## Synchronized Traffic Lights

Create a model of the traffic lights so the drivers encounter a green wave, a series of green lights (the technical term is progression).

This will involve satisfying a huge number of simultaneous conditions. This is exactly the kind of problem that SAT-based tools such as Alloy are designed to solve.

I created an Alloy model to see if the problem can be solved.

Here is a snapshot of a 4x4 grid of intersections and traffic lights:



Each intersection is numbered. Intersection0 is at the bottom, left corner. Intersection15 is at the top, right corner. At each intersection there are up to 4 traffic lights. Intersection0 has just 2 traffic lights: the traffic light facing to the north is red, the traffic light facing west is yellow. Notice that intersection15 has a green light in the east direction and intersection11 has a yellow light in the east direction. At the next time step intersection 15’s east light is red and intersection 11’s east light is green:



Now notice that intersection7 has a yellow east-facing traffic light. At the next time step it is green:



See how the traffic lights are synchronized as a car progresses eastward? Look back at the first graphic and check out all green traffic lights. Follow them through the graphics and you will see that all the lights are synchronized, in each direction (north, south, east, west).

The number of constraints that must be simultaneously satisfied to synchronize all the lights is mindbogglingly huge. Think of the constraints involved:

If intersection0 at time0 has a north-facing light that is green,
 then the intersection connected to it at the north
 (intersection1) must at time1 have its north-facing
 light green, and the intersection connected to intersection1
 at the north (intersection2) must at time2 have its
 north-facing light green. And so forth.

intersection0, time0, north-facing light at intersection0, color, intersection0’s connection to the north, etc. are *variables* in a constraint expression.

I created an Alloy model. I ran the model. Look at the number of variables in the constraint expression:



It took Alloy (i.e., the SAT tool) 12 seconds, running on my laptop to find all the instances that satisfy the constraints.

There are two parts to modeling this traffic light grid system:

1. Model the physical system -- the intersections and the traffic lights. For each intersection i, the intersection that i connects to in the north direction, the intersection that i connects to in the south direction, etc. For each intersection i, the traffic light facing north, the traffic light facing south, etc.
2. Model the constraints on the traffic lights. Here are the constraints:
	1. For each intersection i, if the intersection to the west has its east-facing traffic light green at time t-1, then i's east-facing traffic light (if there is one) must be green at time t. Ditto for the other directions.
	2. At each intersection, at least one traffic light changes color at each time step.
	3. Traffic lights change color in the proper sequence: red -> yellow -> green -> red
	4. At each intersection there is at most one traffic light that is not red.

Let’s modularize the model. Let’s have 3 Alloy (.als) files:



An Alloy file imports another Alloy file using the “open” command. At the top of Constraints.als are two open commands:

**open** Traffic\_Lights
**open** Intersections

A 4x4 grid has 47 traffic lights (see graphics above). Let’s enumerate them:

**abstract** **sig** Traffic\_Light {
 color: Color **one** -> Time
}
**one** **sig** Traffic\_Light0 **extends** Traffic\_Light {}
**one** **sig** Traffic\_Light1 **extends** Traffic\_Light {}
…
**one** **sig** Traffic\_Light47 **extends** Traffic\_Light {}

Here is how Color is modeled:

**abstract** **sig** Color {}
**one** **sig** red **extends** Color {}
**one** **sig** yellow **extends** Color {}
**one** **sig** green **extends** Color {}

Recall that this:

 A -> B

means: All the pairs obtained by matching each member of A with each member of B. The arrow operator is also known as the Cartesian Product.

Suppose that we don’t want all combinations. Suppose we want only those pairs where each member of B is paired with only one member of A. One way to specify this constraint is by preceding the arrow operator with “one”:

 A **one** -> B

That constrains the pairs to each member of B paired to one member of A.

Alternatively, we could use the unqualified arrow operator:

 A -> B

And then eliminate the undesired pairs with a fact:

 **fact** Each\_B\_paired\_to\_one\_A\_only {
 **all** b: B |
 **no** a, a’: A | ((a -> b) **in** Example.pairs) **and** ((a’ -> b) **in** Example.pairs)
 }

Now we can understand this:

 color: Color **one** -> Time

It says that each member of Time is paired with one color. In other words, a traffic light can only have one color at a time.

The following signature specifies the order that traffic lights change color:

**one** **sig** Color\_Sequence {
 sequence: red -> yellow +
 yellow -> green +
 green -> red
}

Create a signature for Time. Order the time values.

**open** util/ordering[Time]

**sig** Time {}

That’s it for Traffic\_Lights.als

Now let’s examine Intersections.als

Intersections are arranged in a north/south, east/west manner, so specify the directions:

**abstract** **sig** Direction {}
**one** **sig** north **extends** Direction {}
**one** **sig** south **extends** Direction {}
**one** **sig** west **extends** Direction {}
**one** **sig** east **extends** Direction {}

It will be convenient to have a signature which specifies, for each direction, its opposite direction:

**one** **sig** OppositeDirection {
 reverse: Direction -> Direction
} {
 reverse = north -> south +
 south -> north +
 west -> east +
 east -> west
}

Notice that after the signature declaration is a set of curly braces containing an expression. That is called a *signature fact*. It is an anonymous fact. The expression applies to each member of the signature (in this case, the signature has just one member). The signature fact constrains reverse to the pairs (north, south), (south, north), (west, east), (east, west).

These two are equivalent:

|  |  |
| --- | --- |
| **Uses signature fact** | **Uses explicit, named fact** |
| **one** **sig** OppositeDirection { reverse: Direction -> Direction} { reverse = north -> south + south -> north + west -> east + east -> west} | **one** **sig** OppositeDirection { reverse: Direction -> Direction} **fact** Constrain\_the\_reserve\_field { OppositeDirection.reverse = north -> south + south -> north + west -> east + east -> west} |

A 4x4 grid has 16 intersections (see above graphic). Let’s enumerate them:

**abstract** **sig** Intersection {
 connections: Direction **lone** -> lone Intersection,
 lights: Direction **lone** -> **lone** Traffic\_Light
}
**one** **sig** Intersection0 **extends** Intersection {} {
 connections = north -> Intersection1 +
 west -> Intersection4
 lights = north -> Traffic\_Light0 +
 west ->Traffic\_Light1
}

Each intersection specifies its connections to other intersections and the lights at the intersection. Above we see how intersection0 is specified. Here is a graphical depiction of the intersection:



intersection0 connects to intersection1 to the north and intersection0 connects to intersection4 to the west. Traffic light0 faces to the north. Traffic light1 faces to the west.

Notice the arrow operator. It is constrained with “lone” on either side. “lone” means optional (zero or one). So, this:

 Direction **lone** -> **lone** Traffic\_Light

means that there are pairs containing (direction, traffic light) but not all possible pairs. Specifically, a traffic light is paired to at most one direction (there is at most one traffic light in a direction). And, each direction is paired to at most one traffic light. For intersection0 there are these pairs:

(north, Traffic\_Light0), (west, Traffic\_Light1)

The connection field creates triples: (Intersection, Direction, Intersection). For example, it creates these triples for intersection0:

(Intersection0, north, Intersection1), (Intersection0, west, Intersection4)

Here’s an alternate way to represent the triples:



The lights field creates these triples: (Intersection, Direction, Traffic\_Light). For example, it creates these triples for Intersection0 and Intersection1:

(Intersection0, north, Traffic\_Light0), (Intersection0, west, Traffic\_Light1), (Intersection1, south, Traffic\_Light3), (Intersection1, north, Traffic\_Light2), (Intersection1, west, Traffic\_Light4)

Here’s an alternate way to represent the triples:



What Alloy expression returns the north-facing Traffic\_Light at Intersection0?

Let d = (north) and i = (Intersection0)

First, join i and lights:



Now, join north with the result of i.lights:



These two forms are equivalent:

|  |  |
| --- | --- |
| d.(i.lights) | i.lights[d] |

The second form uses the *box notation*.

The color field creates these triples: (Traffic\_Light, Color, Time). For example, it creates these triples for Traffic\_Light0 and Traffic\_Light1:

(Traffic\_Light0, red, Time0), (Traffic\_Light1, red, Time)

Here’s an alternate way to represent the triples:



What Alloy expression returns the color of the north-facing Traffic\_Light at Intersection0 at Time0?

Let d = (north) and i = (Intersection0) and t = Time0



i.lights[d].color.t

yields red.

That’s it for Intersections.als

Now let’s move on the Constraints.als

Here are the constraints we want to apply to the traffic lights:

1. The below graphic illustrates the first constraint.



Intersection11 connects to intersection15 to the west, and intersection15’s east-facing traffic light is green at time t-1, so intersection11’s east-facing traffic light must be green at time t.

So, if intersection i connects to intersection j to the west, and j’s east-facing traffic light is green at the previous time step, then i’s east-facing traffic light must be green at the present time step (if i has an east-facing traffic light). Ditto for all of i’s connections and for all intersections.

2. At each intersection, at least one traffic light changes color at each time step.

3. Traffic lights change color in the proper sequence: red -> yellow -> green -> red

4. At each intersection there is at most one traffic light that is not red.

Now let’s see how to express the constraints using the Alloy language.

1. For each intersection i, if the intersection to the west has its east-facing traffic light green at time t-1, then i's west-facing traffic light (if there is one) must be green at time t. Ditto for all of i’s connections and for all intersections. And, of course, for every time step.

**fact** Lights\_are\_synchronized {
 **all** t: Time - first |
 **all** i: Intersection |
 **all** d: i.connections.Intersection |
 i.connections[d].lights[OppositeDirection.reverse[d]].color.(t.prev) = green =>
 **some** i.connections[OppositeDirection.reverse[d]] =>
 i.lights[OppositeDirection.reverse[d]].color.t = green
}

Let’s walk through that **fact**, step by step.













Recap: If intersection i connects to intersection j to the west, and j’s east-facing traffic light is green at the previous time step, then i’s east-facing traffic light (if there is one) must be green at the present time step.

2. At each intersection, at least one traffic light changes color at each time step.

**fact** Constantly\_changing\_lights {
 **all** t: Time - first |
 **all** i: Intersection |
 **some** d: Direction |
 i.lights[d].color.(t.prev) != i.lights[d].color.t
}

3. Traffic lights change color in the proper sequence: red -> yellow -> green -> red

**fact** Lights\_change\_color\_in\_the\_proper\_sequence {
 **all** t: Time - first |
 **all** light: Traffic\_Light |
 light.color.(t.prev) != light.color.t =>
 light.color.t = Color\_Sequence.sequence[light.color.(t.prev)]
}

4. At each intersection there is at most one traffic light that is not red.

**fact** At\_most\_one\_non\_red\_light {
 **all** t: Time |
 **all** i: Intersection |
 **no** **disj** d, d': Direction |
 ((i.lights[d].color.t = green) **or** (i.lights[d].color.t = yellow)) **and**
 ((i.lights[d'].color.t = green) **or** (i.lights[d'].color.t = yellow))
}