POLITECNICO
MILANO 1863

Formal Methods for Concurrent and Real-Time Systems

-

Formal Digital Twin of a Lego Mindstorms Production Plant

Giacomo Dell'Agosto
10658977

Giovanni Alberto Sartorato
10678032

Andrea Zanella
10616046

Ettore Zamponi
10808192

July 2023

Contents

1 High Level Description **3**

2 Components **4**
2.1 Internal representation . . . 4
2.2 Synchronizer . . . 4
2.3 Conveyor Belt . . . 5
2.4 Processing Station . . . 5
2.4.1 Deterministic . . . 5
2.4.2 Stochastic . . . 5
2.5 Station Sensor . . . 6
2.5.1 Deterministic . . . 6
2.5.2 Stochastic . . . 6
2.6 Queue Sensor . . . 6
2.6.1 Deterministic . . . 6
2.6.2 Stochastic . . . 7
2.7 Control Station . . . 7

3 Design Assumptions **7**
3.1 System Speed . . . 7
3.2 Processing Station Check . . . 7
3.3 Scalability Array . . . 8
3.4 Sensor Position . . . 8

4 System configurations **8**
4.1 First Configuration . . . 9
4.2 Second Configuration . . . 9
4.3 Third Configuration . . . 9
4.4 Branch switching policy . . . 9

5 Analysis **10**
5.1 Deadlock . . . 10
5.2 A station never holds more than one piece . . . 10
5.3 Two pieces never occupy the same belt slot . . . 10
5.4 No queue ever exceeds the maximum allowed length . . . 10
5.5 Policy 2 check . . . 10

6 Conclusion **11**

List of Figures

1 Belt segments id enumeration 4
2 Synchronizer 4
3 Belt 5
4 Deterministic processing station 5
5 Stochastic processing station 6
6 Deterministic processing station 6
7 Stochastic processing station 6
8 Deterministic queue sensor 7
9 Stochastic queue sensor 7
10 Control station 7

1 High Level Description

The model representing the *Lego Mindstorms Production Plant* is composed by six main components:

- *Synchronizer*
  Central component that broadcasts synchronizing signals for the whole model, allowing everything to operate at the right time. It uses multiple synch signals because some components need to update before others (for instance, the sensors have to update first);

- *ConveyorBelt*
  This component is responsible for advancing the individual parts on the conveyor belt. Also, it connects the different segments by moving pieces from one conveyor segment to another when there is a piece on the last slot of the segment;

- *ProcessingStation*
  Fundamental component that simulates the processing phase of the workpiece and moves the pieces on the station's conveyor belt. The parameters define the station id (*s_id*), the processing time (*pt* or *mean* and *dev* for the stochastic version) and position relative to belt segment of the respective id (*pos*);

- *StationSensor*
  Laser sensor that recognizes when there is a piece on the last slot of the respective belt and, if the station is not busy, sets the *pieceAvailable* variable for that station true. The parameters allow to choose which station (*s_id*) and which belt (*b_id*) it is relative to;

- *QueueSensor*
  This template handles any queue created by the Station Sensor due to the arrival of multiple pieces at the same point on the belt by changing to false the *canRelease* variable of the previous station. The parameters allow to choose which station (*s_id*) and which belt (*b_id*) it is relative to, the position on the belt (*pos*) and if it's active or not since it has to be possible which ones installed;

- *ControlStation*
  Finally, it allows to choose the policy of the flow controller where the belt forks (after segment 2), setting the logic for how to distribute the pieces between 2 conveyor belts.

2 Components

2.1 Internal representation

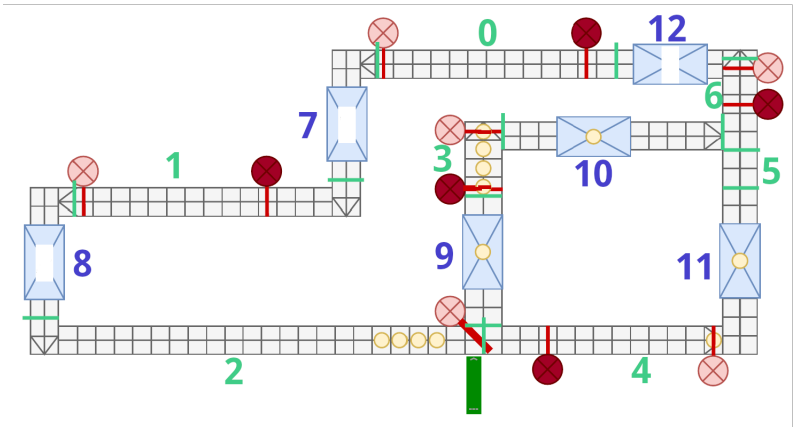

Figure 1: Belt segments id enumeration

The system consists of 13 belt segments, 6 of which are assigned a processing station on a specific slot. The position of the pieces on the belt is stored as a boolean variable in a matrix, with one row for each belt segment and enough columns to store the longest segment. The structure of the whole conveyor belt is described by the array *next*, which stores the id of the next piece for each belt segment. Other arrays are used to store the state of the processing stations, for example whether they are ready to accept a new piece or whether they are allowed to release a piece.

The system could be divided into two main macro-categories, each one characterized by multiple components: working stations and flow components. The working stations are composed of the processing stations and their respective belts and manage the process that allows each piece to be worked properly. The flow components, instead, connects and coordinate the flow of each piece from one station to another. This category is composed by belts and presence sensors.

2.2 Synchronizer

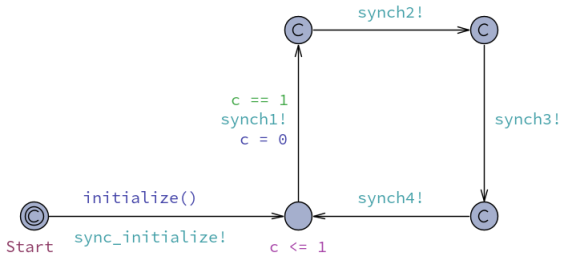

Figure 2: Synchronizer

The Synchronizer is the core of the system. In fact, through this template it is possible to synchronize the various components at each time instant, setting in sequence the broadcast channels *synch*1, *synch*2, *synch*3 and *synch*4. In particular, the synchronization signals happens instantaneously, property enforced through the choice of the committed synchronization states that stops the time flow to let the system evolve synchronously.
Every template is governed by the four synchronization signals, in particular in the deterministic version:

- The control station, station sensors and queue sensors synchronize on *synch*1.
- The belt synchronizes on *synch*3.
- The processing stations synchronizes on *synch*2, *synch*3 and *synch*4.

while in the stochastic version:

- The control station and station sensors synchronize on *synch*1.
- The belt synchronizes on *synch*2.
- The queue sensor synchronizes on *synch*3.
- The processing stations synchronizes on *synch*1, *synch*2 and *synch*3.

Moreover, the Synchronizer allows to properly select the initialization settings of the system through the function *initialize*() that displays the pieces on every belt depending on the chosen configuration.

2.3 Conveyor Belt

Figure 3: Belt

This simple template moves the pieces on the belt every *synch*3 broadcast through the *move*() function. First of all, the function moves the pieces that need to go on the next belt, then the pieces that need to go forward on the same belt, then if the segment is at a confluence point the *next*[] array gets scanned to look for the next previous component with a piece available.

2.4 Processing Station

2.4.1 Deterministic

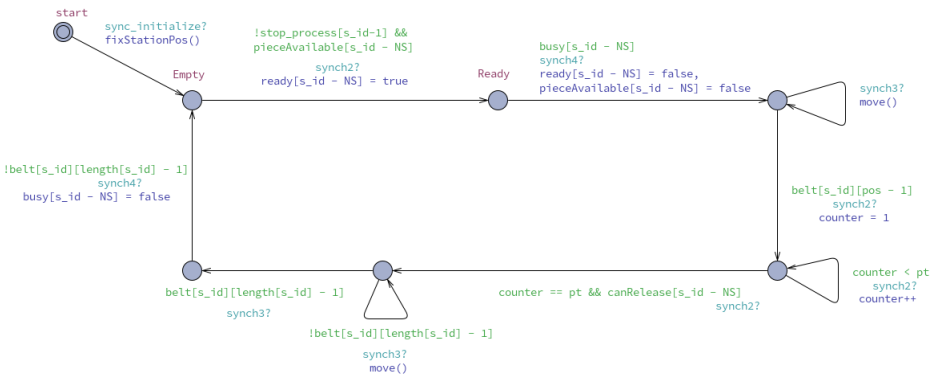

Figure 4: Deterministic processing station

This template manages the processing station and its belt segment. If there isn't a stop process signal for the station and there is a piece available, the station becomes ready. After that it sets itself as busy and starts moving the piece until the station slot on the belt. To simulate the processing time the piece gets held until the counter reaches *pt*. The piece then gets moved to the end of the belt, when it exits the segment the stations set the busy flag as false.

2.4.2 Stochastic

The stochastic version works a little differently. It starts by moving pieces until they are in the slot before the station. It then select a random processing time from a normal distribution with *select_pt*(). After the piece is held *pt* times and the released flag for the station is set to true, the piece gets moved.

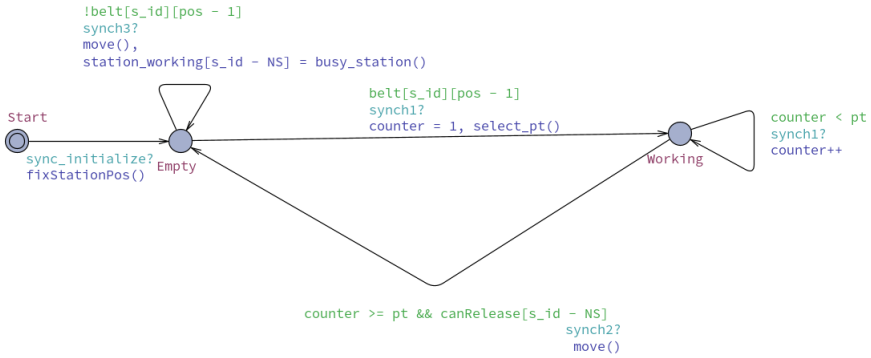

Figure 5: Stochastic processing station

2.5 Station Sensor

2.5.1 Deterministic

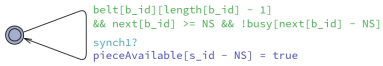

Figure 6: Deterministic processing station

Template of the station sensors associated with every processing station. If there is a piece on the belt on the slot of the sensor and the next station is not busy, it sets *pieceAvailable* for the station as true.

2.5.2 Stochastic

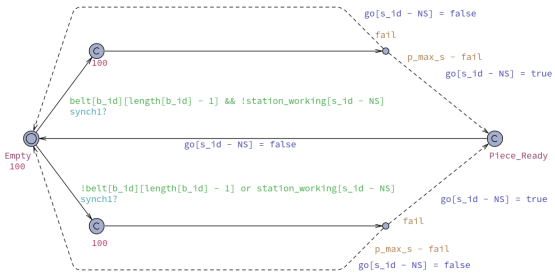

Figure 7: Stochastic processing station

In the stochastic model, if there is a piece on the sensor slot and the station is not already working, the sensor should set the variable in the array *go*[] relative to the station as true, to signal that there is a piece ready to go into the station belt. On the other hand, if there's no piece on the slot or the station is already working the variable in *go*[] should be set to false. If there is a fail in the sensor (with the probability given as a parameter), the opposite happens.

2.6 Queue Sensor

2.6.1 Deterministic

The queue sensor has a Boolean parameter active so in the declaration it is possible to choose which one to turn off. If it's active and there's a piece on its slot the *canRealease* variable of the previous station gets set to false because the queue is full, otherwise it gets set to true.

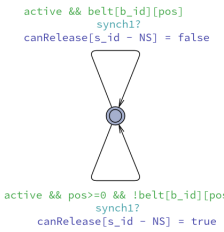

Figure 8: Deterministic queue sensor

2.6.2 Stochastic

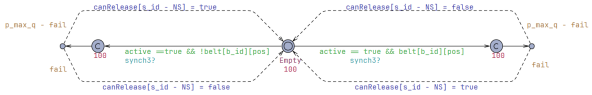

Figure 9: Stochastic queue sensor

The stochastic version does the same check but if it fails *canRelease* of the previous station gets set with the opposite value of what it should.

2.7 Control Station

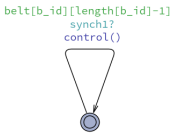

Figure 10: Control station

The control station manages where to send a piece at the fork. When there is a piece on its slot (so the end of the belt) it decides where go next based on the chosen policy (described in section 4.4) by changing the *next*[] variable of the its belt. It works the same way for the deterministic and stochastic version.

3 Design Assumptions

3.1 System Speed

In order to be able to move the workpieces on the belts efficiently, it was decided to exactly match the belt feed rate with a single clock cycle.
Moreover the belt speed is always greater than the station processing speed, resulting in a well-proportioned total time.
Thus, in this model, time is interpreted in a discrete manner and closely linked to the movement of the belts, and thus, of the pieces on it.

3.2 Processing Station Check

The positions of the processing stations are fixed and cannot be modified during the execution of the simulation. However, this does not prevent attempts to modify the positions of the stations within the belts.
Inside the processing station component, it has been decided to implement a small check to ensure that the actual position of each station is not the last slot of the relative belt on which it rests.

```
// Processing station position check algorithm
void fixStationPos() {
    int thisStationLenght;
    int j;
      for ( j = 0; j <NS+NP; j++) {
        //cycle untile next[j] is the station's belt
        if (next[j]==s_id) {
            thisStationLenght = length[next[j]];
            //check if the station is in the last belt slot or further
            if (pos >= thisStationLenght) {
                //move staition in second-last spot
                pos=thisStationLenght-1;
            }
        }
    }
}
```

Indeed, having to represent the production plant as realistically as possible, it was figured out that a processing station cannot physically be placed at the end of a belt. In fact after being processed, in case of a unique conveyor belt, the exiting piece would not have a slot to occupy in order to proceed along the belt or, in case of multiple connected belts, there would not be the minimum space available to connect two different belts.
Doing this, it is increased also the scalability of the project being modifiable without any problems of positioning having this check for all the new possible processing station.

3.3 Scalability Array

In favour of scalability and possible future implementations, it was decided to parameterise as much as possible all the data required for the simulation.
In both models, stochastic and non-stochastic, variables and arrays have been declared in such a way that adding a new belt rather than new stations or sensors remains very simple and ready to use by anyone.

```
// processing stations processing time
const int processing_time[NP] = {7,8,6,5,7,8};
// processing stations location on belts
const int P_pos[NP] = {4, 4, 3, 4, 4, 3};
// standard deviation
const double dev[NP] = {0.1, 0.1, 0.1, 0.1, 0.1, 0.1};
//queue sensors position
const int qs_pos[NP] = {2, 10, 4, 1, 3, 3};
//choose if queue sensor is active
const bool qs_active[NP] = {true, true, true, true, true, true};
```

3.4 Sensor Position

A design choice to speed up execution and avoid unexpected errors is the positioning of queue sensors. This is positioned at a distance that allows to a single piece to slide from the station to the next sensor before another piece is available. So in other words, every queue sensor must be close to the relative processing station.
Otherwise pieces would come out too quickly from a station before the tail sensor is activated, and there would be more pieces than those that should be on the belt, preventing the overflow.

4 System configurations

The system has been simulated in 3 different configurations that approach real life scenarios. To simply switch from one configuration to another we decided to simply comment and uncomment the respective blocks in the project declaration file.

4.1 First Configuration

This is a standard configuration that simulates a full functional system. All the pieces starts from the first belt and the belt speed is fixed to 2 slots over time unit.

4.2 Second Configuration

This configuration simulates the functioning of the system after the resolution of a fault (so, the system has half worked pieces in some belts and the initial set of pieces is not only in the first belt). The belt speed has been increased up to 5 steps over time unit.

4.3 Third Configuration

This configuration simulates a real case scenario in which the position of the queue sensors is poorly chosen, in this case the query related to the number of pieces circulating a belt is not satisfied. In particular through this configuration is demonstrated that this query does not depend on the code itself but only on the physical displacement of the sensors in the system.

4.4 Branch switching policy

The 3 branch switching policies implemented are:

- alternate between the 2 branches (waiting for a piece to be available) if they are available
- take always the external loop (always station 5 over stations 3 and 4)
- take always the internal loop (always stations 3 and 4 over station 5) except if station 3 is occupied, then take the outer loop.

5 Analysis

Queries

Queries refer to the properties or assertions that are checked against the system model to verify its correctness.

5.1 Deadlock

In the deterministic model the absence of deadlocks can easily be verified with the query:

A[] not deadlock.

However, the keyword *deadlock* is not supported in SMC, and another method was used.
It was preferred to analyse the results of this query by means of a simple python script, which allows a deeper understanding of the meaning of the result obtained via UPPAAL.
The control script needs the csv data file generated after the query execution in UPPAAL, positioned in the same folder as the script. To execute it, use the command `python deadlockCheck.py dataFileName.csv`, in this way it can check the progress of the data instant by instant.
In particular, it checks the value of a variable which, as the model was defined, must always vary between 0 and 1 at each time interval. If this, for whatever reason, does not happen and there is any value of either for more than 1 second, then there is the presence of a deadlock.

5.2 A station never holds more than one piece

Since a station is assigned to a belt segment, this query is equivalent to checking that each processing station's belt segment contains at most one piece at any point in time. The number of pieces on a belt segment can be easily calculated using the helper function *nPieces()* implemented in the ProcessingStation template.

5.3 Two pieces never occupy the same belt slot

Two pieces occupying the same slot is reflected in the belt matrix by a cell containing a *true* value being overwritten by another *true* value. We can verify that this never happens by comparing the number of pieces on the belt and the initial number of pieces, if they differ the property is not satisfied. Since the matrix starts empty, the number of pieces can be computed only after the initialization, when the Synchronizer is not on its *Start* state.

5.4 No queue ever exceeds the maximum allowed length

This query is parametric depending on the number of queue sensors instantiated. A queue limit is not exceeded if when the queue is full, the slot immediately before the first queue slot is empty. The first property is verified with a function that takes in input a queue sensor position and checks every slot from that slot to the end of the belt. The second property accounts only for the slot immediately before the start of the queue, as pieces before that point that are still moving are not considered to be part of the queue.

5.5 Policy 2 check

This query verifies that when Policy 2 is selected no piece goes through the inner branch by checking the *counter* attributes of stations 3 and 4.

Results

The comparison obtained between the various configurations is quite obvious. The more simple cases are taken into consideration, the more likely it is to obtain good results.
These results depend very much on the configuration chosen and, consequentially, how the parts are distributed on the chain. Another factor is the processing speed, a station that takes longer to process a piece is also more likely to enter more pieces than necessary in the stochastic model.

Query	**Deterministic Model**			**Stochastic Model**		
	Conf 1	Conf 2	Conf 3	Conf 1	Conf 2	Conf 3
No Deadlock	Sat	Sat	Sat	Sat	Sat	Sat
One piece per station	Sat	Sat	Sat	0.45 ± 0.05	0.73 ± 0.05	0.52 ± 0.05
Max queue length	Sat	Sat	Not Sat	$\geqslant 0.95$	0.70 ± 0.05	$\leqslant 0.05$
No piece overlap	Sat	Sat	Sat	$\geqslant 0.95$	$\geqslant 0.95$	$\geqslant 0.95$
Policy 2 check	Sat	Sat	Sat	$\geqslant 0.95$	$\geqslant 0.95$	$\geqslant 0.95$

Table 1: Queries results

6 Conclusion

In conclusion, the design of the system and the corresponding tasks has been tuned to properly simulate a real world production chain, in order to be ideally implementable on a real application. This design choice has been supported by the great scalability of the code, in fact, any possible configuration is easily implementable by tuning the proper parameters in the "Global Declaration" settings.
For example, it is possible to tune the number of stations and belts and their location in the production chain, the processing time and the respective probability distribution, the number of pieces present in the belt, the position of the sensors and also the selector managing the crossroads between two belts.
It is also important to highlight that the chosen design is highly reliable, but it strongly depends on the design choices of the production chain. In fact, depending on the choice of the sensors (and therefore on their fault probability) and their locations on the production line, the results may change significantly, leading to either a perfect robustness respect to the queries if the design of the line is carried out properly, or a complete violation of the requests (as for configuration 3).
Therefore, a proper harmony between code and physical design choices returns the most robust solutions.