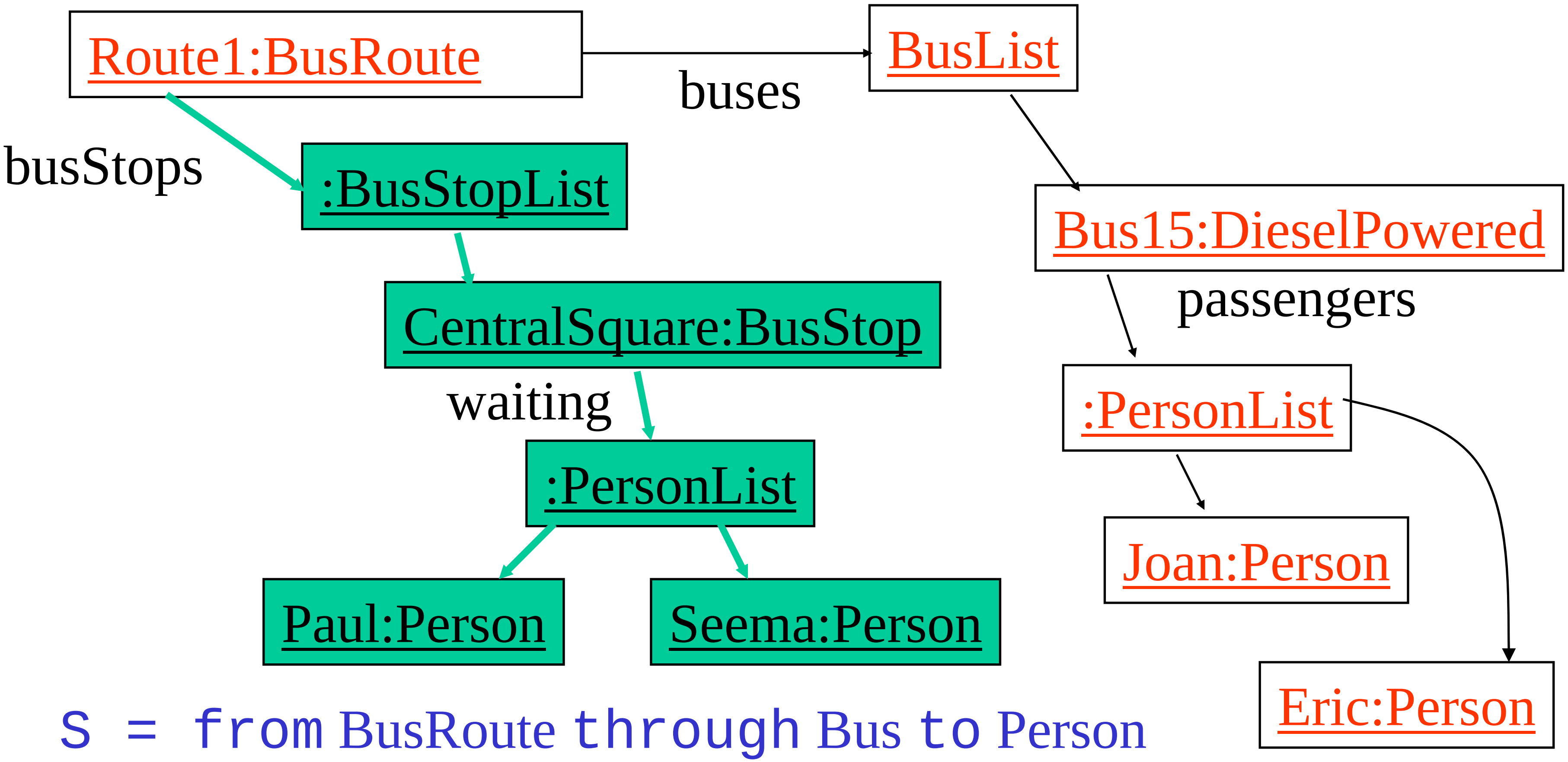
Example D



```
OG : BR BL DP PL P
OG': BR BL B  PL P
S  : BR    B     P
```

Example D1

Only node paths shown for space reasons

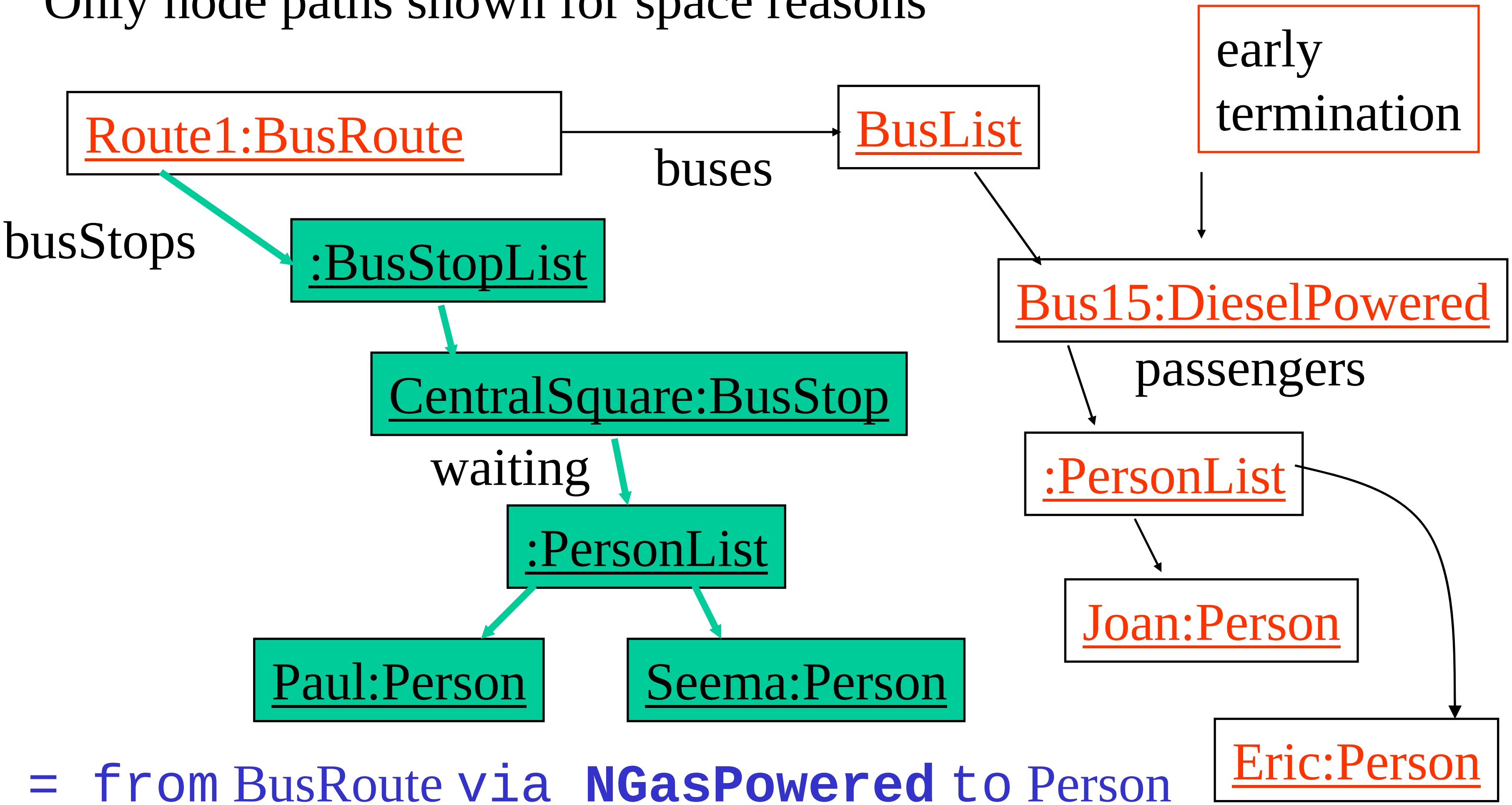


S = from BusRoute through Bus to Person

```
OG : BR BL
OG': BR BL
S  : BR
```

Example D2

Only node paths shown for space reasons



`S = from` BusRoute `via` **`NGasPowered`** `to` Person from c1 bypassing {x1,x2, … ,xn} to c2

Relational Formulation

From object o of class c1, to get to c2, follow edges in the set

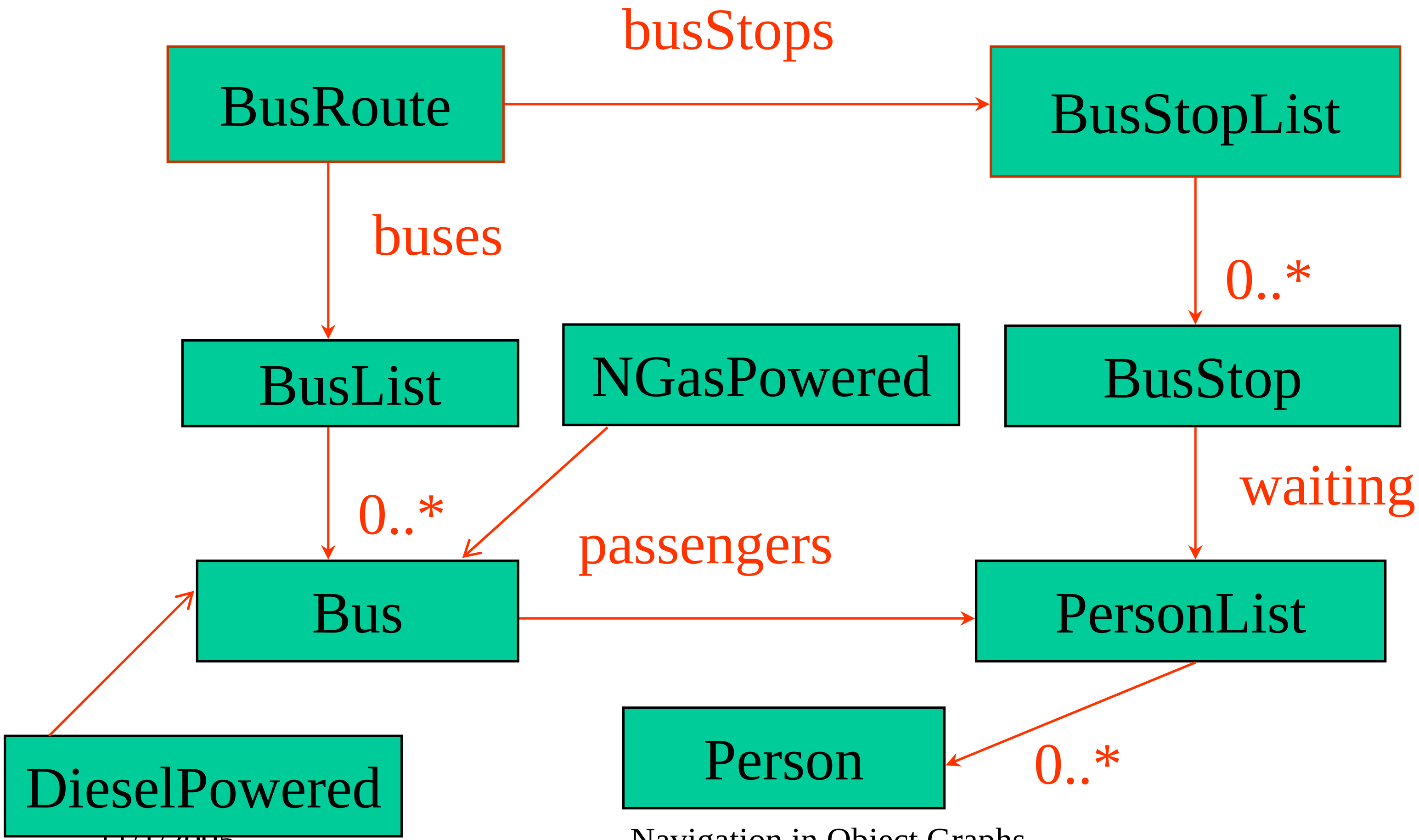
POSS(c1,c2)={e | c1 <=.e.=> (<=.C.=>)* <= c2 }

POSS = abbreviation for: following these edges it is still possible to reach a c2-object for some c1-object rooted at o.

Delete x1,x2, … ,xn and all edges incident with these nodes from the class graph (assumed to be different from c1, c2).

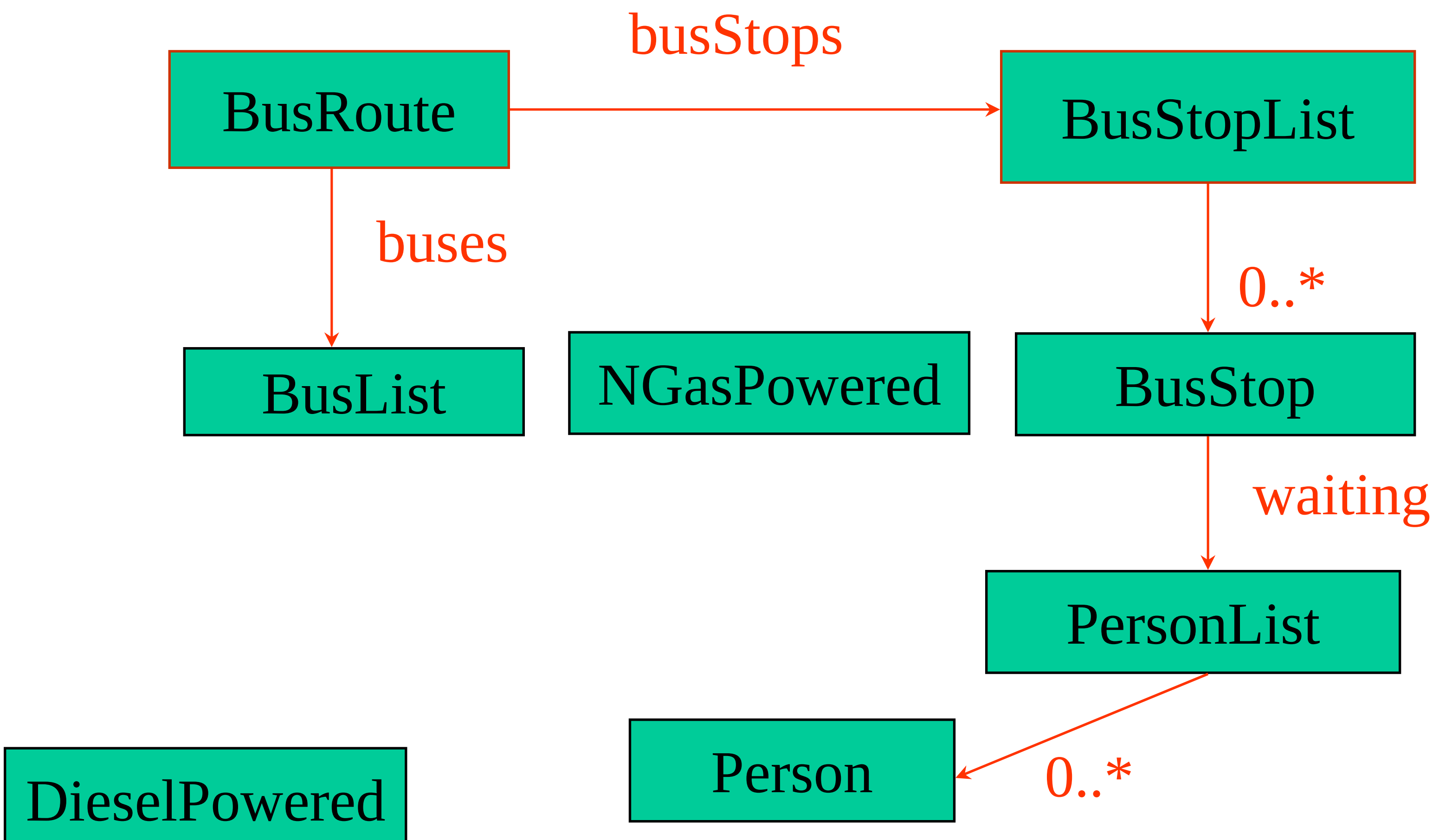
`S = from` BusRoute `bybassing` Bus `to` Person

Example D



`S = from` BusRoute `bypassing` Bus `to` Person

Example D



Conclusions

- Programming language elements are mathematical objects having precise mathematical definitions.
- Exercise in applying an abstract algorithm to concrete inputs. Mapping abstract situations to concrete situations.
- Separation of concerns is also useful for defining programming language elements
  - separate subgraph selected from
  - how the subgraph is traversed (depth-first etc.)
- In earlier works: meaning of a traversal strategy for an object graph
  - was a traversal history
  - now it is a subgraph of the object graph. A traversal history can be defined ...