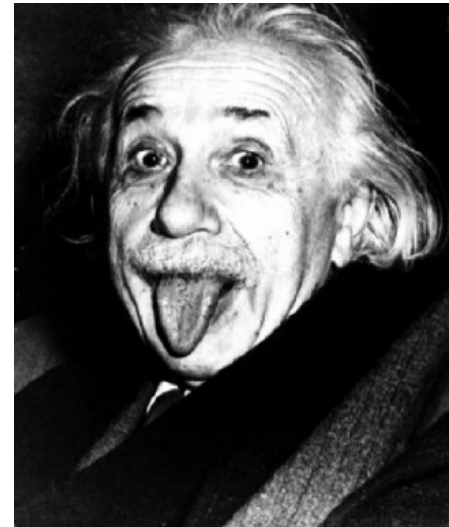


FROM:
FM 2006 Alloy Tutorial

Language and Analysis

Greg Dennis and Rob Seater
Software Design Group, MIT



alloy language & analysis

- language = syntax for structuring specifications in logic
 - shorthands, sugar
- analysis = tool for finding solutions to logical formulas
 - searches for and visualizes counterexamples



“I'm My Own Grandpa” in Alloy

```
module examples/tutorial/grandpa

abstract sig Person {
  father: lone Man,
  mother: lone Woman
}

sig Man extends Person {
  wife: lone Woman
}

sig Woman extends Person {
  husband: lone Man
}

fact {
  no p: Person |
    p in p.^(mother + father)
  wife = ~husband
}
```

```
assert noSelfFather {
  no m: Man | m = m.father
}

check noSelfFather

fun grandpas(p: Person) : set Person {
  p.(mother + father).father
}

pred ownGrandpa(p: Person) {
  p in grandpas(p)
}

run ownGrandpa for 4 Person
```

language: signatures - classes?

sig A {}
set of atoms A

sig A {}
sig B {}
disjoint sets A and B (no A & B)

sig B **extends** A {}
set B is a subset of A (B in A)

sig B **extends** A {}
sig C **extends** A {}
*B and C are disjoint subsets of A
(B in A && C in A && no B & C)*

abstract sig A {}
sig B **extends** A {}
sig C **extends** A {}
*A partitioned by disjoint subsets B and C
(no B & C && A = (B + C))*

sig B **in** A {}
*B is a subset of A – not necessarily
disjoint from any other set*

one sig A {}
lone sig B {}
some sig C {}
*A is a singleton set
B is a singleton or empty
C is a non-empty set*

grandpa: signatures

```
abstract sig Person {  
  . . .  
}  
  
sig Man extends Person {  
  . . .  
}  
  
sig Woman extends Person {  
  . . .  
}
```

- all men and women are persons
- no person is both a man and a woman
- all persons are either men or women

language: fields

sig A {f: e}

f is a binary relation with domain *A*
and range given by expression *e*
f is constrained to be a function
(all *a*: A | *a.f*: e)

sig A {

f1: **one** e1,

f2: **lone** e2,

f3: **some** e3,

f4: **set** e4

}

(all *a*: A | *a.fn* : m e)

sig A {f, g: e}

two fields with same constraints

sig A {f: e1 m -> n e2}

*map each member of e1 to n members of e2
and map m members of e1 to each members of
e2*

(all *a*: A | *a.f*: e1 m -> n e2)

sig Book {

names: **set** Name,

addrs: names -> Addr

}

dependent fields

(all *b*: Book | *b.addrs*: *b.names* -> Addr)

grandpa: fields

```
abstract sig Person {  
  father: lone Man,  
  mother: lone Woman  
}  
  
sig Man extends Person {  
  wife: lone Woman  
}  
  
sig Woman extends Person {  
  husband: lone Man  
}
```

- fathers are men and everyone has at most one
- mothers are women and everyone has at most one
- wives are women and every man has at most one
- husbands are men and every woman has at most one

language: facts

```
fact { F }  
fact f { F }  
sig S { ... }{ F }
```

*facts introduce constraints that
are assumed to **always** hold*

```
sig Host {}  
sig Link {from, to: Host}  
  
fact {all x: Link | x.from != x.to}  
no links from a host to itself  
  
fact noSelfLinks {all x: Link | x.from != x.to}  
same as above
```

grandpa: fact

```
fact {  
  no p: Person |  
    p in p.^(mother + father)  
  wife = ~husband  
}
```

- no person is his or her own ancestor
- a man's wife has that man as a husband
- a woman's husband has that woman as a wife

language: functions

```
fun f(x1: e1, ..., xn: en) : e { E }
```

functions are named expression with declaration parameters and a declaration expression as a result invoked by providing an expression for each parameter

```
sig Name, Addr {}  
sig Book {  
  addr: Name -> Addr  
}  
  
fun lookup(b: Book, n: Name) : set Addr {  
  b.addr[n]  
}  
  
fact everyNameMapped {  
  all b: Book, n: Name | some lookup(b, n)  
}
```

language: predicates

```
pred p(x1: e1, ..., xn: en) { F }
```

named formula with declaration parameters

```
sig Name, Addr {}  
sig Book {  
  addr: Name -> Addr  
}  
  
pred contains(b: Book, n: Name, d: Addr) {  
  n->d in b.addr  
}  
  
fact everyNameMapped {  
  all b: Book, n: Name |  
    some d: Addr | contains(b, n, a)  
}
```

grandpa: function and predicate

```
fun grandpas (p: Person) : set Person {  
  p.(mother + father).father  
}  
  
pred ownGrandpa (p: Person) {  
  p in grandpas (p)  
}
```

- a person's grandpas are the fathers of one's own mother and father

language: assertions

```
assert a { F }
```

*constraint intended to follow
from facts of the model*



```
sig Node {  
  children: set Node  
}  
  
one sig Root extends Node {}  
  
fact {  
  Node in Root.*children  
}  
  
// invalid assertion:  
assert someParent {  
  all n: Node | some children.n  
}  
  
// valid assertion:  
assert someParent {  
  all n: Node - Root | some children.n  
}
```

language: check command

```
assert a { F }  
check a scope
```

*instructs analyzer to search for
counterexample to assertion within scope*

*if model has facts M
finds solution to M && !F*

```
check a  
top-level sigs bound by 3
```

```
check a for default  
top-level sigs bound by default
```

```
check a for default but list  
default overridden by bounds in list
```

```
abstract sig Person {}  
sig Man extends Person {}  
sig Woman extends Person {}  
sig Grandpa extends Man {}
```

```
check a  
check a for 4  
check a for 4 but 3 Woman  
check a for 4 but 3 Man, 5 Woman  
check a for 4 Person  
check a for 4 Person, 3 Woman  
check a for 3 Man, 4 Woman  
check a for 3 Man, 4 Woman, 2 Grandpa
```

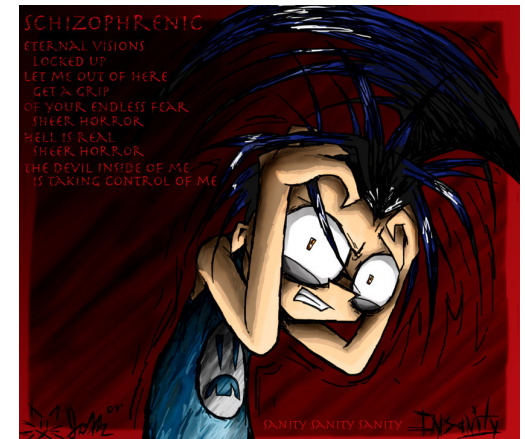
grandpa: assertion check

```
fact {
  no p: Person | p in p.^(mother + father)
  wife = ~husband
}

assert noSelfFather {
  no m: Man | m = m.father
}

check noSelfFather
```

- command instructs analyzer to search for counterexample to *noSelfFather* within a scope of at most 3 *Persons*
- *noSelfFather* assertion follows from fact



language: run command

```
pred p(x: X, y: Y, ...) { F }  
run p scope
```

*instructs analyzer to search for
instance of predicate within scope*

*if model has facts M, finds solution to
M && (some x: X, y: Y, ... | F)*



```
fun f(x: X, y: Y, ...) : R { E }  
run f scope
```

*instructs analyzer to search for
instance of function within scope*

*if model has facts M, finds solution to
M && (some x: X, y: Y, ..., result: R | result = E)*

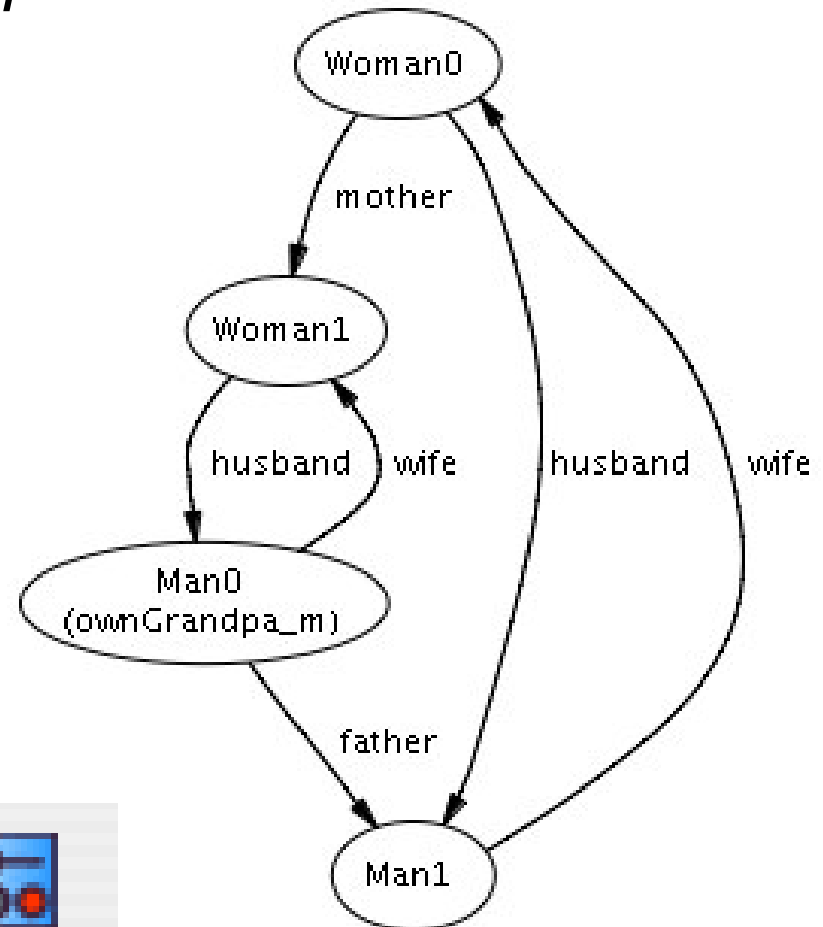
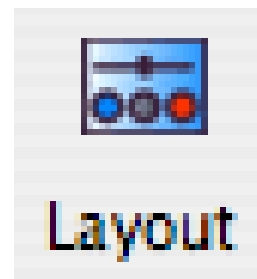
grandpa: predicate simulation

```
fun grandpas(p: Person) : set Person {  
  p.(mother + father).father  
}  
  
pred ownGrandpa(p: Person) {  
  p in grandpas(p)  
}  
  
run ownGrandpa for 4 Person
```

- command instructs analyzer to search for configuration with at most 4 Person in which a man is his own grandfather

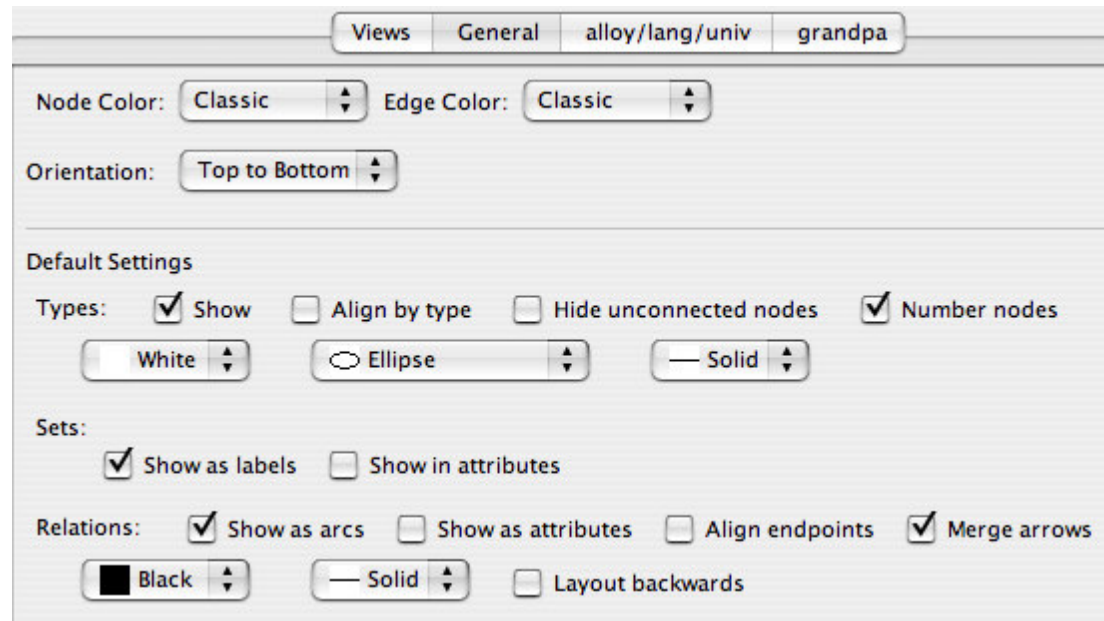
introduction to visualization

- open *examples/tutorial/grandpa.als*
- build-execute
- select “layout”



visualization layout pane

- the tabs
 - palette = set of views (lightweight)
 - general (default for inheritance)
 - univ (ignore)
 - modules (for us, just “grandpa”)



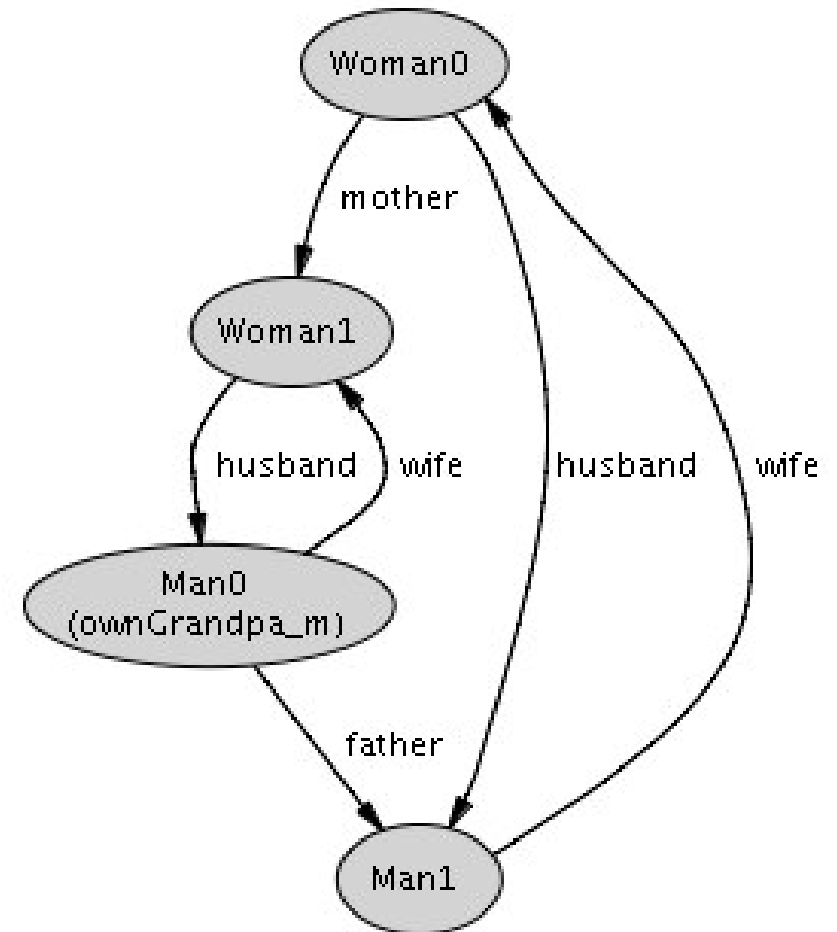
superficial

- general tab
 - default type color → gray



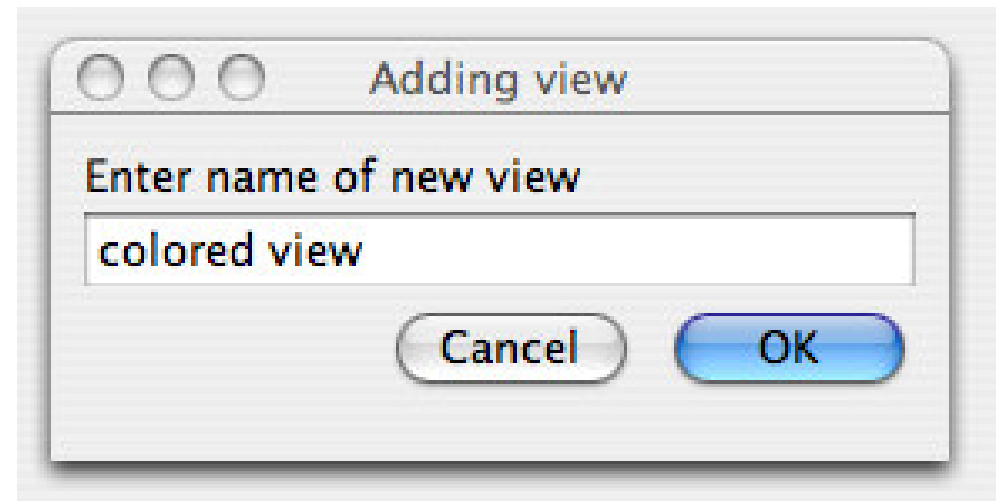
- *update*

- also notice:
 - hide unconnected nodes
 - orientation
 - layout backwards



another view . . .

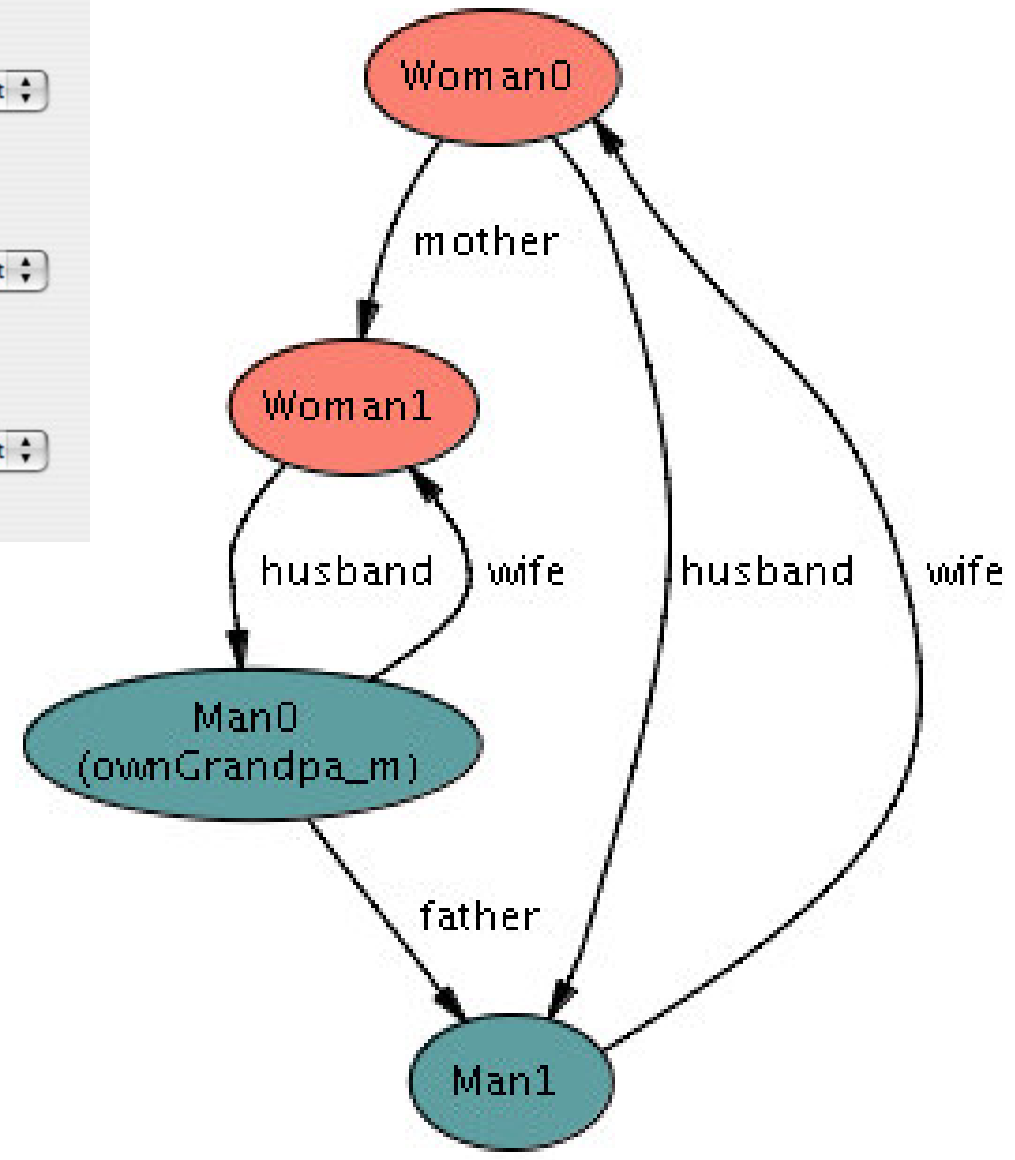
- views tab
 - save palette
 - new view “colored view”
 - automatically selected (but not applied)
- general tab
 - node color → martha
- grandpa tab
 - man color → blue
 - woman color → red
 - *update*



another view . . .

Types

<input checked="" type="checkbox"/> Man	<input checked="" type="checkbox"/> Show			
<input type="text" value="Man"/>	<input type="color" value="Blue"/>	<input type="text" value="Inherit"/>	<input type="text" value="— Inherit"/>	
<input type="checkbox"/> Align by type	<input type="checkbox"/> Hide unconnected nodes	<input checked="" type="checkbox"/> Number nodes		
<input checked="" type="checkbox"/> Person	<input checked="" type="checkbox"/> Show			
<input type="text" value="Person"/>	<input type="color" value="Inh..."/>	<input type="text" value="Inherit"/>	<input type="text" value="— Inherit"/>	
<input type="checkbox"/> Align by type	<input type="checkbox"/> Hide unconnected nodes	<input checked="" type="checkbox"/> Number nodes		
<input checked="" type="checkbox"/> Woman	<input checked="" type="checkbox"/> Show			
<input type="text" value="Woman"/>	<input type="color" value="Red"/>	<input type="text" value="Inherit"/>	<input type="text" value="— Inherit"/>	
<input type="checkbox"/> Align by type	<input type="checkbox"/> Hide unconnected nodes	<input checked="" type="checkbox"/> Number nodes		



types & sets

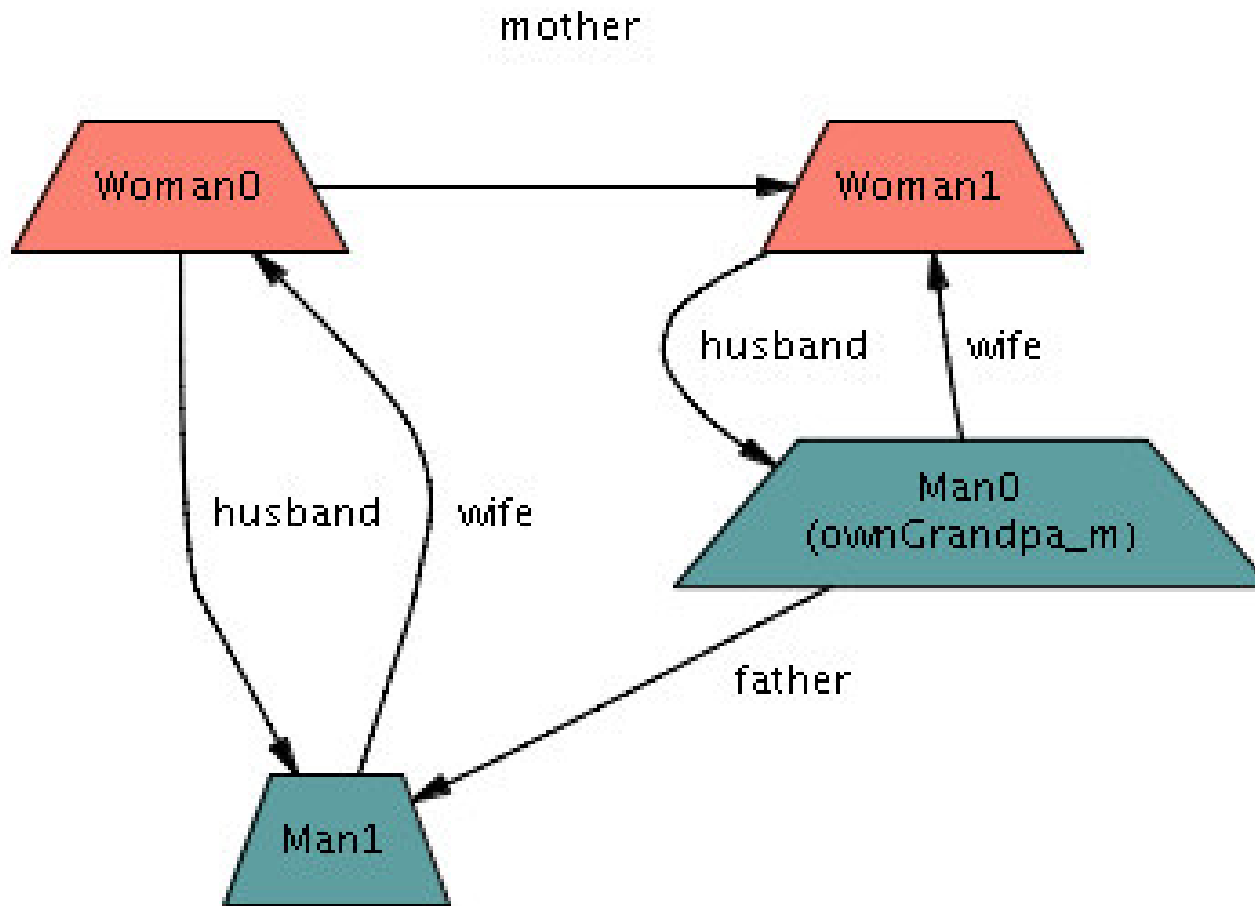
- types: from signatures
 - person shape → trapezoid
 - notice it carries down to man, woman
 - woman: align by type
 - *update*



Types

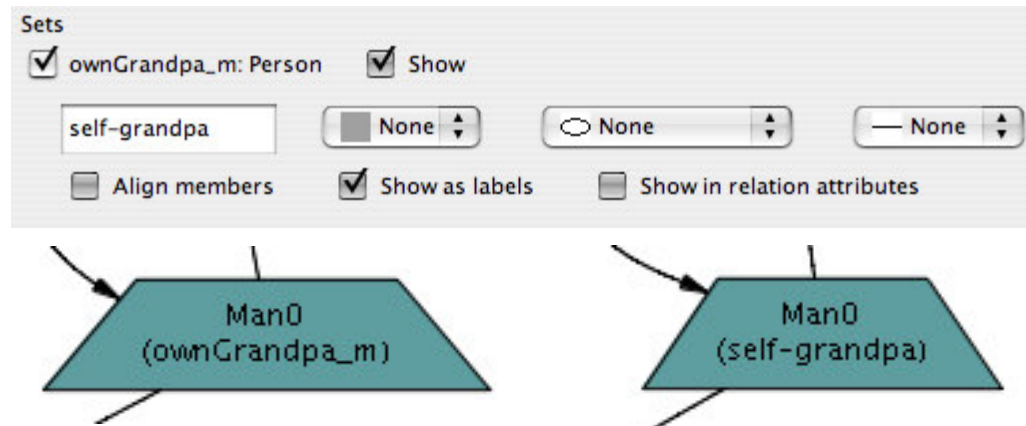
<input checked="" type="checkbox"/> Man	<input checked="" type="checkbox"/> Show			
<input type="text" value="Man"/>	<input type="text" value="Blue"/>	<input type="text" value="Inherit"/>	<input type="text" value="Inherit"/>	
<input type="checkbox"/> Align by type	<input type="checkbox"/> Hide unconnected nodes	<input checked="" type="checkbox"/> Number nodes		
<input checked="" type="checkbox"/> Person	<input checked="" type="checkbox"/> Show			
<input type="text" value="Person"/>	<input type="text" value="Inh..."/>	<input type="text" value="Trapezoid"/>	<input type="text" value="Inherit"/>	
<input type="checkbox"/> Align by type	<input type="checkbox"/> Hide unconnected nodes	<input checked="" type="checkbox"/> Number nodes		
<input checked="" type="checkbox"/> Woman	<input checked="" type="checkbox"/> Show			
<input type="text" value="Woman"/>	<input type="text" value="Red"/>	<input type="text" value="Inherit"/>	<input type="text" value="Inherit"/>	
<input checked="" type="checkbox"/> Align by type	<input type="checkbox"/> Hide unconnected nodes	<input checked="" type="checkbox"/> Number nodes		

types & sets



types & sets

- sets: from existentials, runs, checks
 - somewhat intelligently named
 - ownGrandpa_m label → self-grandpa
 - *update*

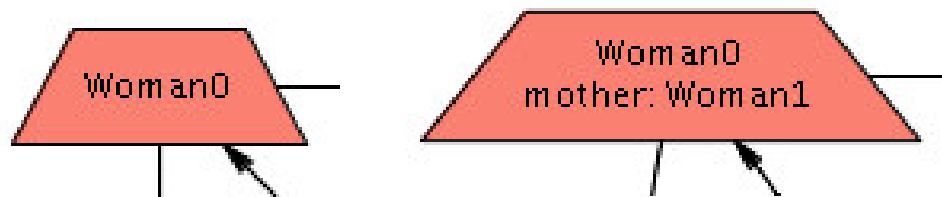
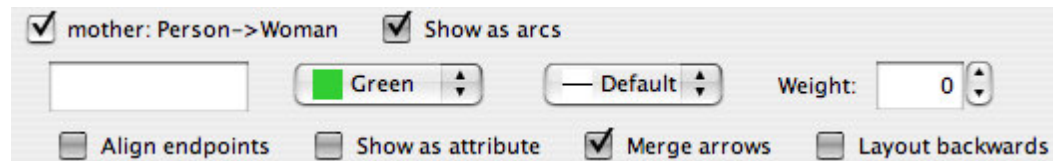
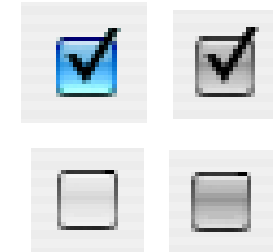


- pitfall: don't show vs. don't show as label (vs. don't show in customizer...)

relations

- relations

- mother: show as attribute → check (still shown as arc)
- gray = inherited (vs. overridden)
- *update*



relations

- relations
 - mother: show as attribute → uncheck
 - father, mother, husband, wife: label → “”
 - father, mother: color → green
 - husband, wife: color → yellow
 - *update*



relations

Relations

father: Person->Man Show as arcs
[] Green Default Weight: 0

Align endpoints Show as attribute Merge arrows Layout backwards

husband: Woman->Man Show as arcs
[] Yellow Default Weight: 0

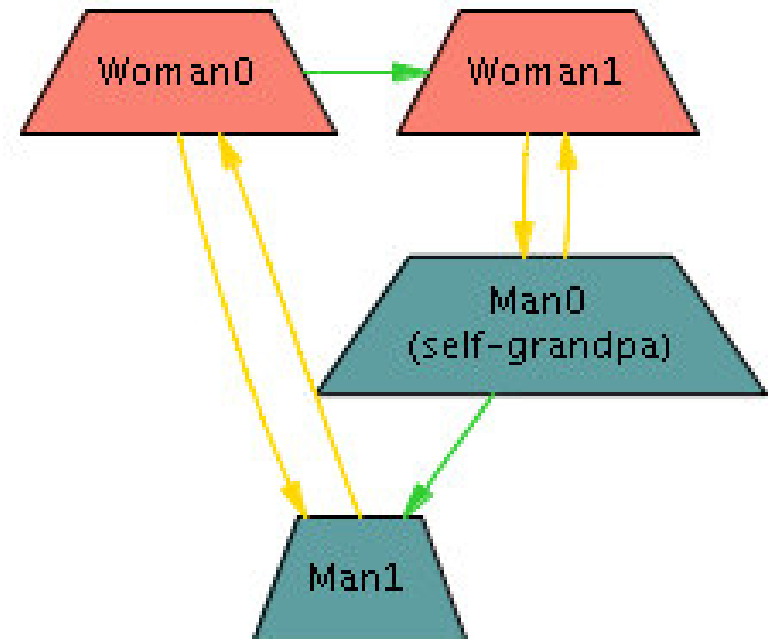
Align endpoints Show as attribute Merge arrows Layout backwards

mother: Person->Woman Show as arcs
[] Green Default Weight: 0

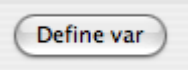
Align endpoints Show as attribute Merge arrows Layout backwards

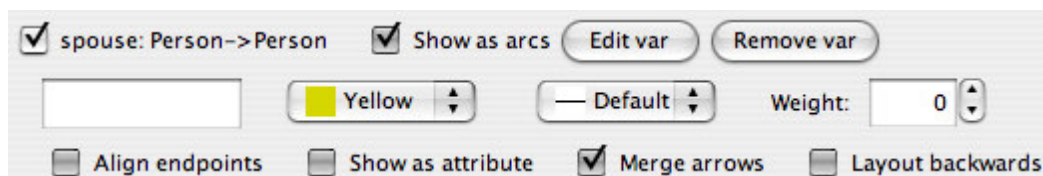
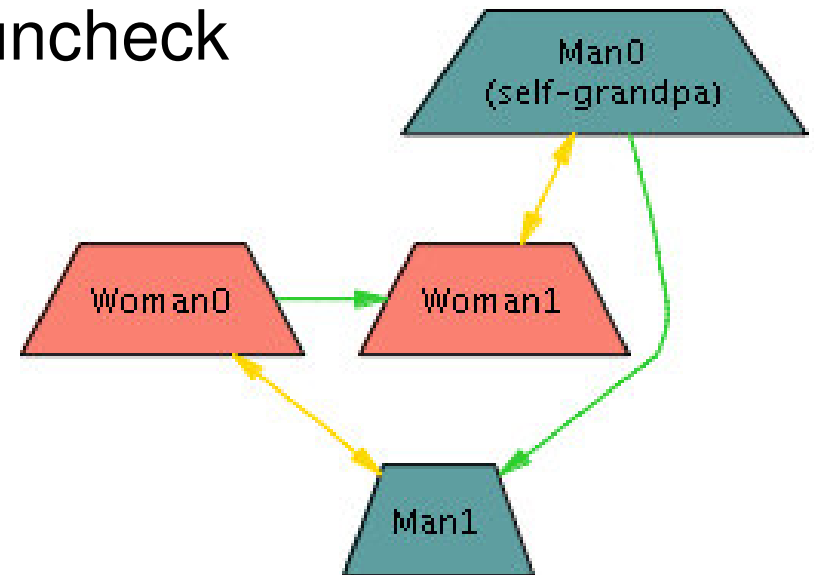
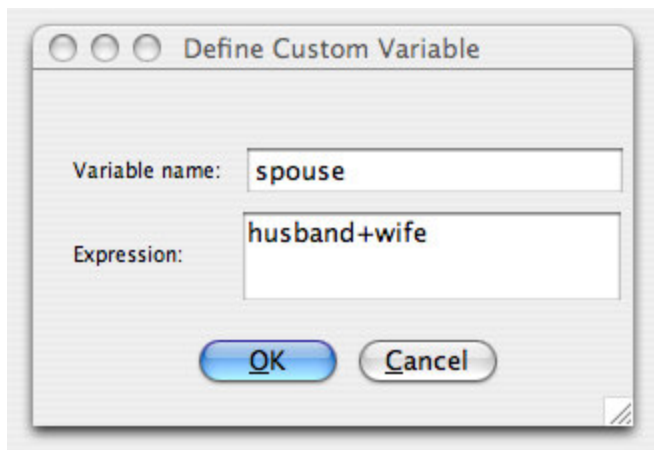
wife: Man->Woman Show as arcs
[] Yellow Default Weight: 0

Align endpoints Show as attribute Merge arrows Layout backwards



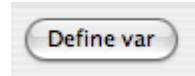
defined variables

- Define var 
 - spouse = husband + wife
 - spouse: label = "", color = yellow
 - husband, wife: show as arcs: uncheck



defined variables

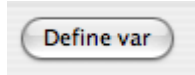
- Define var



– married = ?

defined variables

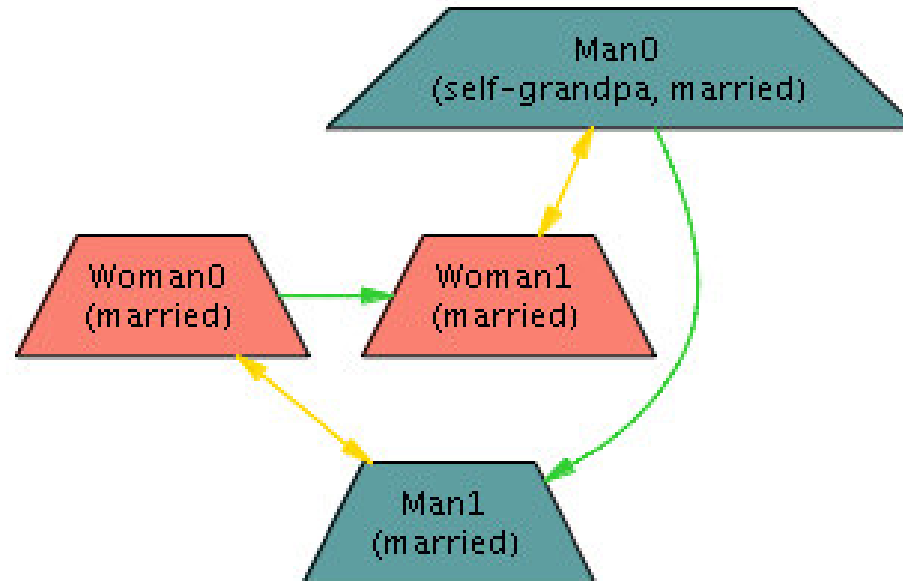
- Define var



- married = *Person.(husband + wife)*



- *update*



- *handy trick: define in order to hide*

finishing up

- views
 - save palette
- close layout



- create your own visualization for the barber exercise!