POLITECNICO
MILANO 1863

Formal Digital Twin of a LEGO® MINDSTROMS™ Production Plant

Formal Methods for Concurrent and Real-Time Systems

A.Y. 2022-2023

Andrea Infantino
Person ID 10671720
Student ID 225757

Riccardo Motta
Person ID 10658639
Student ID 218685

Matteo Negro
Person ID 10642961
Student ID 222025

Contents

1 Model Description **3**

1.1 The Production Plant . . . 3

1.2 General Overview . . . 3

1.3 Initializer (`Initializer`) . . . 4

1.4 Motor (`Motor`) . . . 4

1.5 Conveyor Belt (`ConveyorBelt`) . . . 4

1.6 Processing stations (`Station`) . . . 5

1.7 Processing station's input guard (`InSensor`) . . . 5

1.8 Queue's guard (`OutSensor`) . . . 5

1.9 Flow controller (`FlowController_*`) . . . 6

2 Design Decisions **6**

2.1 Hypothesis . . . 6

2.2 Optimizations . . . 6

2.2.1 The basic idea . . . 6

2.2.2 Unique conveyor belt . . . 6

2.2.3 Automata simplification . . . 7

2.2.4 Unique clock . . . 7

3 Properties **7**

3.1 No two workpieces can be in the same position at the same time . . . 7

3.2 Queues never exceed their maximum length . . . 8

3.3 The plant never incurs in deadlock . . . 8

4 Stochastic Version **8**

4.1 Faulty Sensors . . . 8

4.2 Normally distributed stations' processing times . . . 9

5 Scenarios **9**

5.1 Scenario 1: the normality . . . 9

5.2 Scenario 2: short queues . . . 10

5.3 Scenario 3: one way . . . 10

5.4 Scenario 4: the stochastic case . . . 11

5.5 Scenario 5: the faulty stochastic . . . 12

6 Conclusions **12**

1 Model Description

This section provides a description of the production plant with some implementation details, together with all the components that make it up. The motivations behind these modelling choices can be found in Section 2: Design Decisions (page 6).

1.1 The Production Plant

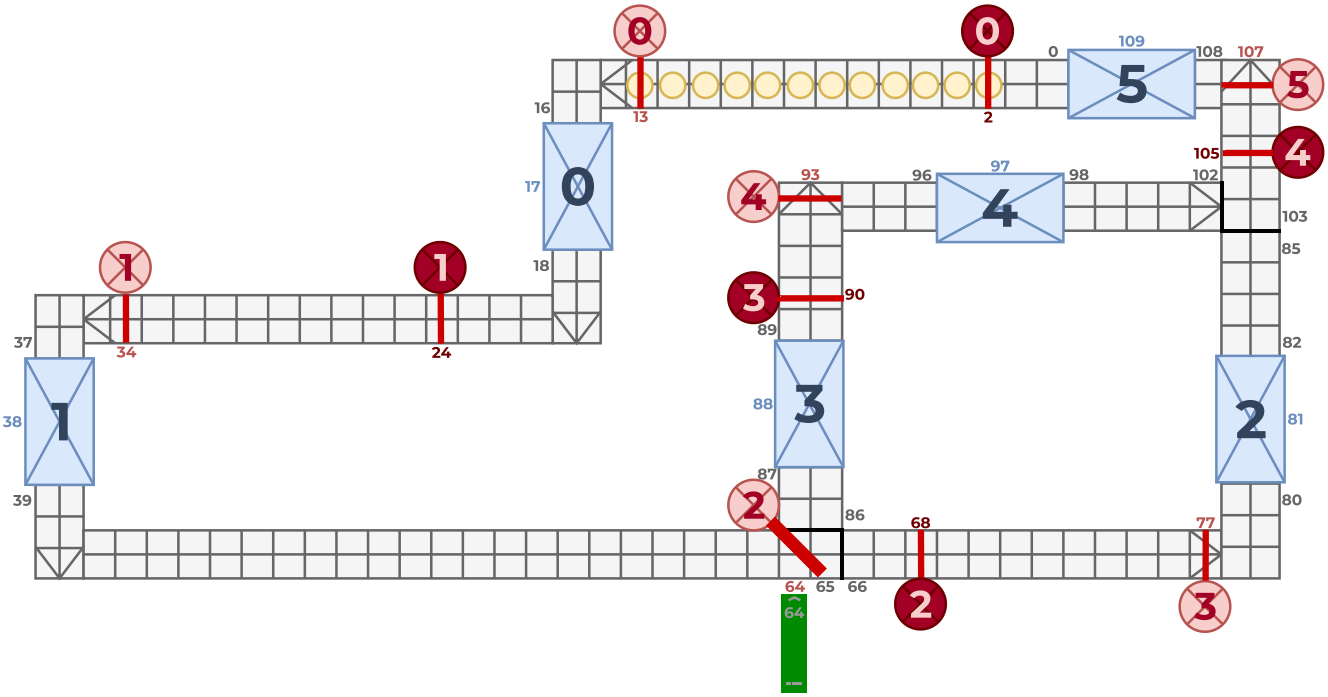


Figure 1: the production plant we modelled.

This is the plant we modelled in our project. The yellow circles are the workpieces, the blue rectangles are the processing stations, the light and dark red circles are the sensors and the green bar is the flow controller.

Notes on the scheme In order to make the entire description of our project as much clear as possible, we enhanced the original scheme with the IDs we assigned to stations and laser sensors, and with the positions of the slots that compose the conveyor belt. Notice that we decided to consider the stations as particular components placed above a normal slot, therefore we have two numbers defining each station: one refers to the position of the slot, while the other the ID of the station.

1.2 General Overview

The model of our system is made of 6 different components which interact between them to coordinate the entire production plant. Some of them are also instantiated many times in order to have a simpler modelling of the entire system.

Initializer It's a fictitious component which represents the entry point of the whole system. It allows us to instantiate all the workpieces in the correct positions of the conveyor belt (positions which start from position 13 and ends in position 2 in our model) and works as the general clock of the system.

Motor This is the real motor of the system. According to the speed at which the conveyor belt should be moving, uses the clock to make all the workpieces move and synchronizes the whole system.

Conveyor belt It's the manager of the conveyor belt, in charge of moving around all the workpieces. In our model it's also the one in charge of preventing workpieces from entering a station, based on the information gathered by the laser sensors. Finally, it handles the positions where the two branches of the plant start and merge.

Processing stations They are the ones in charge of processing the various workpieces, once they get into them. We implemented a single template for them, which is instantiated as many times as needed in order to recreate the plant. As already explained, each station is assumed to be placed above a slot of the conveyor belt.

Laser sensors We have modelled them in two different ways according to their functionalities. The ones guarding the entrance of a station are called `InSensor` (represented as light red circles in Figure 1), while the ones guarding the queue and preventing the station right before it to release a workpiece are called `OutSensor` (represented as dark red circles). Like the stations, they are represented as a single template (one for each type), instantiated multiple times.

Flow controller This is the green piece of Figure 1. It is pre-configured with a specific policy (which is customizable) and decides where to send the workpieces once they get at position 65 of the conveyor belt. The available policies are the following ones:

0. Sends the workpieces on the alternative branch until its queue is full, otherwise lets the workpieces proceed on the main branch;
1. Lets all the workpieces to flow on the main branch;
2. Sends all the workpieces on the alternative branch;
3. Every time a workpiece is in front of him sends it on a different path with respect to the one taken by the previous workpiece.

1.3 Initializer (`Initializer`)

The initializer is the fundamental entry point of our whole system; it is a meta component that performs two different actions.

Initialization Initializes the system by placing the workpieces starting from the slot guarded by the first `InSensor` (index 0), and continuing backward with an upper limit of 12 workpieces.

Global synchronization In order to have a simpler model to verify, we decided to have a unique clock for the whole system. This TA performs as the general clock on which all the components of our plant, directly or indirectly synchronize (usually by means of the signals generated by the motor).

1.4 Motor (`Motor`)

As the name says, this is the meta component that manages all the signals used by the sensors, the conveyor belt and the stations, to synchronize the movement of the workpieces. The two signals which it generates are presented here.

Conveyor belt's tick (`tick_belt`, also used by the stations) It's the signal which makes the conveyor belt move all the workpieces it can one step forward (taking into account also the branch in the conveyor belt).

Laser sensors and stations' tick (`tick_sensor`) In order to avoid any potential race condition, we decided to send a signal right before `tick_belt`. This allows us to update the state of the laser sensors (i.e., check whether there is a workpiece right below them or not) and to perform some state-changing actions on all the stations (if needed).

1.5 Conveyor Belt (`ConveyorBelt`)

It's the entity which handles the plant's conveyor belt. It's a small automaton since all the underlying complexity of the conveyor belt's management is hidden behind the `update()` function. This is the most complex component of our plant since manages a lot of different things.

Workpieces position Any workpiece on a non-critical position (which is any in the following situations) is moved ahead of one slot every time the `update()` function is called.

Stations and sensors management Once a workpiece passes the `InSensor` of a station, it blocks any other workpiece wanting to proceed to the station itself until the sensor tells that the station is free and a new workpiece can flow again towards it.

Queue management Every time a workpiece is blocked in a specific location and some other one wants to proceed, the conveyor belt blocks it until the first one is free to move again. Since we are scanning the belt from the last positions backward this situation is easily managed since we can't place a workpiece on top of another one.

Conveyor belt branch and merge In order to be updated about which branch the next workpiece should take, we decided to keep the Conveyor Belt always ready to receive the signal `switch_branch?`. This allows us to directly use the `update()` function to move the workpieces on the correct branch. Also, the merge of the two branches is (easily) managed by the conveyor belt, since, by scanning it backward, once we move a workpiece in the merge position of the belt (position 103), no other one can be put in the same place, thus, blocking it from proceeding, which is exactly what happens in reality.

1.6 Processing stations (`Station`)

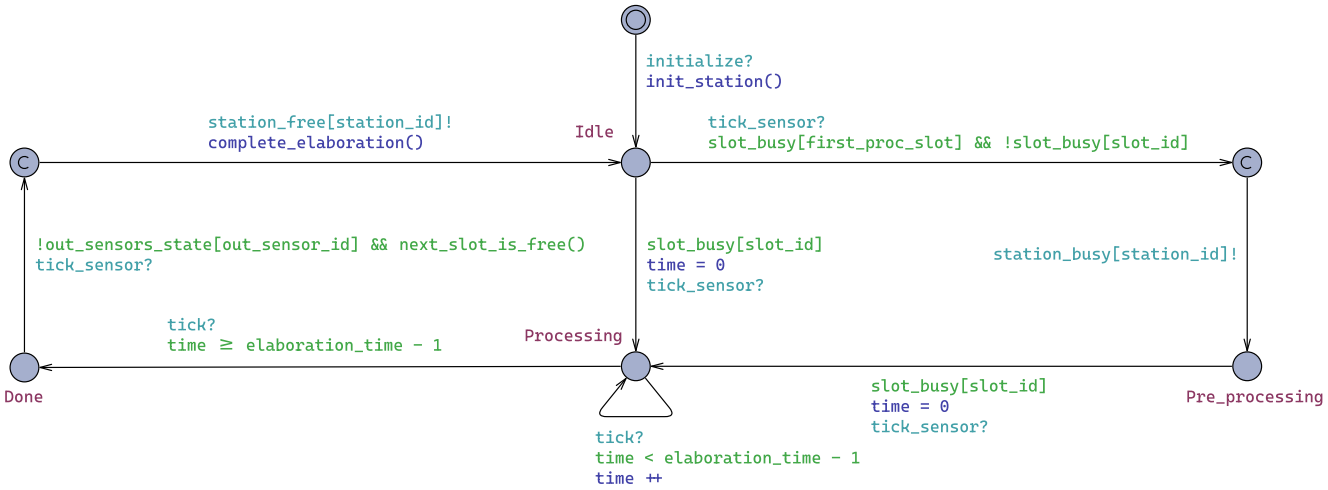


Figure 2: the Timed Automata of every station.

The stations model the various processes the workpieces need to go through during production. The automaton is divided in four different main states, each one representing a status in which the station can be.

Idle This is the main state of the station, in which it doesn't have to perform any action but to wait for a workpiece to be ready for the processing.

Preprocessing When the laser sensors which is guarding the entrance of the station signals that a workpiece is approaching to it, the station moves to this state waiting for the entrance of the workpiece.

Processing Once the workpiece is entered, the station starts processing it. In the model, the automaton isn't performing any real task on the workpiece, but it's simply waiting for its processing time to pass. In order to improve verification performance, we discretized the time, and so we use a counter synchronized with the global clock in order to model the time passing.

Done Once the processing time is elapsed, the station moves to this state, where it waits for the right time to release the workpiece on the conveyor belt. It synchronizes with the conveyor belt itself and with the sensors guarding the queue of the following station, in order to know if the workpiece can be release or has to be held inside the station itself. Once the workpiece is released, the station returns to its *idle* state, waiting for the next processing cycle.

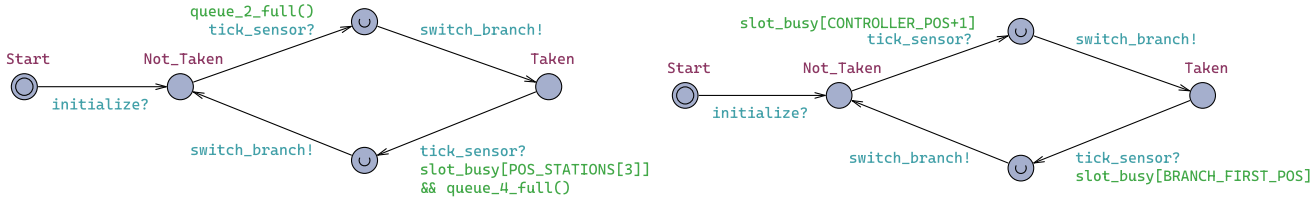
1.7 Processing station's input guard (`InSensor`)

These are the laser sensors that guard the entrance of a specific station. If a workpiece oversteps it, the laser sensor is equipped with a physical barrier which blocks any other following workpiece from proceeding towards the guarded station, until it is free again.

1.8 Queue's guard (`OutSensor`)

Right before a station there is always an `InSensor` which guards its entrance, but it can be preceded by another sensor, which guards the queue. Once the station's guard blocks a workpiece, others may arrive forming a queue of workpieces. This queue may have a maximum length. If the queue reaches its maximum, the sensor prevents the preceding station to release a workpiece until the queue is no longer full.

1.9 Flow controller (`FlowController_*`)



(a) Controller v0.

(b) Controller v3.

Figure 3: the two non-trivial `FlowController` we ideated.

As explained in the introduction, these are the various policy schedules we implemented. Each one of them represents a different way to schedule the workpieces on the processing stations. Starting from the easiest ones, the ones in which all the workpieces follow the same path (either the direct one or the alternative one), to more complex ones, like the one in which we keep track of the queues in front of the first stations of the branch.

Since the effective movement of the various workpieces is managed by the conveyor belt, these automata send a signal (`switch_branch`) to it each time the path followed by the workpieces has to be changed.

2 Design Decisions

2.1 Hypothesis

Slots' numbering We numbered the slots as shown in Figure 1. This helped us in the management of the conveyor belt's update function and implicitly allowed us to solve the problem of merging the two alternative paths at position 103. In the (possible) corner case in which both position 85 and position 102 are carrying a workpiece at the same time (i.e., they are simultaneously trying to place a workpiece on slot 103), we decided to apply the following policy: the first workpiece to flow is always the one coming from station 4. This is, of course, a simplification. A more complicated approach could have been taken, like randomly picking one of the two workpieces, but we decided to keep it as simple as we could.

Maximum number of circulating workpieces We decided to stick with the representation of Figure 1. So we have at most twelve workpieces circulating in the production plant, and they all are initially placed from position 13, proceeding backward. Please, notice that if the number of workpieces exceeds the position of the first `OutSensor` (which can be changed by project hypothesis), one of the properties isn't verified. However, this is normal since during the initialization of the plant we are already exceeding the maximum queue's length of the first one.

Time discretization This will be more clear in the next part, where we describe the main optimization steps we did, but, in brief, we discretized the time in order to have a better synchronization mechanism of the whole plant.

2.2 Optimizations

Here we present all the main optimization steps we performed in order to have the final model.

2.2.1 The basic idea

Idea At first, we decided to model every single entity of the plant. We had the template for generating all the single slots of the conveyor belt. This has been done in order to have a precise (and discrete) representation of the plant pointing out all the atomic components that we have.

Situation Proceeding in this way, we experienced huge verification times, due to the high number of entities and states that the software had to process each time.

2.2.2 Unique conveyor belt

Idea In order to simplify the modelling of the plant and our code, we decided to group the slots of the conveyor belt all together into a single logically entity (the `ConveyorBelt`).

Situation This simplified a lot the management of all the various cases (and, indeed, is still present in our project), since we had a unique entity to manage the complexity derived from the movement of the various workpieces. This also works in synchronicity with the processing stations and the sensors, in order to block the workpieces when needed, and with the flow controller, which communicates to the `ConveyorBelt` where to send the workpieces on the branch. By doing this we also improved significantly the verification time needed to check the properties, but we still had a huge memory consumption.

2.2.3 Automata simplification

Idea One of the things that was making the verification process heavy and slow, was the wide number of states traversed at each iteration. So, we decided to try to reduce the number of total states and transitions composing the entire system at the bare minimum.

Situation We removed some superfluous states and transitions from our model. We also split the flow controller into four different ones (one for each policy), in order to instantiate only the one that we were going to use during the simulation and verification of the plant. This allowed us to have slight improvements both in verification times and memory consumption.

2.2.4 Unique clock

Idea We realized that we actually had 7 different clocks inside our project (one for the general clock of the system and one for each station). Since this fact was one of the biggest bottlenecks during the verification phase, we decided to reduce them to just a general one, and to keep all the various templates synchronized with it.

Situation We added a new template, the `Motor`, which sends the right signals at the right moment to all the components that need it, while respecting the speed of the conveyor belt's movement; then we synchronized the stations' processing times with the general clock of the system (which is not related to the speed of the conveyor belt). By doing this, we reduced the number of possible transitions the system can potentially take at any instance. This allowed us to have a skyrocketing verification time, even with 12 different workpieces circulating in the production plant at the same time.

3 Properties

3.1 No two workpieces can be in the same position at the same time

Since we modelled the plant as a unique conveyor belt with the stations placed on some specific slots, this property embraces two different requirements of the production plant:

- *It never happens that a station holds more than 1 workpiece*;
- *It never happens that two workpieces occupy the same conveyor belt slot.*

Note that the slots are modelled as an array of 110 Boolean values, in which `true` corresponds to the presence of a workpiece in that position.

In order to verify this property, we check that, after the plant has been initialized, the number of the workpieces remains constant and equal to its configured number, which can be represented in TCTL as follows:

$$\forall\Box \left(\underbrace{\texttt{initializer.run\_state}}_{\varphi_1} \implies \left(\underbrace{\sum_{i=0}^{109} \texttt{slot\_busy[i]}}_{\varphi_2} = \texttt{DISKS} \right) \right)$$

- φ_1: the system has been initialized;
- φ_2: the number of circulating workpieces in the production plant.

3.2 Queues never exceed their maximum length

We verified this by exploiting a custom function which, given the positions of a related pair (`InSensor`, `OutSensor`), we check that there is at least one free slot between the position of the `InSensor` and the position preceding the one of the `OutSensor`. If the property is false, it means that there is at most one extra workpiece in the queue. We verified this as follows:

$$\forall\Box\left(\forall i \in [0;4]\left(\underbrace{\texttt{outSensor(i).Locked}}_{\varphi_1} \implies \underbrace{\texttt{check\_queue(i, POS\_OUT\_SENSORS[i])}}_{\varphi_2}\right)\right)$$

- φ_1: the i-th `InSensor` is locked (the guarded station is not in `Idle`);
- φ_2: checks that the queue doesn't exceed its maximum length.

3.3 The plant never incurs in deadlock

In order to verify this, we checked both the automata part and the logic one. This means that we made sure to not incur in a deadlock in the transitions, and that all the workpieces are continuing to circulate in the plant.

The property for the transitions is verified with $\forall\Box\,(\neg\texttt{deadlock})$, while the second one with:

$$\forall\Box\left(\neg\left(\underbrace{(\forall i \in [0;5]\,(\texttt{station(i).Processing} \vee \texttt{station(i).Done}))}_{\varphi_1} \wedge \underbrace{(\forall i \in [0;4]\,\texttt{outSensor(i).Locked})}_{\varphi_2} \wedge \underbrace{(\forall i \in [0;5]\,\texttt{inSensor(i).Locked})}_{\varphi_3}\right)\right)$$

- φ_1: all the stations are either blocked on `Processing` or `Done`;
- φ_2: all the `OutSensor` are preventing workpieces from exiting the stations;
- φ_3: all the `InSensor` are blocking workpieces from reaching the stations.

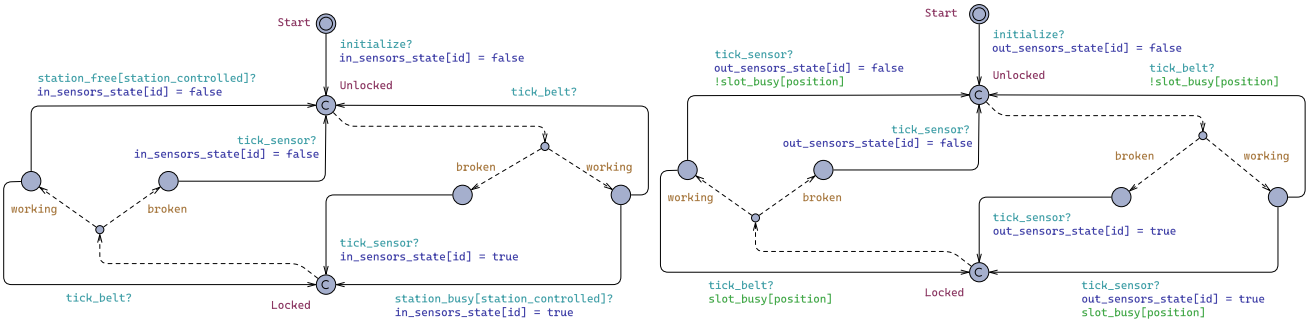
Note This doesn't include all the possible configurations in which a deadlock can occur, since in UPPAAL we didn't find, after a long and deep research, a way to express a formula like the following correct one:

$$\exists\Diamond\,((\exists i \in [0;5]\,\Box\texttt{station(i).Done}) \vee (\exists i \in [0;5]\,\Box\texttt{inSensor(i).Locked}) \vee (\exists i \in [0;4]\,\Box\texttt{outSensor(i).Locked}))$$

4 Stochastic Version

In real world applications, nothing is unbreakable. Every kind of concrete component can be subject to fault, errors and/or failures; that's why, in order to provide a model as much suitable to a real application context as possible, we implemented a separate version of our original project, improved with the two following stochastic features[1].

4.1 Faulty Sensors



(a) Stochastic `InSensor`. **(b)** Stochastic `OutSensor`.

Figure 4: the stochastic version of our `InSensor` and `OutSensor`.

In real world, every kind of sensors might provide false positives (in our case, detection of a not actually present workpiece) and/or false negatives (in our case, non-detection of an actually present workpiece). We took into account this feature by modifying the original TA of both `InSensor` and `OutSensor` as shown in Figure 4. Starting from the original non-stochastic version of the two type of sensors, we added two couples of alternative states, `C1-C2`

[1] Reference: https://www.it.uu.se/research/group/darts/papers/texts/uppaal-smc-tutorial.pdf.

and C3-C4, respectively handling potential false positives and false negatives; in this way, each time the sensor is unlocked, one state among C1 (meaning that the sensor is working properly) and C2 (meaning that a detection error has occurred) is stochastically chosen, based on a probability weight passed as parameter to the template. Similarly, every time the sensor is locked, state C3 is visited if the sensor has performed a misreading, and state C4 otherwise.

Deadlock avoidance As you can see in Figure 4, we added two apparently useless edges: the first from state C1 to state Unlocked, and the second from state C4 to state Locked. These edges allow the sensors to check for the presence of a workpiece on the guarded slot at every tick, emulating the real behaviour of a sensor, which can misread every time. This is also helpful in order to avoid unwanted potential deadlocks, since it can always get back to a fully functioning state.

4.2 Normally distributed stations' processing times

The processing time of any real industrial processing station can be effectively measured in order to coordinate every other component that has to interact with it. However, even if they are usually very close one from each other, the time taken in practice is never exactly the same for every single processing. We abstracted this classical behavior by creating a normal distribution function with mean value and standard deviation passed as parameters of the Station template; then, each time a workpiece enters a station, the processing time needed is stochastically picked based on the computed normal distribution.

UPPAAL limit While implementing the normal distribution function, we experienced that UPPAAL is unable to compare clocks (used to keep track of the passing of the time) with double numbers. In order to deal with that, after have stochastically picked the needed processing time, we decided to round it to the closest integer value.

5 Scenarios

In this section we are going to propose some possible scenarios, aiming at proving the resiliency of our model both in standard and borderline situations. Furthermore, we included also some pathological cases, in which the plant fails to guarantee some properties (as it should be).

5.1 Scenario 1: the normality

This can be considered the "standard" scenario: the structure of the plant is the one depicted in the introduction. The stations' processing times have been chosen without any specific criterion, and we instructed the flow controller to equally send the workpieces on the two different branches of the conveyor belt, in the way defined above as Policy 3.

Parameters These are the parameters used to obtain Figure 5.

SPEED	DISKS	Policy	POS_OUT_SENSORS	STATIONS_ELABORATION_TIME
1	12	3	[2, 24, 68, 90, 105]	[6, 7, 8, 9, 8, 7]

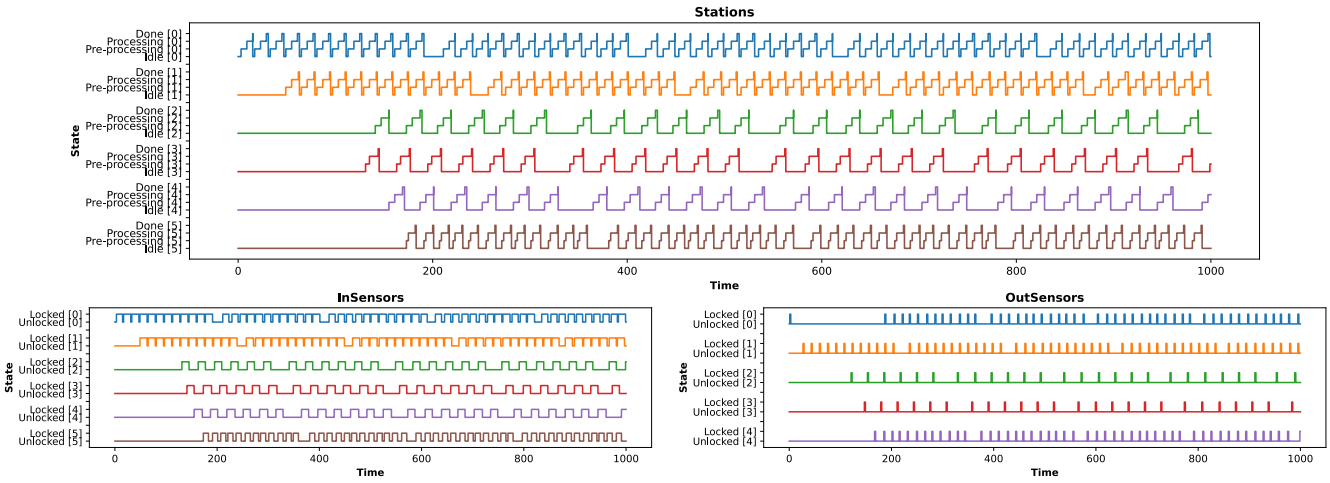


Figure 5: simulation over 1000 time instants of the first scenario.

Results All the properties are *verified*. As it is supposed to, the plant is correctly working: every workpiece flows through the plant without any problem. The sensors are guaranteeing the required properties and the stations are working flawlessly.

5.2 Scenario 2: short queues

The idea behind this scenario is to test whether the plant is working in a realistic situation characterized by short queues of workpieces waiting to be processed by the stations (i.e., each `OutSensor` is placed very close to its related `InSensor`).

Parameters These are the parameters used to obtain Figure 6.

SPEED	DISKS	Policy	POS_OUT_SENSORS	STATIONS_ELABORATION_TIME
1	12	3	[12, 33, 76, 92, 106]	[2, 15, 5, 3, 2, 5]

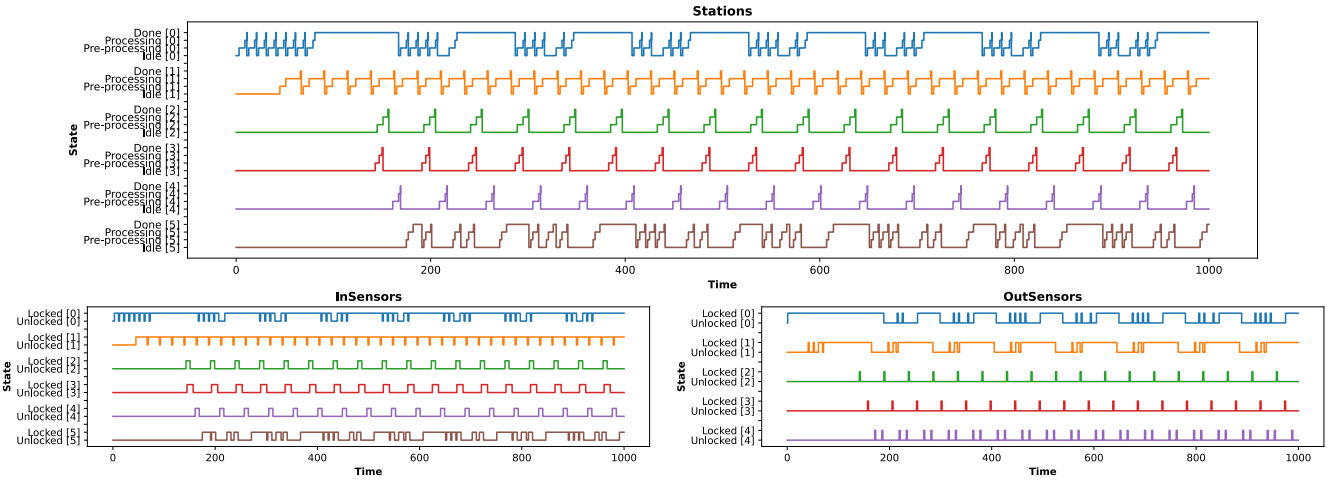


Figure 6: simulation over 1000 time instants of the second scenario.

Results All the properties are *verified*. The graphics is showing the system is working properly without any particular delay of the stations.

Particular case If we slightly change the processing times of the stations, we obtain a situation where stations 2 and 4 release a workpiece, which cause the saturation the queue preceding station 5, and exceeds its limit. For instance, this can be obtained by setting very short processing times on the first two stations, the same overall time on the two branches, and very long one on the last. In this case, the third property is no longer verified. Anyway, this behavior is completely expected, since the plant isn't smart enough to prevent this situation.

5.3 Scenario 3: one way

Here we wanted to test a different scheduling policy: we kept the same parameters of the first scenario and changed only the scheduling policy followed by the `FlowController`. In particular, we chose to route all the workpieces through the branch with the two processing stations (Policy 2).

Parameters These are the parameters used to obtain Figure 7.

SPEED	DISKS	Policy	POS_OUT_SENSORS	STATIONS_ELABORATION_TIME
1	12	2	[2, 24, 68, 90, 105]	[6, 7, 8, 9, 8, 7]

Results All the properties are *verified*. If we take a closer look to the graphics, we notice that the scheduling policy adopted here is slightly more inefficient w.r.t. the standard one (i.e., the one used in Scenario 1): in particular, here station 0 processes 56 workpieces in 1000 time instants, while, considering the same time frame, the standard scenario is capable of processing 58 workpieces. The outcome of this comparison is easy understandable, since balancing the workload is usually the best solution.

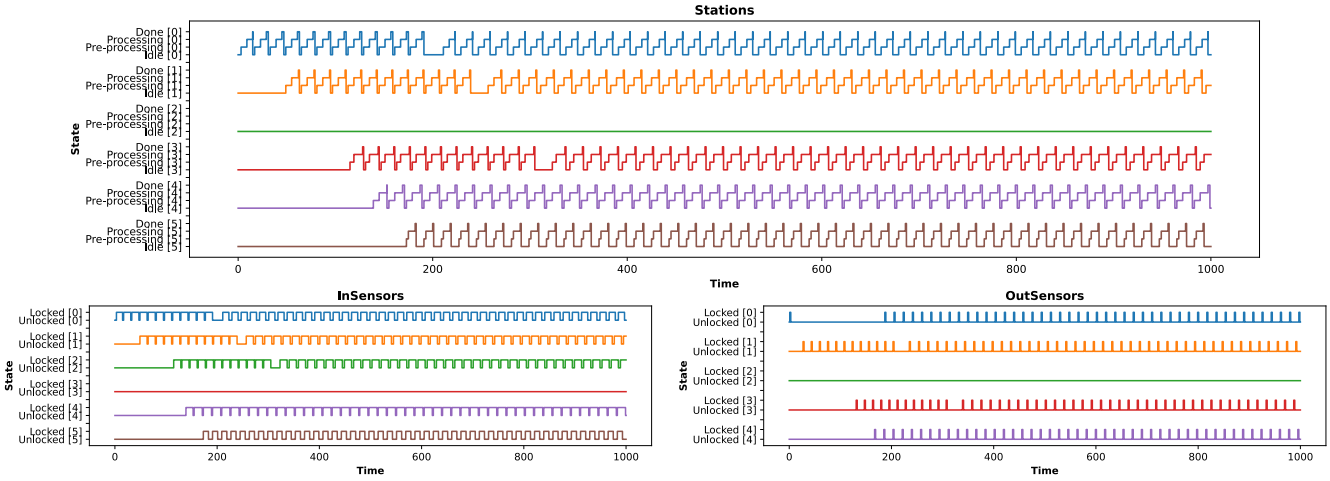


Figure 7: simulation over 1000 time instants of the third scenario.

5.4 Scenario 4: the stochastic case

Here we got the results by running the stochastic version of our project: each station's processing time varies as a Gaussian distribution around an average value, while the sensors can be faulty.

Parameters These are the parameters used to obtain Figure 8.

SPEED	DISKS	Policy	POS_OUT_SENSORS	STATIONS_ELABORATION_TIME
1	12	3	[2, 24, 68, 90, 105]	[6, 7, 8, 9, 8, 7]

STD_DEV_STATIONS	IN_SENSORS_ERR	OUT_SENSORS_ERR
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0]	[1, 1, 1, 1, 1, 1]	[1, 1, 1, 1, 1, 1]

IN_SENSORS_RIGHT	OUT_SENSORS_RIGHT
[9999, 9999, 9999, 9999, 9999, 9999]	[9999, 9999, 9999, 9999, 9999, 9999]

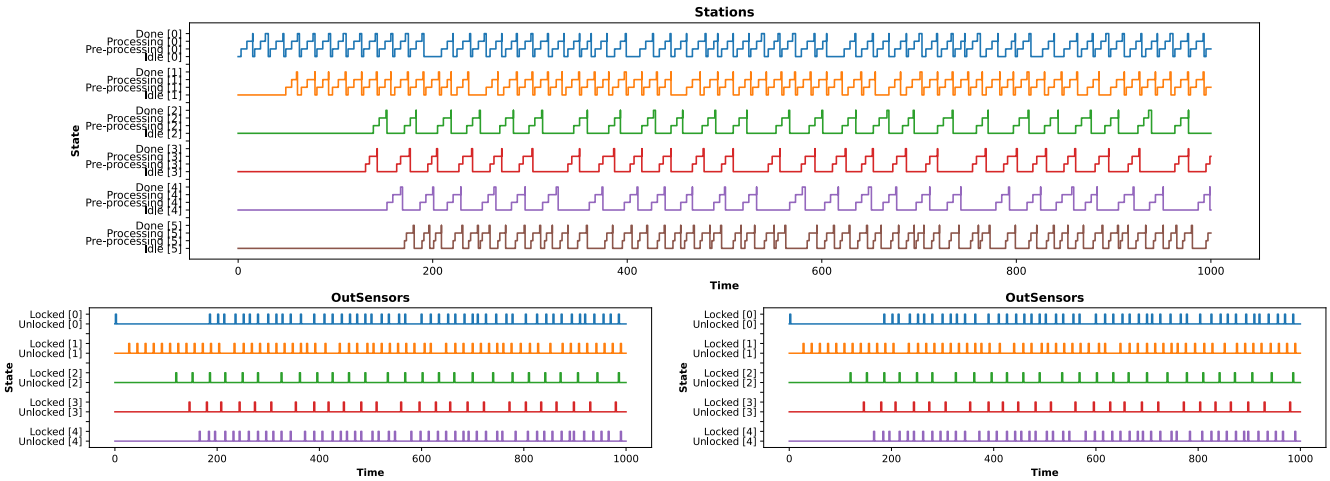


Figure 8: simulation over 1000 time instants of the fourth scenario in a stochastic environment.

Results All the properties are *verified* with a 95% confidence and with the following probabilities:

- Property 1: No two workpieces can be in the same position at the same time: $> 95\%$;
- Property 2: Queues never exceed their maximum length: $84\% \pm 5\%$;
- Property 3: The plant never incurs in deadlock: $> 95\%$.

We can see that the plant is still working in a non-ideal scenario, correctly processing all the workpieces, without breaking physics constraints. Note that it may happen that some queues exceeds their maximum length, because of faulty sensors.

5.5 Scenario 5: the faulty stochastic

The goal here is to show that in the (unrealistic) case of a stochastic production plant with highly faulty sensors, the system keeps working properly, even if the queues maximum lengths can't be guaranteed.

Parameters These are the parameters used to obtain Figure 9.

SPEED	DISKS	Policy	POS_OUT_SENSORS	STATIONS_ELABORATION_TIME
1	12	3	[2, 24, 68, 90, 105]	[6, 7, 8, 9, 8, 7]

STD_DEV_STATIONS	IN_SENSORS_ERR	OUT_SENSORS_ERR
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0]	[1, 1, 1, 1, 1, 1]	[1, 1, 1, 1, 1, 1]

IN_SENSORS_RIGHT	OUT_SENSORS_RIGHT
[99, 99, 99, 99, 99, 99]	[99, 99, 99, 99, 99, 99]

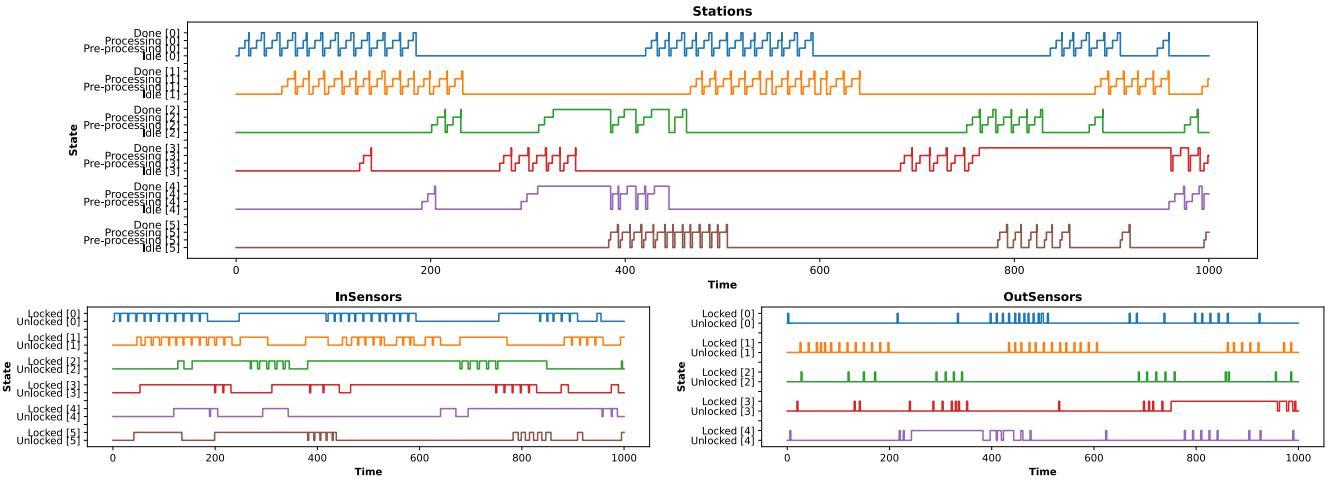


Figure 9: simulation over 1000 time instants of the fifth scenario in a stochastic environment.

Results Some properties are *not verified*, but all with a 95% confidence. We achieved the following probabilities:

- Property 1: No two workpieces can be in the same position at the same time: $> 95\%$;
- Property 2: Queues never exceed their maximum length: $< 5\%$;
- Property 3: The plant never incurs in deadlock: $> 95\%$.

It is clear that, also in this case, the plant is still functioning, even if it's almost certain that at least one queue exceeds its maximum length. This is due to the very high probability of a sensor to be faulty, for which in 10000 time instants, this property can't be fully verified. However, the plant doesn't incur in any deadlock and physical constraints are respected.

6 Conclusions

In conclusion, we argue that the results obtained in the various scenarios just presented can be considered proofs of the efficiency and the resiliency achieved by the proposed model. We can state that our project provides a complete abstraction of the LEGO® MINDSTROMS™ Production Plant, by considering both common and rare situations, without neglecting any corner case. From the perspective of performance, we think that, after several optimizations and refinements, we have achieved remarkable outcomes, both in terms of verification speed and memory consumption. By virtue of the considerations exhibited so far, we deeply believe that our model is sticking close to the specifications required by the LEGO® MINDSTROMS™ Production Plant's model.