Specifications and Assertions

Jan van Eijck

Master SE, 29 September 2010

Abstract

As a start, we give further examples of Alloy specifications. Next we turn to specification of imperative programs. Assertions about programs are specifications of how the program is supposed to behave. Assertions can be used for correctness reasoning and for testing. We illustrate the important notions of preconditions and postconditions. We demonstrate how the state transitions of imperative programming can be modelled as relations in Alloy.

Correctness reasoning can be linked to testing and debugging by means of executable assertions, and by means of random generation of test cases based on preconditions and postconditions.

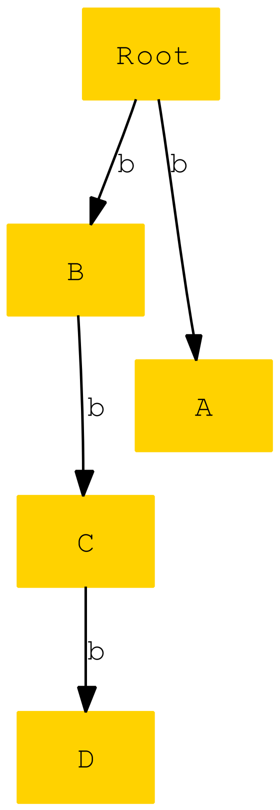
Declarative Specification: Trees

- What makes a binary relation B (for Branching) 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



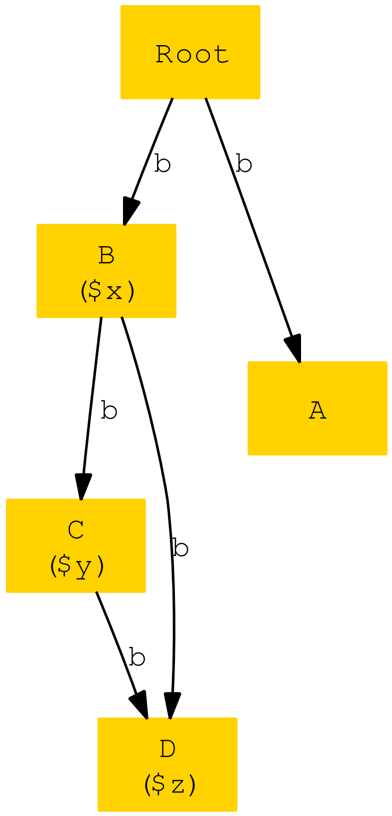
Check

- 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.

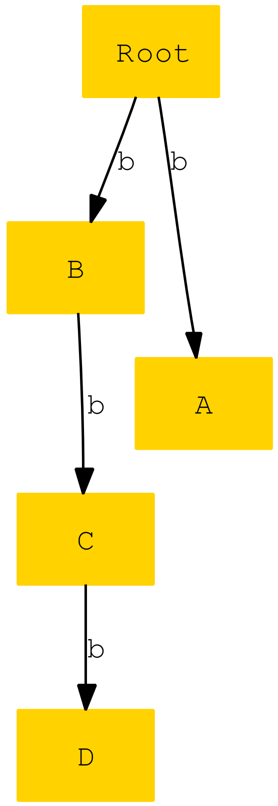
Counterexample to Single Parenthood



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


$$\begin{tikzcd}
 & \text{Root} \arrow[dl, "b"] \arrow[d, "b"] \\
\text{B} \arrow[d, "b"] & \text{A} \\
\text{C} \arrow[d, "b"] & \\
\text{D} &
\end{tikzcd}$$

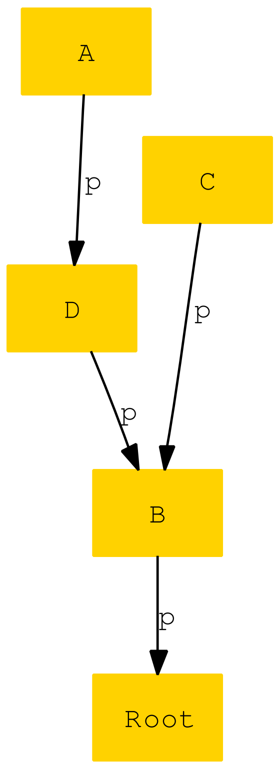
Tree Specification Again

- What makes a binary relation P (for Parenthood) 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



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
```

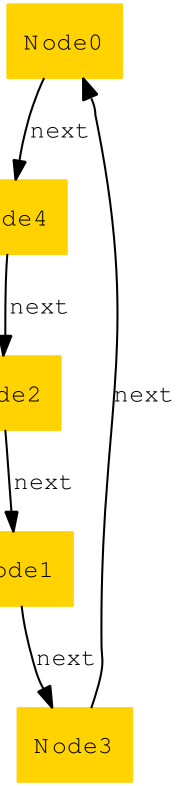
Specifying Rings

- 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 a 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 $f : X \to Y$ is a surjection if every element of Y is the f-image of some element of X.
- Equivalently: a function $f : X \to Y$ is surjective if

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

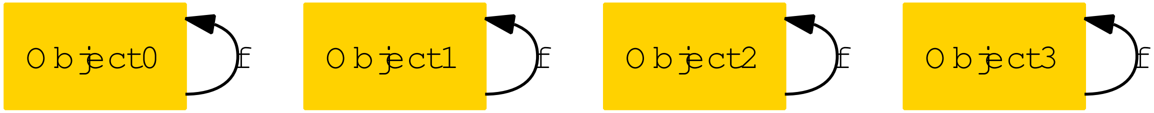
Sudoku Constraints as Surjectivity Requirements

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

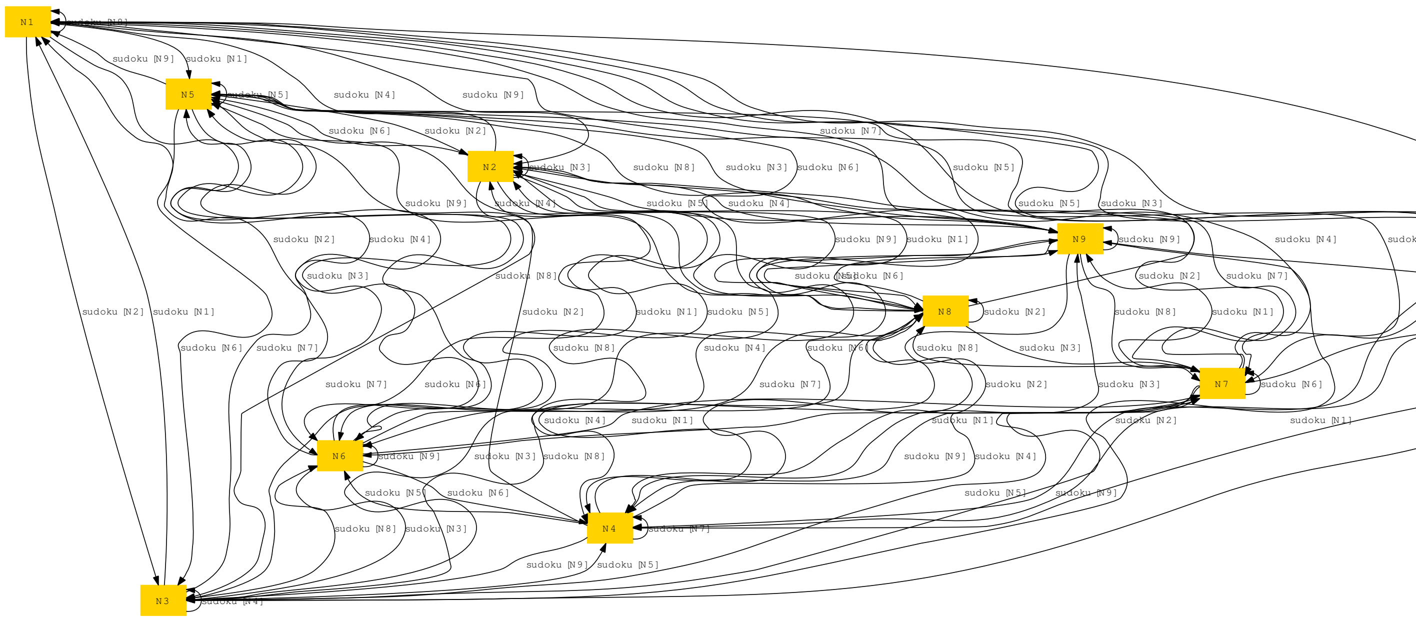
```
assert rows_onto { all x: Num | Num in sudoku[x,Num] }
check rows_onto
```

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

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 October 2, 2010. The puzzle appears in the glossy part of the newspaper.

```
+---------+---------+---------+
|         | 3       |         |
|   +-----|--+   +--|-----+   |
|   |     | 7|   |  | 3   |   |
| 2 |     |  |   |  |     | 8 |
+---------+---------+---------+
|   |   6 |  |   |5 |     |   |
|   +-----|--+   +--|-----+   |
|    9  1 | 6       |         |
|   +-----|--+   +--|-----+   |
| 3 |     |  | 7 |1 | 2   |   |
+---------+---------+---------+
|   |     |  |   |  |    3| 1 |
|   |8    |  | 4 |  |     |   |
|   +-----|--+   +--|-----+   |
|       2 |         |         |
+---------+---------+---------+
+---------+---------+---------+
| 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 |
+---------+---------+---------+
```

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.

An Abstract Look at Imperative Programming

Example Program with Variables a, b, c

```
puts "give two whole numbers"
puts "first number"
a = gets.to_i
puts "second number"
b = gets.to_i
if a > b then c = a else c = b end
puts "#{c} is the maximum of #{a} and #{b}"
```

State of a Program at an Execution Point

This very simple example program uses three integer variables a, b, c.

Assume a has value 3, b has value -8 and c has value 11. Then we say that the state of the program at that point is given by:

$a == 3$ and $b == -8$ and $c == 11$

Or, using logical symbols:

$a == 3 \ \wedge \ b == -8 \ \wedge \ c == 11$

Such triples of values for the variable triple can also be represented as points in three dimensional space. The point $(3, -8, 11)$ is such a point in 3D space.

Predicates

Programs are often used to compute values. A program with two program variables x and y can ask the user for a value for x, and then return the value x^2 in y:

```
puts "give a whole number"
x = gets.to_i
y = x**2
puts "y equals #{y}"
```

$y == x^2$ is a statement about programming variables that we call a predicate.

Predicates are in fact open formulas of first order logic (open formulas are formulas with unbound variables).

More Examples of Predicates

$$x == y$$

$$x + y \geq 0$$

$$x^2 + y^2 == 25$$

$$x == 5$$

$$y < 0$$

If x and y have type integer then the state pairs (x, y) will have whole values for x en y, so the states are grid points in 2D space (or: grid-points in the plane).

A predicate on x, y denotes a subset of the plane. The predicate $x == y$ denotes the grid points on a line through the plane origin $(0, 0)$. The predicate $x + y \geq 0$ denotes a half-plane, and so on.

Examples, continued

The predicate $x \geq 0$ denotes the set of points (x, y) with $x \geq 0$ and y arbitrary. In a picture, these are the points on the y-axis and to the right of the y-axis, a half plane.

The predicate $y \geq 0$ defines the points (x, y) up and above the x-axis.

The conjunction $x \geq 0 \wedge y \geq 0$ corresponds with the intersection of these two sets, i.e., it denotes the first quadrant of the plane.

The disjunction $x \geq 0 \vee y \geq 0$ corresponds with the union of these two sets, i.e., it denotes everything in the plane except for the third quadrant.

Empty Set and Universe

No point satisfies the predicate $x^2 == -4$ (if x ranges over integers, or over floating points numbers). We say: the predicate denotes the empty set. Notation for the empty set: $\emptyset$.

We also say: the predicate is equivalent to $\perp$ or to false. No state satisfies false. No program ever ends up in a state that satisfies false.

The counterpart of the empty set is the whole domain of discourse, or the whole universe. In the case at hand: the set of all pairs.

Examples of predicates denoting the whole universe are, for example:

$$x^2 \geq 0 \wedge y^2 \geq 0$$

$$(x+1)^2 == x^2 + 2x + 1$$

Such predicates we call equivalent to $\top$, or to true.

Every state satisfies true, and if a program terminates then its terminating state always satisfies true.

Stronger, Weaker

The predicate true is the weakest of all predicates: it does not rule out anything at all, so it does not constrain the programming variables in any way. The predicate false is the strongest of all predicates: it never holds.

In general we call a predicate P stronger than a predicate Q in case everywhere $P \Rightarrow Q$ holds. In such a case we call Q weaker than P.

For instance, we have:

$x > 0$ is stronger than $x \geq 0$	$[x > 0 \Rightarrow x \geq 0]$
$x^2 == 1$ is weaker than $x == 1$	$[x^2 == 1 \ \Leftarrow \ x == 1]$
false is stronger than $x == 3$	$[\text{false} \Rightarrow x == 3]$
true is weaker than $x^2 + 2x == 15$	$[\text{true} \ \Leftarrow \ x^2 + 2x == 15]$

Meanings of ‘Stronger Than’ and ‘Weaker Than’

Explication of stronger than in terms of set denotations:

P is stronger than Q if the denotation of P is a subset of the denotation of Q.

Note that P is both stronger than and weaker than P.

It may happen that predicates are not related by stronger/weaker at all. This will occur if the denotation of one predicate is not contained in that of the other, nor vice versa.

Substitution

When in a predicate each occurrence of variable x gets replaced by an expression E we get a new predicate. For this replacement or substitution we use the following notation:

$$P(x \leftarrow E) \qquad P \text{ with } x \text{ replaced by } E$$

Example:

$$\begin{array}{ll} & (x \geq 0)(x \leftarrow x{-}1) \\ \equiv & \{\text{substitution}\} \\ & x{-}1 \geq 0 \\ \equiv & \{\text{algebra}\} \\ & x \geq 1 \end{array}$$

More Examples

$$\begin{aligned}
& (x-y \geq 0)(y \leftarrow x+y) \\
\equiv\ & \{\textsf{substitution}\} \\
& x-(x+y) \geq 0 \\
\equiv\ & \{\textsf{algebra}\} \\
& y \leq 0
\end{aligned}$$

$$\begin{aligned}
& (x < 0)(x \leftarrow x^2) \\
\equiv\ & \{\textsf{substitution}\} \\
& x^2 < 0 \\
\equiv\ & \{\textsf{algebra}\} \\
& \perp
\end{aligned}$$

Reasoning about Assignment

```
    if a>b then c=a else c=b
```

If prior to this selection statement the following assertion holds:

$$a == A \wedge b == B$$

then afterwards the following holds:

$$a == A \wedge b == B \wedge c == \text{the maximum of } A \text{ and } B$$

Hoare Triples

We state the relation between the initial state of this statement, where "$a == A \wedge b == B$" holds, and the final state where

$$a == A \wedge b == B \wedge c == \text{the maximum of } A \text{ and } B$$

holds, as follows:

$$\{a == A \wedge b == B\}$$

```
if a>b then c=a else c=b
```

$$\{a == A \wedge b == B \wedge c == \text{the maximum of } A \text{ and } B\}$$

The first element is a predicate in curly brackets. This specifies the initial state.

The second element is a program statement.

The third element is a predicate in curly brackets. This specifies the final state.

In general a triple

initial state – statement – final state

$$\{P\}\ S\ \{Q\}$$

has the following operational meaning:

> If execution of S in a state that satisfies P terminates, then the termination tate is guaranteed to satisfy Q.

Such triples $\{P\}\,S\,\{Q\}$ are called Hoare triples after the British computer scientist and Turing award winner C.A.R. Hoare (Tony Hoare).

The predicate for the initial state if often called the precondition, and the predicate for the final state is often called the postcondition.

Partial Correctness + Termination = Total Correctness

Note that a Hoare triple $\{P\}\ S\ \{Q\}$ makes a conditional assertion:

if the initial state satisfies P and if the execution of S in that state terminates, then the result state will satisfy Q.

In cases where S involves repetition, termination is often a crucial issue.

To express that S terminates we need extra machinery (see below).

The Hoare triple assertion does not express that the program will terminate. That's why we call Hoare triple assertions partial correctness assertions.

Total correctness $=$ partial correctness $+$ termination.

Example of Non-deterministic Processing

Consider the following Ruby program:

```
x = rand 4; y = 3 - x; puts(x,y)
```

The Ruby function `rand 4` generates a random integer number smaller than 4 and greater than or equal to 0. The result is that both x and y get assigned a number from the set $\{0, 1, 2, 3\}$, and that it holds that $x + y = 3$. From these two facts it follows that this postcondition has to hold:

$$x \geq 2 \vee y \geq 2$$

Clearly, we can not derive from this that $x \geq 2$, nor that $y \geq 2$.

Invalid Inference

The example shows that from

$$\{P\}\, S\, \{Q \vee R\}$$

one cannot infer that

$$(\{P\}\, S\, \{Q\}) \vee (\{P\}\, S\, \{R\})$$

Meaning of $\{\textbf{true}\}\, S\, \{Q\}$

Taking P in $\{P\}\, S\, \{Q\}$ to be the predicate true, one gets:

$$\{\text{true}\}\, S\, \{Q\}$$

with — in accordance with what we stipulated above — the following meaning:

> Execution of S starting from a state that satisfies true
> will, if S terminates, end in a state satisfying Q.

Since every state satisfies true, this can be simplified to:

> Execution of S starting from any state
> will, if S terminates, end in a state satisfying Q.

Hoare Triples for Assignment

To apply the above to a real programming construct, let us look at the assignment statement. In Ruby (or C, or Java) the simplest form of assignment looks like this: $x = E$ ("x becomes E"). The programming language Pascal uses $x := E$ for assignment.

Examples of assignments are:

$$\begin{aligned} x &= 3 && \text{(give } x \text{ the value 3)} \\ x &= y + 1 && \text{(give } x \text{ the value of } y + 1) \\ x &= x + 1 && \text{(increment } x \text{ by 1)} \end{aligned}$$

Operational meaning of $x = E$

replace the value of x by the value of E

We can ask ourselves whether we can come up with a rule that expresses the appropriate form of the Hoare triple:

$$\{P\}\ x = E\ \{Q\}$$

For the simple first example $x = 3$ it seems reasonable to conclude that in the final state $x == 3$ holds, irrespective of the initial state.

This we can express as:

$$\{\textsf{true}\}\ x = 3\ \{x == 3\}$$

For the second example ($x = y + 1$) we can assume that the following assertions are all correct:

$$\begin{aligned}
\{y == 3\}\ \ & x = y + 1\ \ \{x == 4\} \\
\{y == 5\}\ \ & x = y + 1\ \ \{x == 6\} \\
\{y > 0\}\ \ & x = y + 1\ \ \{x > 1\} \\
\{y == x\}\ \ & x = y + 1\ \ \{x == y + 1\}
\end{aligned}$$

How about a general rule?

Let's have a closer look at $x = x + 1$.

Assume that just before execution of $x = x{+}1$ holds that $x == 3$. Then immediately after execution of this assignment it will surely hold that $x == 4$. Thus, we have:

$$\{x == 3\}\ x = x{+}1\ \{x == 4\}$$

This says that if x equals 3 and then x gets incremented by 1, then afterwards x equals 4. Surely correct.

For more complicated expressions it is more difficult to work out these things by just staring at them.

The Substitution Rule for Assignment

To ensure that after $x = x{+}1$ it holds that $x == 4$ it has to be the case that beforehand $x{+}1 == 4$ holds. This is because the value of x gets replaced by that of $x{+}1$.

More generally: the initial condition that has to be satisfied to guarantee some particular condition Q holds in the state after execution of $x = E$ is calculated by replacing x by E in Q.

This rule is known as the substitution rule for assignment.

In the example case: the final condition is $x == 4$. To get the initial condition, we have to replace x by $x{+}1$. This gives $x{+}1 == 4$, which is exactly the initial condition $x == 3$.

A Fallacy

One might be tempted to calculate the final condition from the initial condition by doing the substitution in the precondition to find the postcondition.

This is wrong.

Consider the example

$$\{x == 0\} \;\; x = x + 1 \;\; \{Q\}.$$

Subsitution of $x + 1$ for x in $x == 0$ gives $x + 1 == 0$, i.e., $x == -1$. This gives the wrong predicate for Q.

So: always calculate the initial condition from the final condition, not vice versa.

Applying the Substitution Rule

As an example, we calculate the initial condition for the following case:

$$\{\cdots\}\ x = 2 * x + 4\ \{0 \leq x < 8\}$$

Substitution of $2 * x + 4$ for x in the final condition $0 \leq x < 8$ gives as initial condition $0 \leq 2*x+4 < 8$. This can be simplified as follows:

$$\begin{aligned} & 0 \leq 2 * x + 4 < 8 \\ \equiv\ & \text{subtract 4} \\ & -4 \leq 2 * x < 4 \\ \equiv\ & \text{divide by 2} \\ & -2 \leq x < 2 \end{aligned}$$

Therefore

$$\{-2 \leq x < 2\}\ x = 2 * x + 4\ \{0 \leq x < 8\}$$

Weakest Preconditions

The initial condition $-2 \leq x < 2$ is the condition that has to be satisfied at least. Every value within the interval is an appropriate initial value for x. Therefore the following is also correct:

$$\{0 \leq x < 2\}\ x = 2 * x + 4\ \{0 \leq x < 8\}$$

This is because from $0 \leq x < 2$ it follows that $-2 \leq x < 2$, or, more formally:

$$0 \leq x < 2 \Rightarrow -2 \leq x < 2\,.$$

We call $-2 \leq x < 2$ the weakest precondition.

Summing up the above, we can say that the weakest precondition for the assignment $x = E$, given postcondition Q, is calculated by replacing x by E in Q.

Rule for Assignment, Two Versions

Finding the weakest precondition from the postcondition:

$$\boxed{\{Q(x \leftarrow E)\} \quad x = E \quad \{Q\} \text{ always holds.}}$$

Checking whether a given postcondition fits the precondition:

$$\boxed{\{P\}\, x = E\, \{Q\} \text{ is equivalent with } P \Rightarrow Q(x \leftarrow E)}$$

Example Calculation

We want to show the following:

$$\{\, x == n^2 \,\} \;\; x = x + 2 * n + 1 \;\; \{\, x == (n{+}1)^2 \,\}$$

According to the rule for assignment we have to show that the post-condition with x replaced by $x + 2n + 1$ follows from the precondition. In other words:

$$x == n^2 \Rightarrow (x == (n{+}1)^2)(x \leftarrow x + 2n + 1)$$

Work out the righthand-side:

$$
\begin{array}{cl}
 & (x == (n{+}1)^2)(x \leftarrow x + 2n + 1) \\
\equiv & \text{substitution} \\
 & x + 2n + 1 == (n{+}1)^2 \\
\equiv & \text{algebra} \\
 & x + 2n + 1 == n^2 + 2n + 1 \\
\equiv & \text{algebra} \\
 & x == n^2
\end{array}
$$

This proves what we had to show.

Next Example

We want to show:

$$\{\, x == (n{+}1)^2 \,\}\ n = n{+}1\ \{\, x == n^2 \,\}$$

According to the rule for assignment we have to show that:

$$x == (n{+}1)^2 \Rightarrow (x == n^2)(n \leftarrow n{+}1)$$

Working with the righthand-side again:

$$\begin{aligned} & (x == n^2)(n \leftarrow n{+}1) \\ \equiv\ & \text{substitution} \\ & x == (n{+}1)^2 \end{aligned}$$

This is indeed the precondition we are after.

The Rule for Sequencing

In Ruby, as in many other programming languages, semicolon ; can be used to compose statements into a new statement on a single line. Here is a rule for this:

$$\frac{\{P\}\ S_1\ \{R\} \qquad \{R\}\ S_2\ \{Q\}}{\{P\}\ S_1; S_2\ \{Q\}}$$

Using this rule, we can combine our examples:

$$\{\, x == n^2 \,\}\ x = x + 2 * n + 1;\ n = n{+}1\ \{\, x == n^2 \,\}$$

Note: it does not follow from this that nothing has changed. Instead, the Hoare triple expresses that the relation $x == n^2$ has not changed.

The statements have incremented the value of n by 1.

If the precondition is

$$x == n^2 \wedge n == 10$$

then the postcondition will be

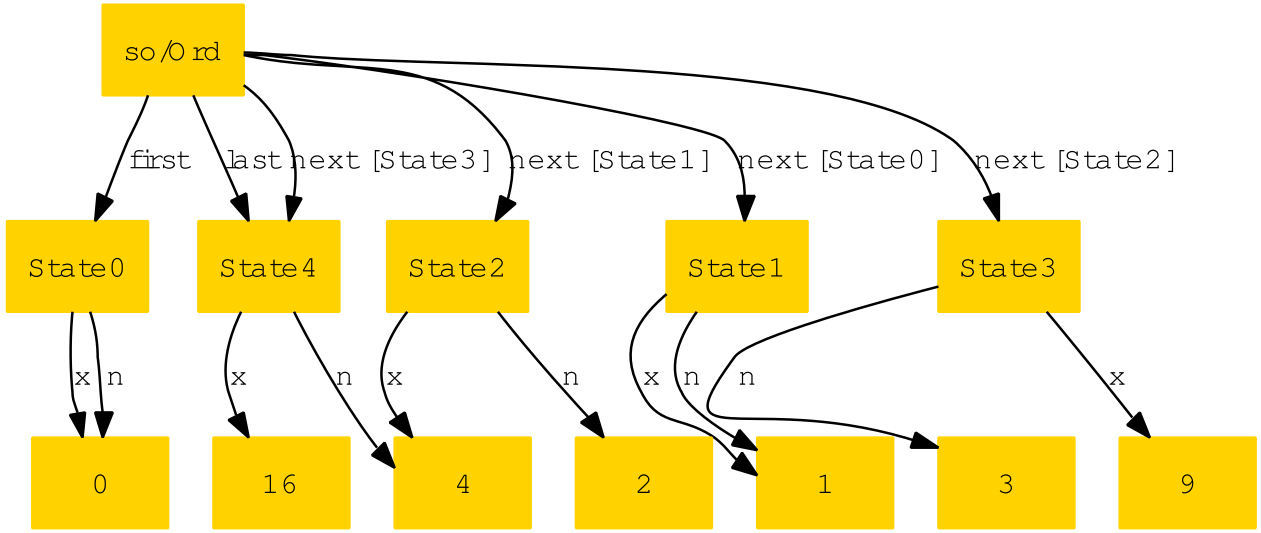
$$x == n^2 \wedge n == 11$$

You should check this.

An Alloy Specification of an Imperative Program

```
module myexamples/invar
open util/ordering[State] as so
open util/integer as integer

sig State {
  x: Int,
  n: Int
}

fun inc [n : Int]: Int { add [n,Int[1]] }

pred init {
  let fs = so/first |
   { fs.x = Int[0] and fs.n = Int[0] }
}
```

```
pred extend [pre, post: State] {
  some X,N: Int | pre.x = X
                  and pre.n = N
                  and post.x = inc[add[X,add[N,N]]]
                  and post.n = inc[N]
}

fact createStates {
   init
   all s: State - so/last |
          let s’ = so/next[s] | extend[s,s’]
}

run {} for exactly 5 State, 6 int
```


so/Ord
first
last
next [State3]
next [State1]
next [State0]
next [State2]
State0
State4
State2
State1
State3
x
n
x
n
x
n
x
n
n
x
0
16
4
2
1
3
9