Formal Digital Twin of a Lego Mindstorm production plant

Fernando Morea

Matricola 224189 - Codice Persona 10707314

Contents

1 Design **3**
1.1 Design assumptions . . . 3

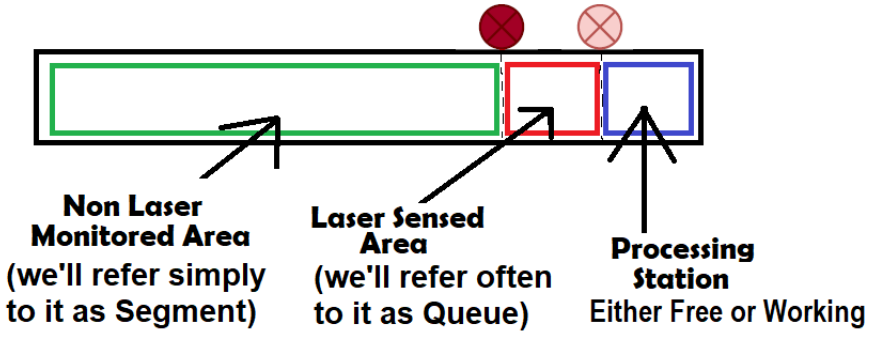
2 UPPAAL Model **4**
2.1 Global declarations . . . 4
2.1.1 Constants of the system . . . 4
2.1.2 Variables and channels . . . 4
2.2 Templates . . . 5
2.2.1 ProcessingStation . . . 5
2.2.2 Piece . . . 5
2.2.3 LaserSensors . . . 6
2.2.4 ConveyorBelt - The centralized controller of the Belt . . . 7
2.3 Deadlocks and Conflicts between pieces . . . 7

3 Flow controller proof-of-concept **8**

4 Formal verification in TCTL Logic **9**
4.1 No deadlock property . . . 9
4.2 Home-state property . . . 9
4.3 Piece location is always tracked . . . 9
4.4 No queue ever exceed the maximum allowed length . . . 9
4.5 It never happens that a station holds more than 1 piece . . . 10
4.6 It never happens that two pieces occupy the same belt slot . . . 10
4.7 Tested Configurations - Conclusions . . . 10

1 Design

A recurring pattern has been identified in the structure of the Conveyor Belt that allows for a well-defined topological ordering of its components, meaning that the elements can be numbered in a clear and consistent way. This is important for ensuring that the Conveyor Belt operates correctly and that pieces can be easily identified and tracked as they move through the system.

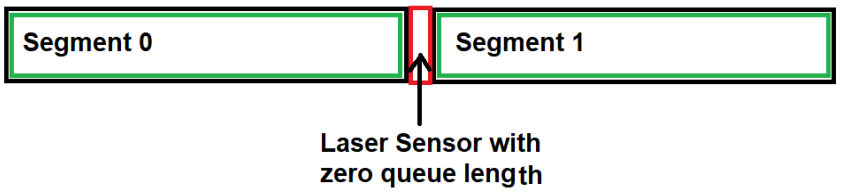


1.1 Design assumptions

A good assumption to be made for our system is the following:

> **A1: If there is a Processing Station there must be a Laser Sensed Area preceding it**

Think about it, there is the need to physically block pieces: as Segments, in compliance with the specifications, are assumed always with a certain non-null constant speed. As can also be observed from the demonstration video, the motors of the different segments of the Lego Mindstorms never stop; there must be a mechanical actuator at the end of the segment's run that physically locks the pieces in certain situations (Is the station free or not? –> Structural Hazard).



For the same reason, namely the need for a mechanical actuator that physically moves the piece from one segment to another (and to simplify the model), we **always** consider a queue of length 0 in our model to **connect two adjacent segments**.

So in our simplified model:

> ▷ Laser Sensor with queue length == 0 => **There is no station**

So after the Assumption A1 and the explaination of this particular case, we derive the requirement for our model:

> **R1: The number of Laser Sensed Area is exactly equal to the number of Segments (at the end of a Segment there is always a Laser Sensed Area to connect it)**

Then we can reason on the Processing Stations:

> **A2->R2: The number of Processing Stations is either less than or equal to the number of Segments.**

Basically the blue box in the first drawing, the Processing Station, may or may not be there.

By juxtaposing **blocks** (Segment, Queue, and optionally a Processing Station), any geometry can be obtained.

2 UPPAAL Model

2.1 Global declarations

2.1.1 Constants of the system

Constant	Example Initialization
NUM_PIECES	5
NUM_SEGMENTS	2
SEGMENT_LENGTH[NUM_SEGMENTS]	{9,9}
MAX_SEGMENT_LENGTH	9
MAX_QUEUE_BUFFER	10
INITIAL_QUEUE_BUFFER[MAX_QUEUE_BUFFER]	{-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}
MAX_QUEUE_LEN[NUM_SEGMENTS]	{2,3}
PROCESSING_TIME[NUM_SEGMENTS]	{2,2}
PIECE_SPEED	1
policy	0

Table 1: Constants of the system

NUM_PIECES defines the number of the moving pieces in the conveyon belt
NUM_SEGMENT defines the number of segments of witch the conveyon belt is composed
SEGMENT_LENGTH is the vector containing the length of each segment
MAX SEGMENT_LENGTH is the length of the maximum size segment in the system
MAX_QUEUE_BUFFER is the length of the buffer used by each queue, so should be at least big enough to contain the maximum size queue of the system
INITIAL_QUEUE_BUFFER is the initial state of the buffer used by each queue, his content should be a vector of size specified by MAX_QUEUE_BUFFER and filled with -1 (-1 means free space).
MAX_QUEUE_LENGTH is a vector that represent the length of each queue
PROCESSING_TIME is a vector representig the processing time needed for each station
PIECE_SPEED encode the speed (in slot/time unit) of every segment
policy is a constant to easily change the flow control policy

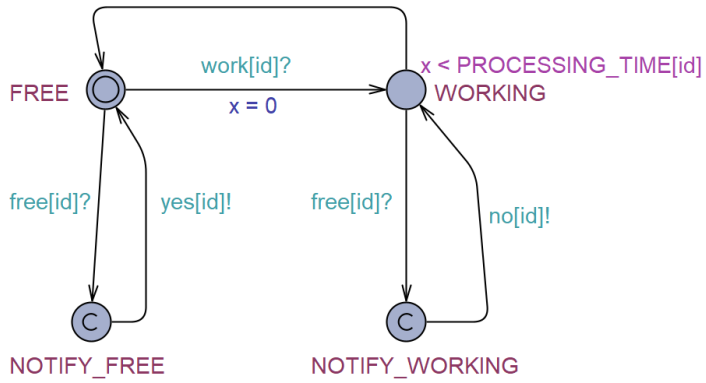
2.1.2 Variables and channels

Code	Description
typedef int [-1, 2*MAX_SEGMENT_LENGTH] limited_int;	Helps limiting the state space
typedef int[0, NUM_PIECES-1] piece;	Useful limitation
typedef int[0, NUM_SEGMENTS-1] segment;	Useful limitation
typedef int[0, NUM_SEGMENTS-1] queue;	Useful limitation
typedef int[0, NUM_SEGMENTS-1] station;	Useful limitation
chan free[NUM_SEGMENTS];	Is the station free?
chan full[NUM_SEGMENTS];	Is the queue full?
chan onTop[NUM_SEGMENTS];	Is the current piece on top?
chan yes[NUM_SEGMENTS];	yes answer
chan no[NUM_SEGMENTS];	no answer
chan addToQueue[NUM_SEGMENTS];	belt -> queue
chan removeFromQueue[NUM_SEGMENTS];	belt -> queue
chan work[NUM_SEGMENTS];	belt -> station
chan isPiece[NUM_PIECES];	belt -> piece
chan move[NUM_PIECES];	belt -> piece
chan localize[NUM_PIECES];	belt -> piece

Table 2: Summary of code

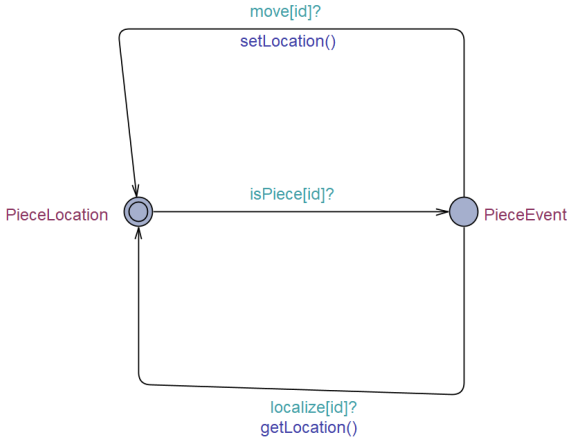
2.2 Templates

2.2.1 ProcessingStation



This is a classic timed automaton that precisely models the station's behavior, which can be in either the free or working state. Additionally, it features a communication mechanism with the ConveyorBelt, which allows the latter to inquire about the station's state using a notification mechanism.

2.2.2 Piece



A similar approach was used to model the timed automaton of a piece. This automaton serves the sole purpose of storing the last known position of the piece on the conveyor belt, and it features a communication mechanism that closely resembles the one employed by the ProcessingStation.

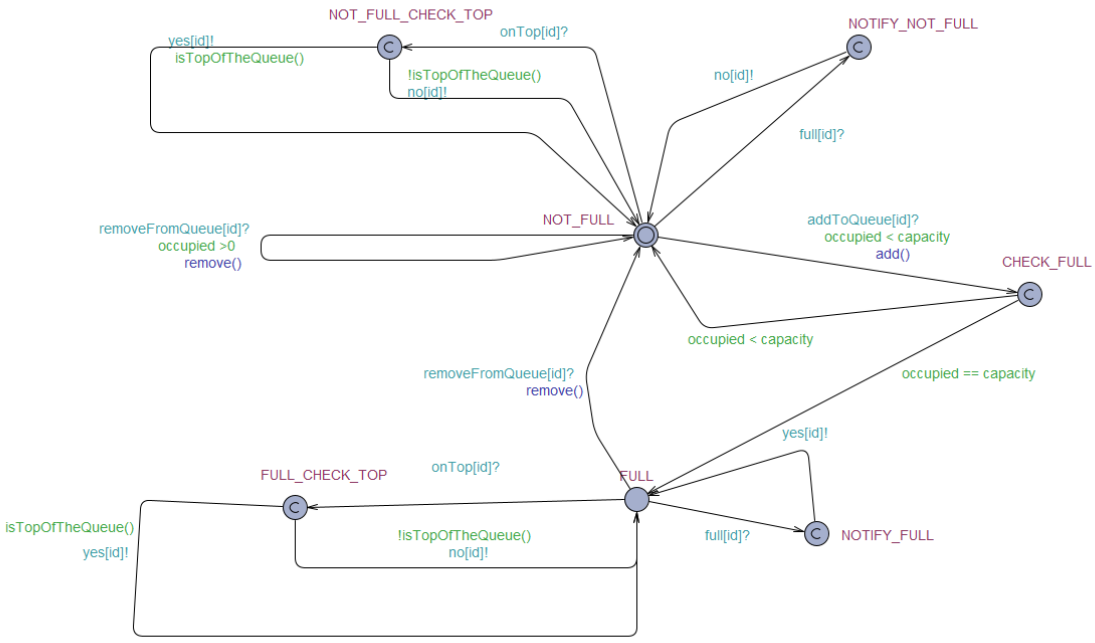
This template also has local declaration, in particular 4 integers to store the location of the piece.

Note that **id** is the identifier of the piece.

A piece can be queried by the main ConveyorBelt controller, and it moves from the state **PieceLocation** to **PieceEvent**.

When the piece is on the PieceEvent state, it can either update the current position or get localized and send the current position to the controller.

2.2.3 LaserSensors



The laser sensor has also been modeled as a timed automaton, which is a bit more complex than the ones seen previously. It needs to manage its queue, which can be either full (**FULL** state) or not full (**NOT_FULL** state), and thus requires a communication mechanism to notify the conveyor belt of its state changes. It also needs to provide a mechanism for the conveyor belt to request information on whether a piece is at the top of the queue (**FULL/NOT FULL CHECK_TOP** states).

The code in the local declatation of this template defines a queue buffer with a maximum length of MAX_QUEUE_LEN[id] and an initial buffer INITIAL_QUEUE_BUFFER.

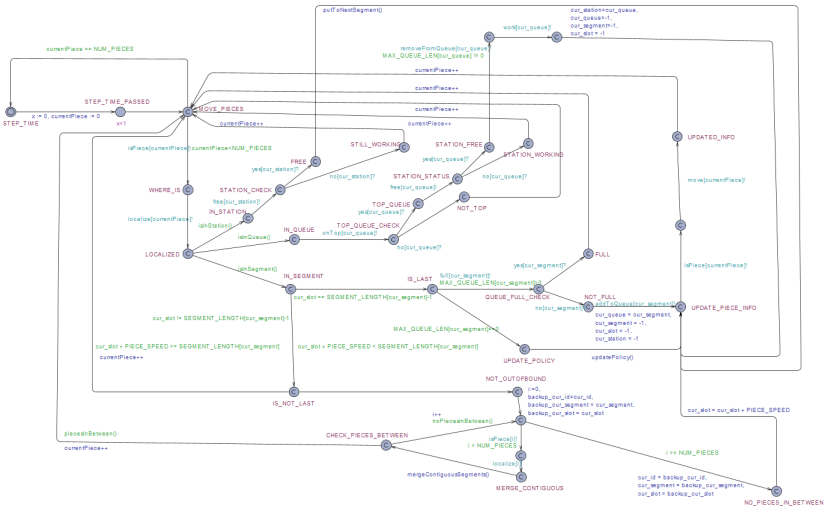
The **add()** function adds a piece, identified by cur_id, to the end of the queue buffer and increments the occupied counter.

The **remove()** function removes the first piece in the queue buffer and shifts the remaining elements to the front.

The **isInBuffer()** function checks whether the piece, identified by cur_id, is already present in the queue buffer and returns its position.

The **isTopOfTheQueue()** function checks whether the piece, identified by cur_id, is at the front of the queue buffer and returns true if it is, otherwise false.

2.2.4 ConveyorBelt - The centralized controller of the Belt



This timed automaton describes the behavior of the Conveyor Belt. To better understand its operation, we can divide it into two main sections. In the upper left corner there are two states, **STEP_TIME** and **STEP_TIME_PASSED**, which describe the **temporal behavior** of the machine, i.e., the passage of time. The committed states, on the other hand, describe the **functional behavior** of the conveyor belt (i.e., advancing the pieces).

The use of committed states allows us to see the movement of pieces as a **single atomic sequence**, so that when the automaton returns to the STEP_TIME state, the position of the pieces is unambiguous (i.e. there are no pieces in the same slot, or no piece in more than one slot).

To move a piece, it must first be located, and based on its location, decisions must be made. In particular, if it is in a segment, it must be determined whether it is at the end of the segment (state **IS_LAST**), and therefore, it must be able to jump to a queue or possibly to another segment (with a well-defined policy, for now, consider the default policy, i.e., jump to the adjacent segment with the next id). If it needs to jump to a queue, it is necessary to check if the queue has not already reached its maximum capacity.

Suppose instead that the piece is not in the last slot of the segment. We need to check whether there are pieces between its slot and the one in witch it wants to jump to (i.e., the one "PIECE_SPEED" further ahead). To do this, we use the state **CHECK_PIECES_BETWEEN** that check if there are no pieces in between (it is a little bit more complicated than this simplified explaination, because the case of two adjacents segments has to be considered, with the **MERGE_CONTIGUOUS** state, and decisions have to be taken based on this predictive analysis).

In case the controller wants to move a piece located in a queue, it must be the top of the queue (state **TOP_QUEUE_CHECK**).

To move a piece from/to the station, it is necessary to evaluate its state (is it free? no? -> Structural Hazard). To do this, additional states are needed, including **STATION_CHECK**, **STATION_STATUS**, **STATION_FREE**, and **STATION_WORKING**.

2.3 Deadlocks and Conflicts between pieces

This model, although very complex and articulated, is free from the possibility of having deadlocks. This is because, if attention is paid to how it is constructed, any anomalies detected during the movement of a piece would still return the automaton to the **MOVE_PIECES** state (one could think of this state as the so-called **sink** of finite state automata).

Regarding the issue of access conflicts to resources (in this case slots, queues, and stations) by the pieces, the problem has been solved using the **atomicity** offered by the "committed states" feature of UPPAAL. With a transactional mechanism, either the transaction occurs and the piece moves (state **UPDATE_PIECE_INFO**), or the system state is not modified, so that when the automaton returns to the STEP_TIME state, a consistent situation is obtained (no duplicate pieces or pieces in the same place, or similar anomalies).

3 Flow controller proof-of-concept

The default policy (triggered with policy = 0 in global declarations) is this one:

▷ **Branch always to the next contiguous segment**

So simply ignore any kind of geometry and always route pieces on a default segment.

```
1  // ConveyonBelt Template declarations
2  // ...
3  void updatePolicy(){
4  if (policy == 0){
5          // if it is in the last segment
6          // then jump to the first
7      if (cur_segment == NUM_SEGMENTS-1){
8              cur_segment = 0;
9          }
10         else{
11         cur_segment = cur_segment + 1;
12         }
13     cur_slot = 0;
14     }
15 }
```

Any kind of special branch policy can be created.

As a proof of concept, if we want to recreate the topology given in the requirement pdf, we can make the following modifications creating this new static policy:

▷ **If a piece has an even identification number then goto alternative destination, else goto default route (branch to the next contiguous segment)**

```
// Global declarations
+const int source = 3
+const int alternativeDestination = 5
...
-policy = 0
+policy = 1

// ConveyonBelt Template declarations
// ...
void updatePolicy(){
...
+if (policy == 1){
+       if(cur_segment == source){
+               if (cur_id % 2 == 0){
+                       cur_segment = alternativeDestination;
+               }
+               return;
+       }
+        // default route
+       // in the case of 3 and cur_id odd then 3 + 1 = 4
+    if (cur_segment == NUM_SEGMENTS-1){
+            cur_segment = 0;
+         }
+         else{
+         cur_segment = cur_segment + 1;
+         }
+    cur_slot = 0;
+    }
}
```

4 Formal verification in TCTL Logic

Keep in mind that by increasing the topological complexity of the system, you might cause State Space Explosion.[1]. Also the available RAM memory and the processor of your system could play an important role[2].

4.1 No deadlock property

▷ **A[] !deadlock**

For all possible paths in the system, there is never a state where a deadlock is reached.

4.2 Home-state property

▷ **E[] ConveyorBelt.MOVE_PIECES imply ConveyorBelt.STEP_TIME**

If the ConveyorBelt is in the MOVE_PIECES state, then it will eventually move to the STEP_TIME state. this property checks if the ConveyorBelt is always able to move its pieces and eventually reach the STEP_TIME state.

This property is guaranteed to be verified as we explained in section 2.3.

4.3 Piece location is always tracked

▷ **E[] forall(i:piece) ConveyorBelt.STEP_TIME imply Piece(i).PieceLocation**

There exists a path in the system where for all pieces, if the ConveyorBelt keeps moving according to the STEP_TIME, then the location of the piece is tracked.

4.4 No queue ever exceed the maximum allowed length

▷ **A[] forall (i : queue) (LaserSensors(i).capacity >= LaserSensors(i).occupied)**

For all queues i, the capacity of LaserSensor(i) must be greater than or equal to the number of occupied slots in LaserSensor(i) at all times.

This property ensures that there are no queues that are over capacity, i.e., having more pieces in the queue than the number of available slots in the corresponding LaserSensor.

[1] The State Space Explosion is a common problem in the analysis of complex systems such as those modeled with finite state automata. It occurs when the number of possible states of the system grows exponentially with the number of state variables and possible transitions. This makes checking for Always true properties very difficult, if not impossible.

To overcome this problem, it is possible to use symbolic model checking techniques, such as inductive property checking or SAT/SMT based model checking. Alternatively, abstraction techniques can be used to reduce the size of the state space, such as limiting the number of transitions allowed in each state or abstracting some state variables.

However, even with these techniques, checking Always properties can be difficult or impossible on very complex systems. In these cases, it may be necessary to limit oneself to verifying the properties on a finite subset of the states, or to carry out a verification based on simulation and testing, accepting the risk of not finding any rare but possible counterexamples.

[2] Our system consist of 24 GB RAM DDR4, clocked at 2400 mhz, an Intel Core i5-11400H @ 2.70GHz

4.5 It never happens that a station holds more than 1 piece

Reasoning about our model, the station can be only free or working, so no information about quantity of pieces is stored (if the station is working it is implicitly working only one piece).

We can instead verify if more than two pieces reports to be in the same station.

▷ **A[] forall (i : piece) forall (j : piece) ConveyorBelt.STEP_TIME && Piece(i).isInStation() && Piece(j).isInStation() imply Piece(i).mystation!=Piece(j)**

4.6 It never happens that two pieces occupy the same belt slot

▷ **A[] forall (i : piece) forall (j : piece) ConveyorBelt.STEP_TIME && Piece(i).isInSegment() && Piece(j).isInSegment() && Piece(i).mysegment == Piece(j).mysegment imply Piece(i).myslot != Piece(j).myslot**

4.7 Tested Configurations - Conclusions

Three configurations have been chosen and tested in the verification process:

▷ Simplest configuration: 3 pieces and 2 segments -> verification success in seconds. Too trivial...

▷ Let's try to increment segments keeping the same number of pieces: 3 pieces and 5 segments -> verification success in 30 seconds ÷ 5 minutes. I'll keep this configuration in the config file.

▷ Middle size configuration: 5 pieces and 5 segments -> Probabily hours, untill the RAM saturates.

So the difficult thing for the model seems the number of pieces to keep track of. between 3 and 4 seems easy, from 5 and upper seems difficult.