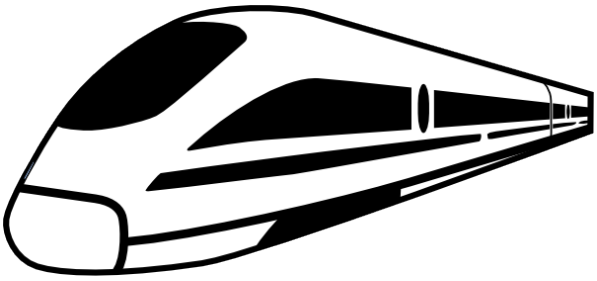
INFO-F-412 : Formal Verification
Bidirectional switch for trains

Feras Almasri, Colas Goeminne, Tim Lenertz,
Quentin-Emmanuel Vajda and Renaud Vilain

June 22, 2014



Contents

List of Figures

1 Introduction

This paper will introduce the problem we tackled ; to create a controller for a problem of critical sections in bidirectional train tracks using a model based approach.

At first, the general idea of the problem and how the controller should behave will be explained.

To follow that, we will describe the model we used to define the system, and the properties of it we formally verified.

1.1 General idea

The goal of this project is to elaborate and simulate a bidirectional switch for trains. We can see on Figure 1 our train crossing. There are four incoming ways

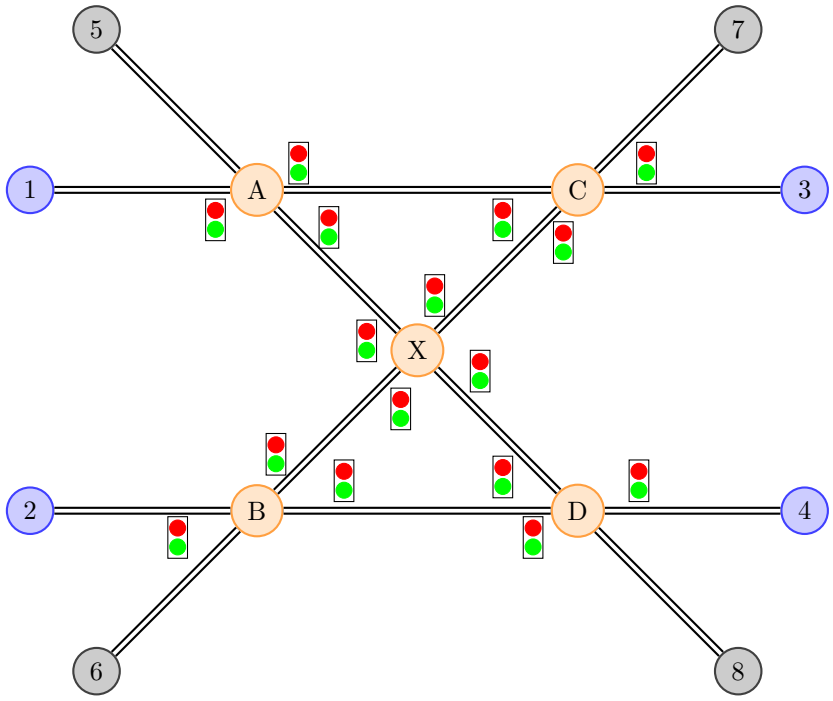


Figure 1: Plan of our train crossing

into the system (points 1, 2, 3 and 4) and four outgoing ways (points 5, 6, 7 and 8). There are also five switches (A, B, C, D and X).
Here can you find the different possible paths trains passing through the system can take:

- $1 \to A \to C \to 7$
- $1 \to A \to X \to D \to 8$
- $2 \to B \to X \to C \to 7$
- $2 \to B \to D \to 8$
- $3 \to C \to A \to 5$
- $3 \to C \to X \to B \to 6$

- $4 \to D \to B \to 6$
- $4 \to D \to X \to A \to 5$

In this system, the tracks are bidirectional and as such major critical sections as the main goal of our controller is to avoid collisions of two trains on the same segment going in opposite directions. Collisions avoidance within each of the five intersections is also desired.

1.2 Embedded and critical system

The model produced covers two aspects of the problem : the environment and the (embedded) controller.

In this case, the environment is simply the movement of the trains across the tracks as well as the choice of origin-destination pair for each train entering the system.

The controller itself is split into two parts ; the controller of the actual physical train track switches and that of the traffic lights. For the sake of simplicity, it is assumed that trains will always stop at a red light, continue when the light becomes green, and avoid collision with another train in front of them on the same track.

2 Model

2.1 Entities

Hereunder is the description of the logical entities of the model ; abstracted from their implementation.

2.1.1 Train

In our model, we can have several trains in the system at the same time. Each train comes from point the incoming way at point A, B, C or D and follows one of the possible paths, as described above.

These is the *environment* part of our model ; meaning they are not controlled in any way. As such, trains will seemingly randomly appear at one of the entrance to the system with a just as random destination. In practice we limit the total number of trains in the system at any point to a reasonable and realistic amount.

The train model is based around the progress of each train along its chosen[1] path. Trains travel over each segment within a minimal and maximal amount of time, dependent on the segment length, and will wait at each intersection for the traffic light to turn green.

When multiple train wait at an intersection they will queue up in front of the traffic light ; a queue exists for each incoming way of each intersection. As such, point X has four distinct queues and each other switch only three. These *FIFO* queues model the property that trains on one track cannot overtake each other.

The progress of trains in the system is the succession of travels within a segment of tracks, then waiting at the intersection, and so on.

[1] origin-destination pair chosen non-deterministically

2.1.2 Switch

As shown in Figure 1, we have five switches representing each of the intersections between multiple set of tracks. These physical switches can be modelled very easily based on the fact that they can only have two positions. In our model, the controller for the traffic lights commands the physical switch directly by changing from one position to the other when needed.

For the four corner points, the two possible switch positions are either towards the X point, or towards the straight portion of the track. For the X switch, the two positions simply join either the A-D or B-C track segments.

2.1.3 Traffic lights

To manage trains, we have implemented two types of traffic lights controller. One overseeing switches A, B, C and D ; another one specifically designed for switch X. The former controlling three traffic lights (one for each incoming way) and the latter, four.

The controller is notified when a train is incoming and waiting at a traffic light of the correct intersection as well as when a train leaves the intersection, freeing the critical section. The traffic light can then be switched if possible to any train waiting for it ; depending on a set of static priorities. The four corner intersections also need to check the availability of the next tracks segment whenever a train is entering the system, in order to avoid having trains going in different directions on the same tracks segment.
After deciding which incoming way will be allowed to enter the intersection (if any) the controller will first check the position of the underlying tracks switch and force it to change if needed. After that, the traffic light will be updated accordingly which should result in the movement of the waiting train.

2.2 Models in UPPAAL

In this section, we will further formalize the description of the logical entities for implementation using the UPPAAL model creation and verification tool.

2.2.1 Train

The train model represents one train as it is moving through the railway tracks. The 8 rows of states on the UPPAAL model represent the possible paths for the train, with the different intermediary positions on the track, before and after switches.

At the beginning, a train non-deterministically chooses an incoming way and destination.[2] After leaving the system, the train will respawn at the beginning after some time.

When a train approaches a first switch, it is added to the corresponding queue waiting behind the incoming train, the train waiting for the traffic light. In other words, we have a *FIFO* queue of trains waiting behind the train expecting a traffic light switch. Whenever a train enters this incoming state, the controller of the corresponding traffic light is notified and can then change the

[2]When simulating the model, this means that trains will randomly choose a path. For the model verification, it means that properties must hold for possible combinations of choices.

light to allow the train to enter the intersection. The traffic light controller also makes the corresponding switch change position if needed, before lettinng the train pass.

When a train enters an intersection, it allows other train waiting in the corresponding queue to advance a step, and for the one on the top of the queue to exit it, placing itself as the new incoming train.

When a train leaves an intersection, the controller is also notified.

The travel time constraints (minimum and maximum) are present for both travelling through a track segment (from exiting an intersection to arriving at the next one) and for the time needed to go through an intersection itself.

The model is illustrated on Figure 2.

2.2.2 Queue

For each incoming way of a switch, we implement a *First-In First-Out* (*FIFO*) queue in order to maintain the order of arrival of each train at each switch. These are used to handle several trains on a same segment and model the realistic fact that trains cannot overtake one another.

The model is illustrated on Figure 3. It is inspired by the bakery algorithm.

2.2.3 Switch

Based on the two possible outgoing ways, each switch has two settings or positions. These are changed from one to the other whenever the switch receives a synchronization[3] from the corresponding traffic lights controller.

The model is illustrated on Figure 4.

2.2.4 TrafficLight

This part of the model governs the A, B, C and D points. It receives a synchronization from a train whenever it start waiting at a traffic light and whenever it leaves an intersection. At which point the traffic light controller will check each of the switch's 3 incoming queues, and based on that will determine if a train can be allowed to enter the intersection.

First test is to determine if the intersection is free ; then if a train is waiting. The based on the fixed priority that a train leaving the system is prioritized over a train entering it, and that a train coming from the X point should also be (arbitrarily) prioritized over a train in the straight segment of tracks, the next potential entry is determined. If the next train to enter is coming from the outside, the controller needs to check the availability of the next track segment that the train will travel ; this test is based on the presence of a train on said segment and its direction. If no train is present or if this(these) train(s) are going the same direction that our waiting train will do, it should be allowed in the intersection.

Before a train is allowed in the intersection (by changing the traffic light colour), the controller performs a final check of the position of the switch, and changes it if needed.

The model is illustrated on Figure 5.

[3]This is implemented using UPPAAL *channels*.

2.2.5 XTrafficLight

The controller for the X point is a modified and simplified version of the controller for the four corner intersection. It indeed does not need to check the availability of the next track segment, as it was already taken care of by the corresponding corner controller. The set of priorities for this controller is the following : train coming from point A are prioritized over trains from B, themselves prioritized over trains from C which are finally of a higher priority than the trains from D.

These are the two slight changes as the rest of the controller (check if intersection is free, position of the tracks, ...) is the same.

The model is illustrated on Figure 6.

3 Simulation

As part of the *Embedded Systems Design* part of the project, a simulator was written which visually represents the movement of the trains on the rail; based on simulation traces from the UPPAAL model.

4 Verification

The goal of the verification part of this project is to prove the correctness of the model by formally verifying that the system satisfies a set of properties described using temporal logic formulae.

This is done by using the verifier `verifyta`, which is part of UPPAAL. It implements the algorithms to test the timed automaton against a given temporal logic formula. However, the verifier would require too much computation time and memory to run on our original model, so we instead built a simplified version of the model.

4.1 Simplified Model

The simplified model we built for verification differs from the original one in three ways:

- The number of trains in the system is limited to 2. Having more trains requires much more computation time for verification because the number of combined states increases exponentially.
- Trains don't respawn. Instead, they enter the system once, but with a non-deterministic, upper bounded time offset. This decreases the total number of states.
- The update procedures for the traffic lights at the 5 switches are done sequentially (in a fixed order), instead of concurrently. This removes the state explosion created by the possible interleavings of the commited state transitions in `TrafficLight` and `XTrafficLight`, by making their order of execution deterministic. This should not change the properties of the model.

Since this simplified model covers a subset of the original model, we know that if a property is satisfied for the original model, it is necessarily also for this one. If a property is verified for this model, it provides at least a strong hint that it should also be satisified for the original model.

4.2 Properties

We verify four kinds of properties of the system:

Switch safety There must not be more than one train crossing one switch at the same time.

Segment safety There must never be two trains on one segment running in opposite directions.

No deadlock The system is never deadlocked. That is, the two trains will never both wait for the other one.

Train Liveness Any train will always eventually reach its destination.

This section will formalize the properties in times CTL and UPPAAL syntax, and explain why they are satisfied by the model.

4.2.1 Switch safety

These properties require a system with at least 2 trains. With one train they are trivially true. The train instances `Train(1)` and `Train(2)` are used to express the property. If there were more than 2 trains ($\neq$ simplified model), choosing any other pair of train instances would not make a difference because they are all equivalent. In CTL, the property for switch S could be expressed:

$$\forall\Box \quad \neg(\text{train(1).onSwitch(S)} \wedge \text{train(2).onSwitch(S)}) \tag{1}$$

For each switch, there are a number of states in `Train` that correspond to the train currently crossing that switch. The property asserts that trains 1 and 2 are never both in one of these states at the same time.

In UPPAAL requirement specification language, the properties are described as follows:

X : `A[] not( (Train(1).AInX or Train(1).BInX or Train(1).CInX or Train(1).DInX) and (Train(2).AInX or Train(2).BInX or Train(2).CInX or Train(2).DInX))`

D : `A[] not( (Train(1).enteringDA or Train(1).enteringDB or Train(1).AatD or Train(1).BatD) and (Train(2).enteringDA or Train(2).enteringDB or Train(2).AatD or Train(2).BatD))`

C : `A[] not( (Train(1).enteringCA or Train(1).enteringCB or Train(1).AatC or Train(1).BatC) and (Train(2).enteringCA or Train(2).enteringCB or Train(2).AatC or Train(2).BatC))`

B : `A[] not( (Train(1).enteringBC or Train(1).enteringBD or Train(1).CatB or Train(1).DatB) and (Train(2).enteringBC or Train(2).enteringBD or Train(2).CatB or Train(2).DatB))`

A : `A[] not( (Train(1).enteringAC or Train(1).enteringAD or Train(1).CatA or Train(1).DatA) and (Train(2).enteringAC or Train(2).enteringAD or Train(2).CatA or Train(2).DatA))`

Any transition in `Train` that leads into a state that corresponds to the train crossing a switch S; is decorated with a *guard* that checks whether the flag `S_crossing` is not set. When it enters the state, it sets the flag, and when it exits the switch, it unsets the flag again. The flags are global boolean variables, and so they provide mutual exclusion.

4.2.2 Segment safety

These properties assert that no two trains are on the same segment, but running in opposite directions. This is checked for segments A-C, A-D, B-C and B-D. It implies that the property remains true for the sub-segments split by X.

The model is built so that it is possible to have more than one train on the same segment. The invariant order (i.e. the fact that trains cannot overtake each other) of trains on one segment is handled by the queues. It is assumed that the trains themselves avoid collision with others on the the same segment, this is not part of this model.

The `Train` model uses different sequences of states for the two possible directions along each way. Once a train has chosen an incoming way and a target, it is bound to go through that sequence of states. So to verify the property for any segments S-T and T-S, it is sufficient to check that no train enters S-T while there is at least one train on T-S. By symmetry, it is sufficient to check this in one order only, and like for the switch safety, it doesn't matter which two train instances are used.

General formula in CTL:

$$\forall\Box \quad \neg(\text{train(1).onSegment(S-T)} \wedge \text{train(2).entering(T-S)}) \tag{2}$$

In UPPAAL:

B-C `A[] not( (Train(1).enteringBC or Train(1).BCbegin or Train(1).BapproachingC or Train(1).BIncomingC or Train(1).BatC or Train(1).BapproachingX or Train(1).BIncomingX or Train(1).BInX or Train(1).afterXToC) and (Train(2).enteringCB))`

A-D `A[] not( (Train(1).enteringAD or Train(1).ADbegin or Train(1).AapproachingD or Train(1).AIncomingD or Train(1).AatD or Train(1).AapproachingX or Train(1).AIncomingX or Train(1).AInX or Train(1).afterXToD) and (Train(2).enteringDA))`

B-D `A[] not( (Train(1).enteringBD or Train(1).BDbegin or Train(1).BapproachingD or Train(1).BIncomingD or Train(1).BatD) and (Train(2).enteringDB))`

A-C `A[] not( (Train(1).enteringAC or Train(1).ACbegin or Train(1).AapproachingC or Train(1).AIncomingC or Train(1).AatC) and (Train(2).enteringCA))`

Trains enter into a segment when the corresponding traffic light becomes green. In the UPPAAL model, an instance of `Train` enters the commited state before `enteringST` upon synchronization with the channel `A_updated?`. The message is sent by the traffic light controller for switch S. Additional guards

ensure that this does not happen as long as the traffic light is not green or when another train is currently on that switch S.

The system associates a *direction* to each segment, stored in the global variables `ST_dir`. The traffic light controller checks whether there is an incoming train waiting on that switch S, its target, and the current direction for the segment to decide whether to let the train enter. If the segment direction is opposite to the one required, it only lets the train enter when there are no other trains on the segment. Upon entering, the train controller then updates the segment direction. The traffic light controller also handles adjusting the switch setting.

Upon moving on to the state `enteringST`, the train sends out the synchronization message `S_inc_free!`. This pops this train from the queue, and lets the next trains follow.

4.2.3 No deadlock

The *no deadlock* property asserts that there can never occur a situation in the system where no transition can be taken. This would mean that the system has stalled and can no longer control the traffic lights and switches to let new trains through.

In UPPAAL the property is expressed as: `A[] not deadlock`.

This would occur if no transition is available. A transition is available only if its guard condition is satisfied, and if it has a synchronization, both the transition sending out the channel message and the one(s) receiving it must be available. Also for timed automata such (i.e. the `Train` model), a deadlock could occur when a trains waits indefinitely instead of taking the transition. This is avoided by adding *invariants* to the states with a upper bound on the clock. Here, lower clock bounds are can also be part of the guard conditions.

The positive verification of the *no deadlock* property confirms that the model is correct in the sense that the traffic light controllers always let some train through.

4.2.4 Train Liveness

These properties verify that any train entering the system will eventually get through and reach its intended destination.

If these were not satisfied, it could mean that trains can take the wrong path, the system deadlocks, or one train can be blocked by other trains for too long (constrained by the trains' upper time bounds).

Each incoming train goes through a state `StargetT` which indicates that the train comes in from switch S and intends to get T on the other side. After having left the system, it then goes through a state `reachedT`. The formula expresses that if a train is at `StargetT` once, it must in the future get to state `reachedT`. In CTL:

$$\forall\Box \quad (\text{train(1).StargetT} \longrightarrow \forall\Diamond \text{train(1).reachedT}) \tag{3}$$

Properties in UPPAAL:

A → C `Train(1).AtargetC --> Train(1).reachedC`

A → D `Train(1).AtargetD --> Train(1).reachedD`

B $\rightarrow$ **C** `Train(1).BtargetC --> Train(1).reachedC`

B $\rightarrow$ **D** `Train(1).BtargetD --> Train(1).reachedD`

C $\rightarrow$ **A** `Train(1).CtargetA --> Train(1).reachedA`

C $\rightarrow$ **B** `Train(1).CtargetB --> Train(1).reachedB`

D $\rightarrow$ **A** `Train(1).DtargetA --> Train(1).reachedA`

D $\rightarrow$ **B** `Train(1).DtargetB --> Train(1).reachedB`

It can be seen immediately in the `Train` UPPAAL model, that once a train is at state `StargetT`, it can not take any other path but to go towards `reachedT`.

The *no deadlock* property confirms that some train is always getting through (as all other parts of the model will always only take a finite number of transitions when triggered by a synchronization message from `Train`). If a train takes too much time blocked on one state and the upper time bound is no longer satisfied, that time bound could simply be increased. It remains to show that a train can never be blocked indefinitely.

Short step by step analysis of the possible blockings on the state sequence of `Train`.

- Ticket queue on entrance: `S_inc_req!` leads to execution of a finite sequence of states in `Queue`. This will send out the message `A_inc_out!`. There will be one train on that queue whose `ticket` variable is currently at 0, and that one will then get into the state `StargetT`. If another train (in front of it) was currently going through the S switch, the queue does not send out the message yet, but does so when that train left the switch (`S_inc_free!`).
- The train starts crossing the S switch upon synchronization with `S_updated?`, when the traffic light green. This is handled by the traffic light controllers.
- As described, `S_inc_free!` pops the queue and allows the next train to enter. The queue always returns to its idle state, or synchronizes with letting the next train in, after a short sequence of committed states when it has gotten the message, so this never blocks.
- The same logic applies for passing through the X crossing (when needed), and leaving through the T switch.
- The `moved!` (`updated!` on the original model) message makes all the traffic lights controllers update. This is always a finite sequence of committed states that can never block. It may also make switches change setting, and it verifies that this is possible (i.e. the switch is not already at the required position).
- When leaving through T, the system does not check that there is no incoming train waiting at T. That such a situation should not occur is part of the external dependencies of the system. If the model is to verify this, additional constraints would need to be added to the choice of incoming trains and their destinations, otherwise reachability would not be satisfied. This is why in section 1, the incoming and outgoing ways are represented as separate tracks.

5 Conclusion

The verified properties confirm that the system works as intended, that allows incoming trains pass through to get to their intended destination within a time limit, without deadlocking or causing collisions. This is proven for a simplified version of our original model, which has two trains pass through only once (but with in any combination and with any time offset).

Our simulation made for the *Embedded Systems Design* part shows how the original system also appears to function correctly.

5.1 Possible improvements

In its current state, the controller part of the model uses fixed priorities and does guarantee that the trains will reach their destination. It would however be more efficient to design a more adaptive and dynamic way of prioritizing access to the critical sections.

To achieve this goal, a partial redesign of both the traffic lights entities would be required. The main idea would be to optimize the number of track switch required by biasing the traffic light towards trains need the current switch set-up. This would however require some memory usage to keep track of the waiting time of trains ready to enter the intersection, in order to prioritize trains that have been waiting for too long as to guarantee no starvation.

Another potential improvement would be to condense both traffic lights entities by regrouping states that follow one another and the condition on their edges. This could boost the speed of the automatic verification tool (due to the reduced amount of states) while unfortunately reducing the readability of the model.

Appendices

A Figures

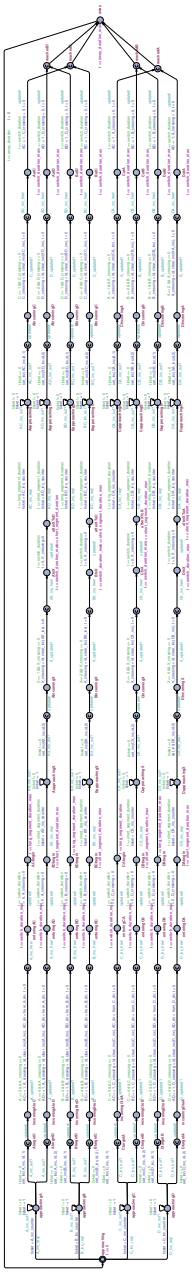


Figure 2: UPPAAL model : Train

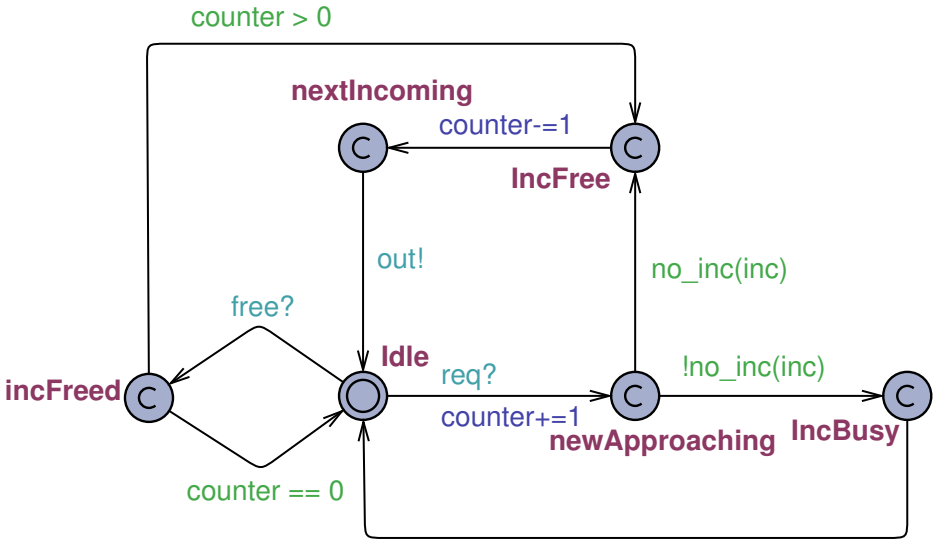


Figure 3: UPPAAL model : Queue

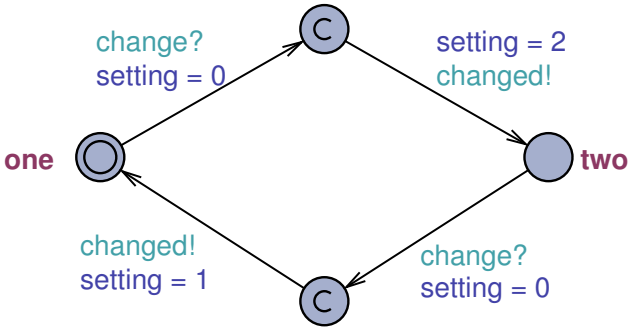


Figure 4: UPPAAL model : Switch

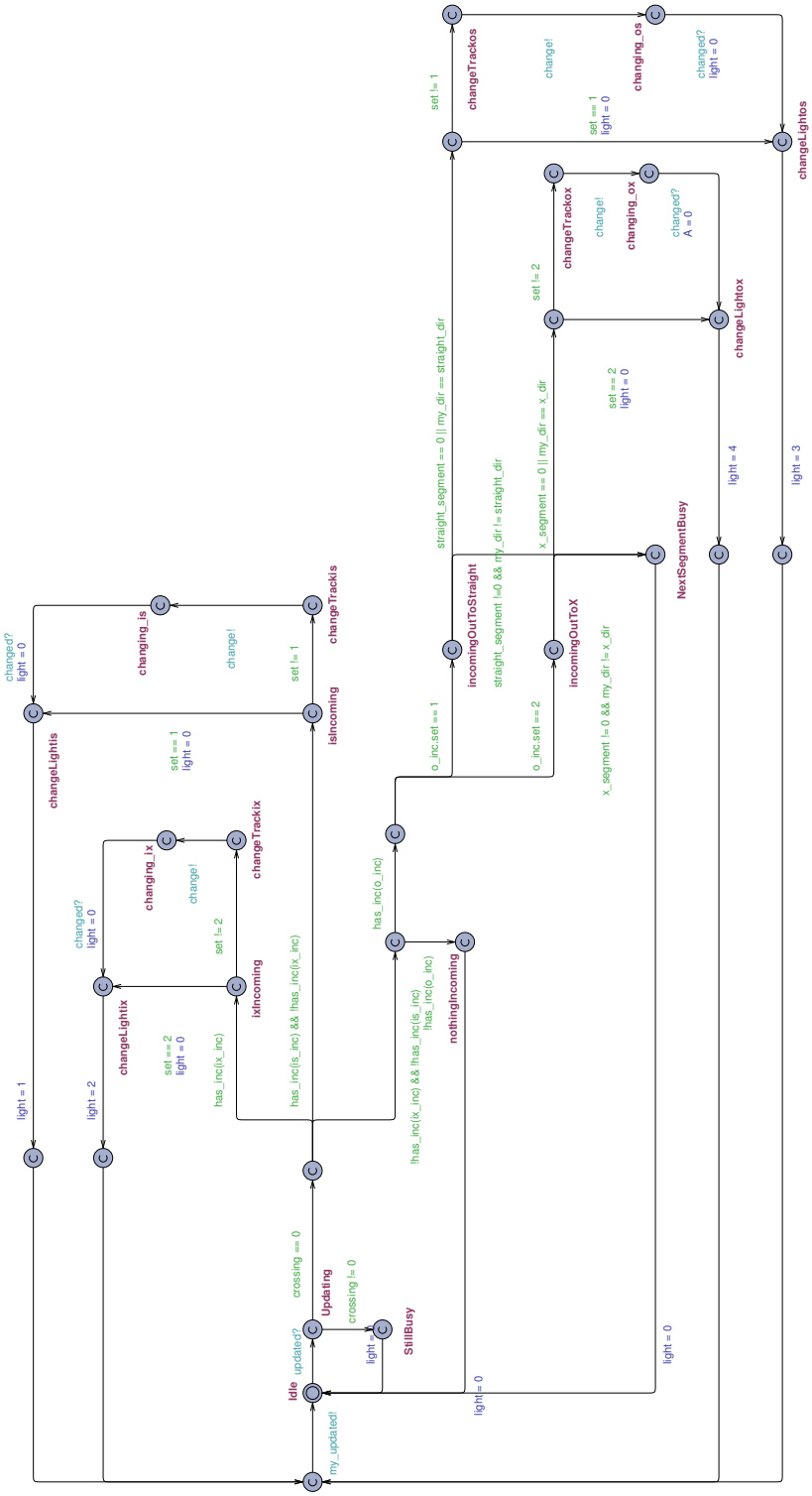


Figure 5: UPPAAL model : Traffic light

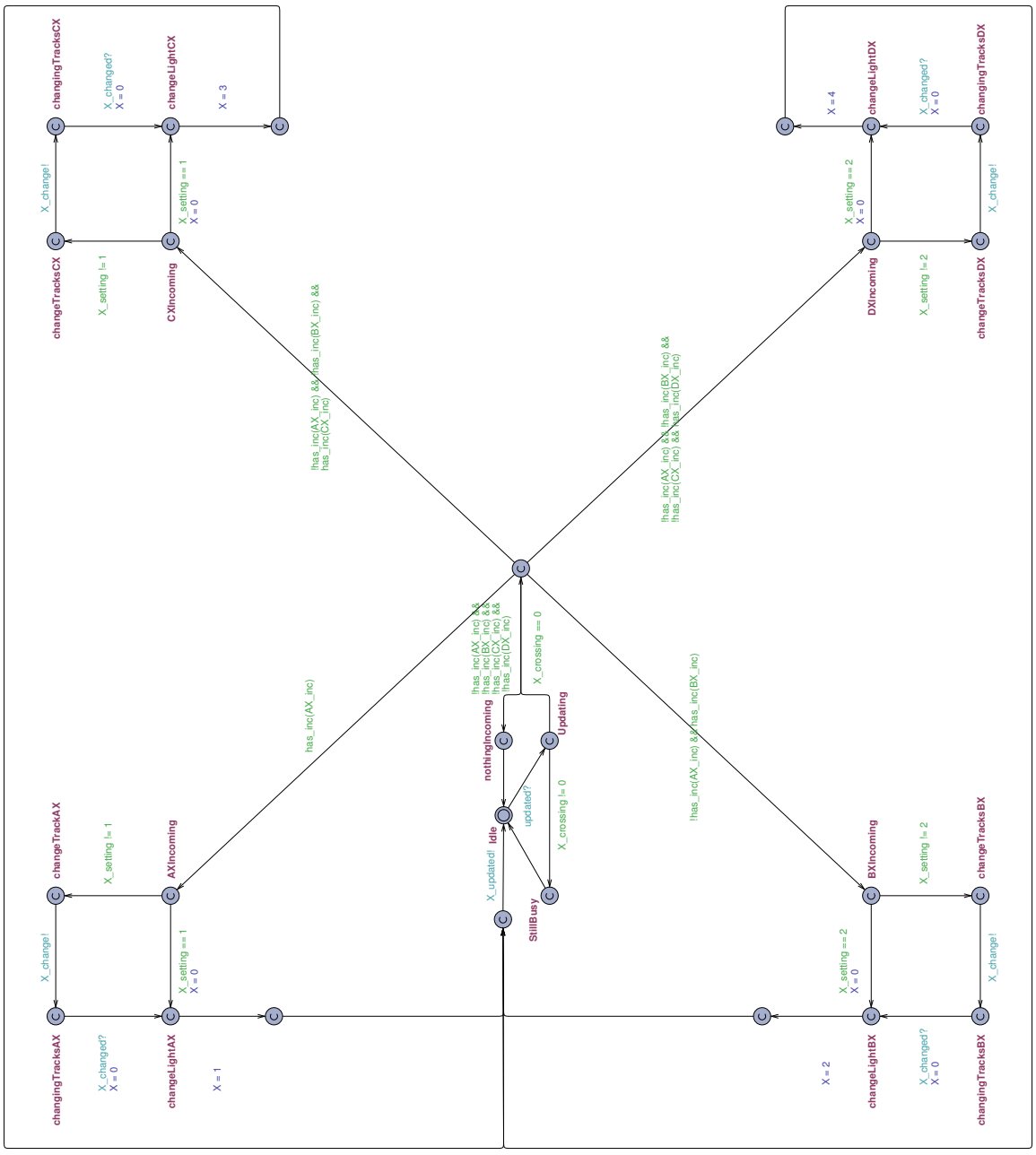


Figure 6: UPPAAL model : Traffic light for switch X