

Specification with Alloy — Further Examples, More Foundations

Jan van Eijck

jve@cwi.nl

Master SE, 19 November 2008

Declarative Specification: Trees

- What makes a binary relation B (for **B**ranching) into a tree?
- There should be exactly one root r , where the root r is defined as an object without B -predecessors.
- Every other object should have exactly one B -predecessor.
- The relation B should be a-cyclic.

Tree Specification in Alloy

```
module myexamples/tree
```

```
sig Object { b: set Object }
```

```
one sig Root, A, B, C, D extends Object {}
```

```
fact OneRoot { all x: Object | x = Root <=> no b.x }
```

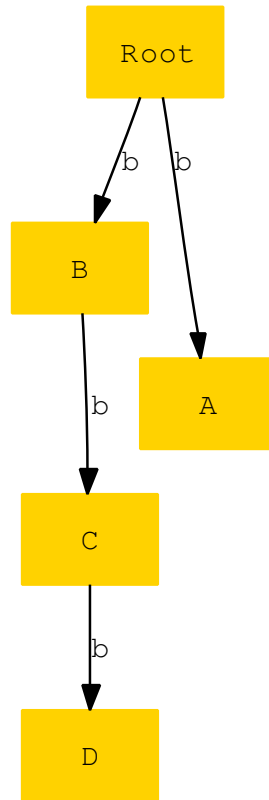
```
fact b_acyclic { no ^b & iden }
```

```
fact { C in B.b and D in C.b }
```

```
pred show () {}
```

```
run show for 5
```

Example Model



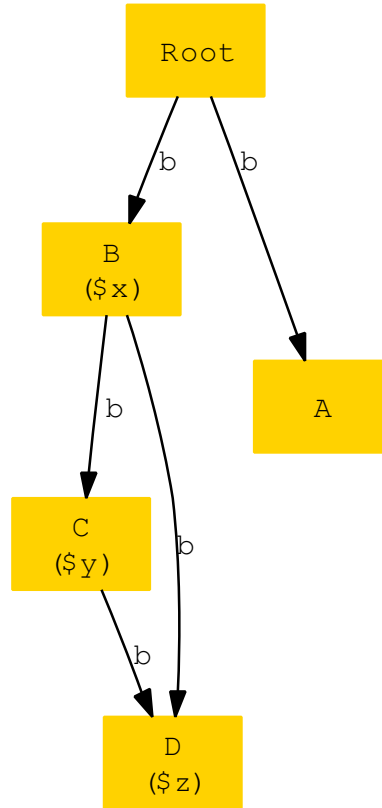
Check

- This looks OK, but is it?
- Hmm, maybe not. We forgot to say that every other object but the root has exactly one B -predecessor.
- Maybe it follows from our specification. Let us check.
- Alloy assertion:

```
assert SingleParent
  { all x,y,z: Object | z in x.b and z in y.b => x=y }
check SingleParent for 5
```

- Check this.

Counterexample to Single Parenthood



Corrected Specification

```
module myexamples/tree
```

```
sig Object { b: set Object }
```

```
one sig Root, A, B, C, D extends Object {}
```

```
fact OneRoot { all x: Object | x = Root <=> no b.x }
```

```
fact SingleParent
```

```
  { all x,y,z: Object | z in x.b and z in y.b => x=y }
```

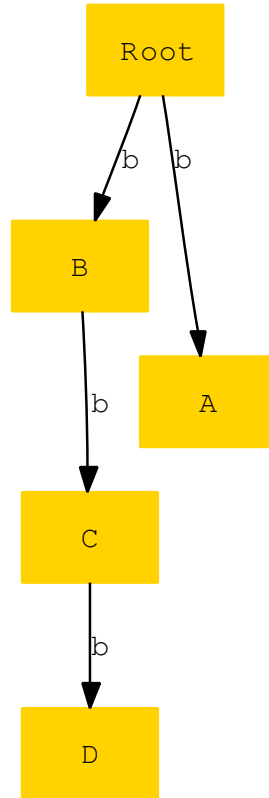
```
fact b_acyclic { no ^b & iden }
```

```
fact { C in B.b and D in C.b }
```

```
pred show () {}
```

```
run show for 5
```

Example Model Again



Tree Specification Again

- What makes a binary relation P (for **P**arenthood) into a tree?
- There should be exactly one root r , where the root r is defined as an object without parents (P -successors).
- Every other object should have exactly one P -successor.
- The relation P should be a-cyclic.
- Note the difference with the previous specification: in the previous specification the arrows pointed towards the branches, now the arrows point to the parents.

Alloy Version of This

```
module myexamples/newtree
```

```
sig Object { p: lone Object }
```

```
one sig Root, A, B, C, D extends Object {}
```

```
fact OneRoot { all x: Object | x = Root <=> no x.p }
```

```
fact SingleParent
```

```
  { all x,y,z: Object | y in x.p and z in x.p => y=z }
```

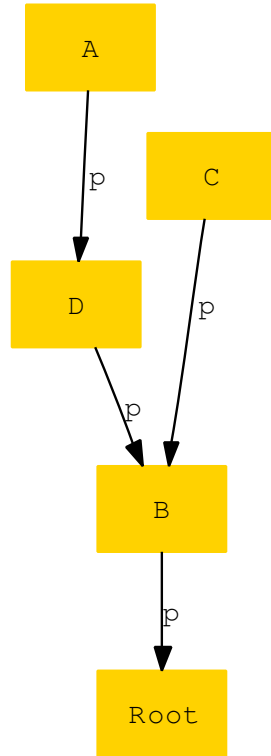
```
fact p_acyclic { no ^p & iden }
```

```
fact { C in p.B and D in p.B }
```

```
pred show () {}
```

```
run show for 5
```

Example Model



A Check

- Hmm, we expressed single parenthood now as:

```
fact SingleParent
  { all x,y,z: Object | y in x.p and z in x.p => y=z }
```

- I wonder, is this equivalent to the following:

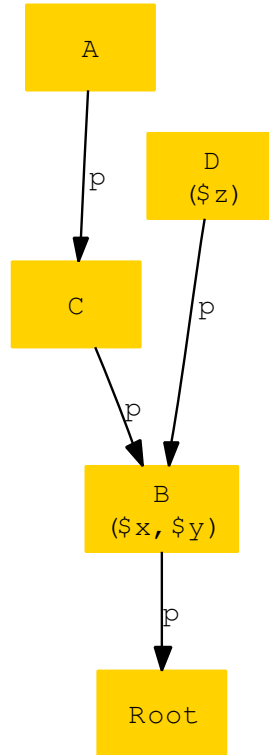
```
{ all x,y,z: Object | z in p.x and z in p.y => y=z }
```

- Let us check:

```
assert SingleParent'
  { all x,y,z: Object | z in p.x and z in p.y => y=z }
check SingleParent' for 5
```

- No, there is a counterexample ...

Counterexample to Single Parenthood'



Analysis and Repair

- Hmm, I see. I made a mistake in the assertion.
- What I meant to say is that I wonder whether the single parenthood fact is equivalent to the following:

$$\{ \text{all } x,y,z: \text{Object} \mid z \text{ in } p.x \text{ and } z \text{ in } p.y \Rightarrow x=y \}$$

- Let us check again:

```
assert SingleParent''
```

```
{ all x,y,z: Object | z in p.x and z in p.y => x=y }  
check SingleParent'' for 5
```

- No counterexample found. Assertion may be valid.
- Still not good enough for me. I am starting to wonder if a single parenthood fact is needed at all ...

Checking for Redundancy in the Specification

- Look at the specification of p .
- p : lone Object.
- Doesn't this imply the single parenthood fact?
- Let us check: replace the **SingleParent** fact by the following assertion:

```
assert SingleParent
```

```
{ all x,y,z: Object | y in x.p and z in x.p => y=z }
```

```
check SingleParent for 5
```

- No counterexample found. Assertion may be valid.
- That's good enough for me. I am satisfied.

Final Version of Alloy Tree Specification

```
module myexamples/newtree
```

```
sig Object { p: lone Object }
```

```
one sig Root, A, B, C, D extends Object {}
```

```
fact OneRoot { all x: Object | x = Root <=> no x.p }
```

```
fact p_acyclic { no ^p & iden }
```

```
fact { C in p.B and D in p.B }
```

```
pred show () {}
```

```
run show for 5
```

```
assert SingleParent
  { all x,y,z: Object | y in x.p and z in x.p => y=z }
check SingleParent for 5
```

```
assert SingleParent2
  { all x,y,z: Object | z in p.x and z in p.y => x=y }
check SingleParent2 for 5
```

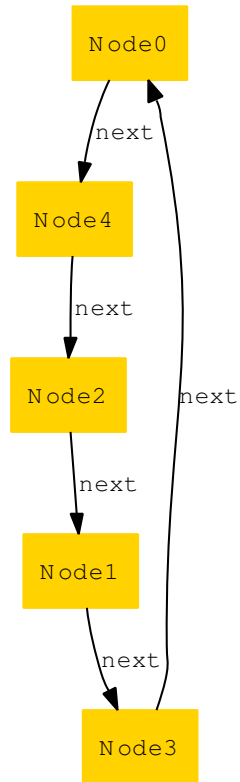
Specifying Rings

- Some communication protocols assume that nodes in a network form a ring. Links from node to node should form a circle.
- Assume a binary relation `next`, and specify the constraints on `next` that force this relation to form a ring.
- Alloy specification:

```
module myexamples/ring
sig Node { next : set Node }
pred isRing () {
    // your constraints go here ...
}
run isRing for exactly 5 Node
```

- Part of your homework for this week ...

Ring Example Model



Declarative Specification: Sudoku Solving

- If declarative specification is to be taken seriously, all there is to solving sudokus is **specifying what a sudoku problem is**.
- A sudoku is a 9×9 matrix of numbers in $\{1, \dots, 9\}$ satisfying a number of constraints:
 - Every **row** should contain each number in $\{1, \dots, 9\}$
 - Every **column** should contain each number in $\{1, \dots, 9\}$
 - Every **subgrid** $[i, j]$ with i, j ranging over 1..3, 4..6 and 7..9 should contain each number in $\{1, \dots, 9\}$.
- A sudoku **problem** is partial sudoku matrix (a list of values in the matrix).
- A **solution** to a sudoku problem is a complete extension of the problem, satisfying the sudoku constraints.

Example Problem, With Solution

5	3	7		5	3	4	6	7	8	9	1	2			
6		1	9	5		6	7	2	1	9	5	3	4	8	
	9	8			6		1	9	8	3	4	2	5	6	7
8		6		3	8	5	9	7	6	1	4	2	3		
4		8	3	1	4	2	6	8	5	3	7	9	1		
7		2		6	7	1	3	9	2	4	8	5	6		
	6			2	8	9	6	1	5	3	7	2	8	4	
		4	1	9		5	2	8	7	4	1	9	6	3	5
		8		7	9		3	4	5	2	8	6	1	7	9

Sudoku Constraints: Surjectivity

- To express the sudoku constraints, we have to be able to express the property that a function is **surjective** (or: **onto**, or: a **surjection**).
- A function $f : X \rightarrow Y$ is a surjection if every element of Y is the f -image of some element of X .
- Equivalently: a function $f : X \rightarrow Y$ is surjective if

$$Y \subseteq \{f(x) \mid x \in X\}.$$

Sudoku Constraints as Surjectivity Requirements

- Represent a sudoku as a function $f[i, j]$.
- Requirements:
 - Every **row** should contain each number in $\{1, \dots, 9\}$.
I.e., for every i , the function $j \mapsto f[i, j]$ should be surjective (onto).
I.e., for all i : $\{1, \dots, 9\} \subseteq \{f[i, j] \mid j \in \{1..9\}\}$.
 - Every **column** should contain each number in $\{1, \dots, 9\}$
I.e., for every j , the function $i \mapsto f[i, j]$ should be surjective (onto).
I.e., for all j : $\{1, \dots, 9\} \subseteq \{f[i, j] \mid i \in \{1..9\}\}$.
 - Every **subgrid** $[i, j]$ with i, j ranging over 1..3, 4..6 and 7..9 should contain each number in $\{1, \dots, 9\}$.

I.e., $\{1, \dots, 9\} \subseteq \{f[i, j] \mid i, j \in \{1..3\}\}$, and so on ...

Surjectivity in Alloy

```
module myexamples/surjection
```

```
sig Object { f : Object }
```

```
pred f_surjective { Object in f[Object] }
```

```
run f_surjective for exactly 4 Object
```

Example Surjective Function



Sudoku Constraints in Alloy

Signature:

```
module myexamples/sudoku
```

```
abstract sig Num { sudoku : Num one -> one Num }
```

```
abstract sig B1, B2, B3 extends Num {}
```

```
one sig N1, N2, N3 extends B1 {}
```

```
one sig N4, N5, N6 extends B2 {}
```

```
one sig N7, N8, N9 extends B3 {}
```

Rows and columns are onto:

```
fact rows_onto { all x: Num | Num in sudoku[x,Num] }
```

```
fact columns_onto { all y: Num | Num in sudoku[Num,y] }
```

Sudoku Constraints in Alloy (ctd)

Subgrids are onto:

```
fact B1B1_onto { Num in sudoku[B1,B1] }
fact B1B2_onto { Num in sudoku[B1,B2] }
fact B1B3_onto { Num in sudoku[B1,B3] }
fact B2B1_onto { Num in sudoku[B2,B1] }
fact B2B2_onto { Num in sudoku[B2,B2] }
fact B2B3_onto { Num in sudoku[B2,B3] }
fact B3B1_onto { Num in sudoku[B3,B1] }
fact B3B2_onto { Num in sudoku[B3,B2] }
fact B3B3_onto { Num in sudoku[B3,B3] }
```

Redundancy in the Specification

- Look at the declaration of `sudoku`.
- `sudoku : Num one -> one Num.`
- Doesn't this constrain each row relation to be a one-to-one function?
- If so, then the `rows_onto` constraint should be redundant.
- Let us check: replace the `rows_onto` fact by an assertion:

```
assert rows_onto { all x: Num | Num in sudoku[x,Num] }  
check rows_onto
```
- No counterexample found. Assertion may be valid.
- Since the domain is fixed to 9 numbers, we can conclude that the assertion `is` valid. So the rows constraint `is` redundant.

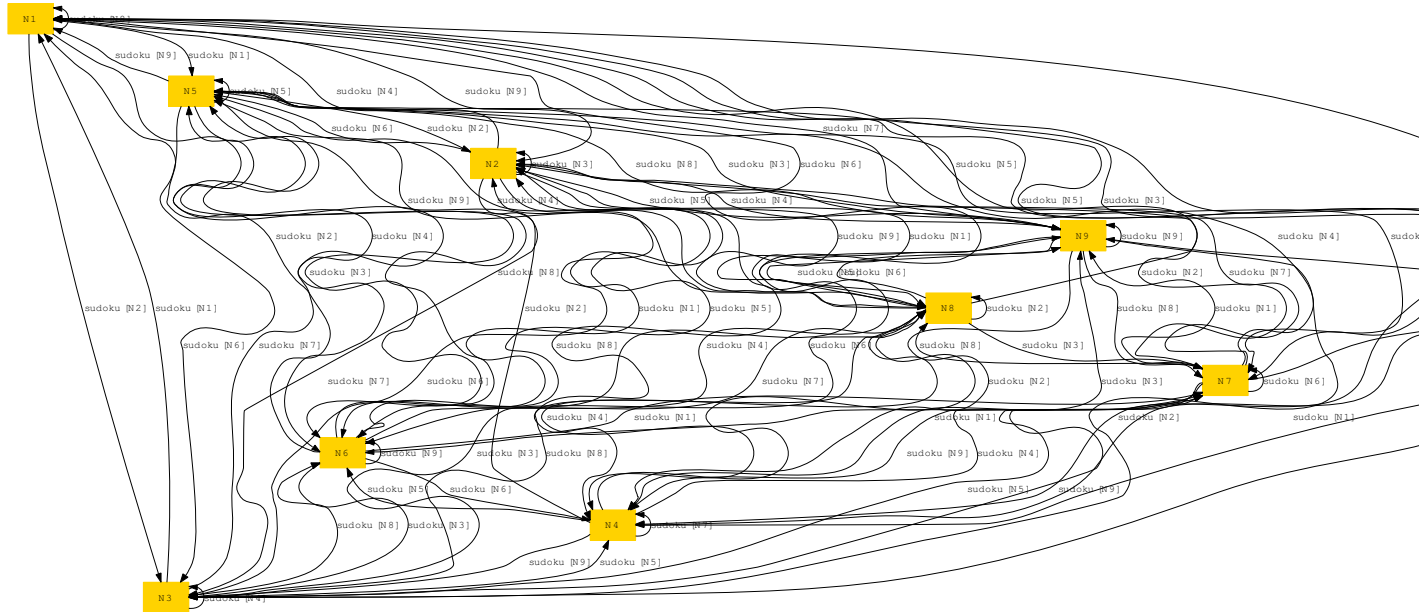
Example Sudoku Problem in Alloy

```
fact problem {  
    N1->N5 + N2->N3 + N5->N7                in sudoku[N1]  
    N1->N6 + N4->N1 + N5->N9 + N6->N5 in sudoku[N2]  
    N2->N9 + N3->N8 + N8->N6                in sudoku[N3]  
    N1->N8 + N5->N6 + N9->N3                in sudoku[N4]  
    N1->N4 + N4->N8 + N6->N3 + N9->N1 in sudoku[N5]  
    N1->N7 + N5->N2 + N9->N6                in sudoku[N6]  
    N2->N6 + N7->N2 + N8->N8                in sudoku[N7]  
    N4->N4 + N5->N1 + N6->N9 + N9->N5 in sudoku[N8]  
    N5->N8 + N8->N7 + N9->N9                in sudoku[N9]  
}
```

```
pred show () {}
```

```
run show
```

Solution Model



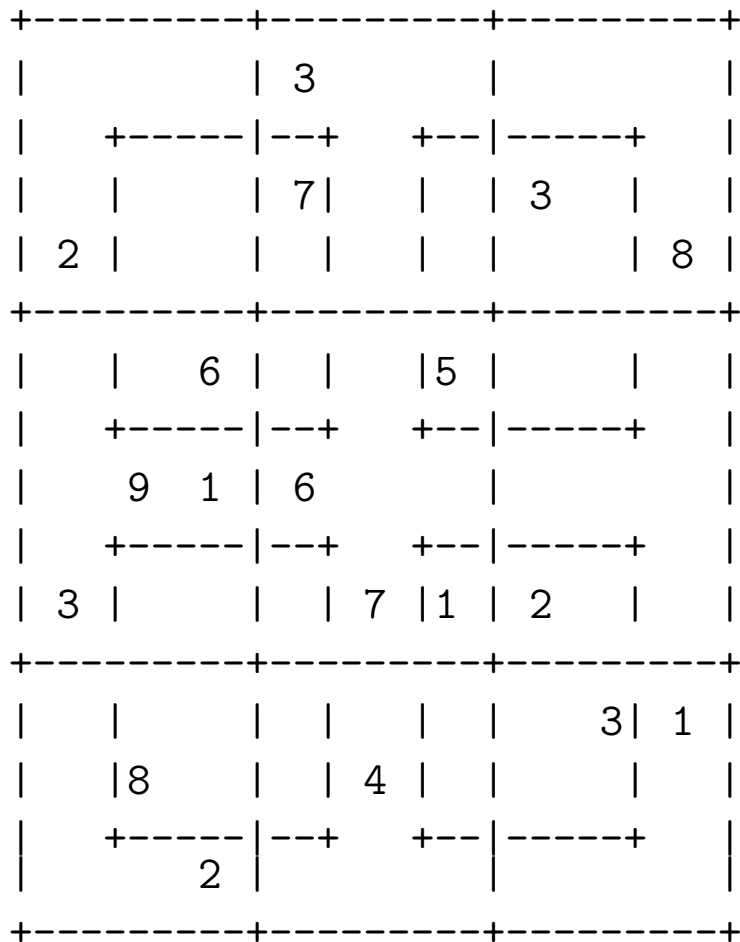
Oops

Another View

```
<atom name="N1.10"/> <atom name="N1.10"/> <atom name="N5.10"
<atom name="N1.10"/> <atom name="N2.10"/> <atom name="N3.10"
<atom name="N1.10"/> <atom name="N3.10"/> <atom name="N4.10"
<atom name="N1.10"/> <atom name="N4.10"/> <atom name="N6.10"
<atom name="N1.10"/> <atom name="N5.10"/> <atom name="N7.10"
<atom name="N1.10"/> <atom name="N6.10"/> <atom name="N8.10"
<atom name="N1.10"/> <atom name="N7.10"/> <atom name="N9.10"
<atom name="N1.10"/> <atom name="N8.10"/> <atom name="N1.10"
<atom name="N1.10"/> <atom name="N9.10"/> <atom name="N2.10"
<atom name="N2.10"/> <atom name="N1.10"/> <atom name="N6.10"
<atom name="N2.10"/> <atom name="N2.10"/> <atom name="N7.10"
<atom name="N2.10"/> <atom name="N3.10"/> <atom name="N2.10"
...
```

Extended Sudokus: Peter Ritmeester Constraints

- The sudokus that appear in NRC-Handelsblad each Saturday (designed by Peter Ritmeester, from Oct 8, 2005 onward) are special in that they have to satisfy a few extra constraints.
- In addition to the usual sudoku constraints, each of the 3×3 subgrids with left-top corner $(2,2)$, $(2,6)$, $(6,2)$, and $(6,6)$ should also yield a surjective function.
- Part of your homework: formalize these extra constraints in Alloy and use this to solve the NRC-Handelsblad sudoku puzzle of Saturday November 25, 2006. The puzzle appears in the **Leven Etcetera** part of the newspaper.



4	7	8	3	9	2	6	1	5
6	1	9	7	5	8	3	2	4
2	3	5	4	1	6	9	7	8
7	2	6	8	3	5	1	4	9
8	9	1	6	2	4	7	5	3
3	5	4	9	7	1	2	8	6
5	6	7	2	8	9	4	3	1
9	8	3	1	4	7	5	6	2
1	4	2	5	6	3	8	9	7

Semantics of Alloy

M : formula \rightarrow env \rightarrow boolean

X : expr \rightarrow env \rightarrow value

env = (var + type) \rightarrow value

value = $\mathcal{P}(\text{atom} \times \text{atom}) + (\text{atom} \rightarrow \text{value})$

$$M[a \text{ in } b]e = X[a]e \subseteq X[b]e$$

$$M[!F]e = \neg M[F]e$$

$$M[F \ \&\& \ G]e = M[F]e \wedge M[G]e$$

$$M[F \ || \ G]e = M[F]e \vee M[G]e$$

$$M[\text{all } v:t \ | \ F]e = \bigwedge \{M[F](e \oplus v \mapsto x)(x, \text{unit}) \in e(t)\}$$

$$X[a + b]e = X[a]e \cup X[b]e$$

$$X[a \& b]e = X[a]e \cap X[b]e$$

$$X[a - b]e = X[a]e \setminus X[b]e$$

Alloy FAQs

- What are similarities and differences between Alloy and Rscript?
- What are similarities and differences between Alloy and Prolog?
- What are similarities and differences between Alloy and other specification languages?

Alloy vs Rscript

- Similarity: Both Use First Order Logic with Relational Operations
- Differences:
 - Alloy specifications are constraints on possible models
 - Rscript programs provide descriptions of actual models.
- Alloy is more abstract than Rscript.
- Alloy allows automated testing of assertions, Rscript does not.
- Alloy and Rscript serve different purposes. Rscript is not a specification language.
- Rscript is geared towards generating software metrics, Alloy is not.

Alloy vs Prolog

- Alloy uses full first order logic with relational operations
- Prolog uses a subset of first order logic (plus fixpoint operations).
- Alloy does not have search strategies, while Prolog allows operations on search trees.
- Prolog uses a particular theorem proving strategy, while Alloy hides the details of theorem proving (these are taken care of by the SAT component).
- Alloy vs Prolog: Specification vs Programming.

Alloy vs Other Specification Languages

- Alloy vs UML:
 - Alloy has a much more precise semantics.
 - Alloy has automated testing of assertions, UML has not.
- Alloy vs Z:
 - Z is more expressive than Alloy
 - Alloy has automated testing of assertions, Z has not.
 - The expressive power of Alloy is still considerable.
 - Limitation to first order logic is not a matter or concern for many specification tasks.