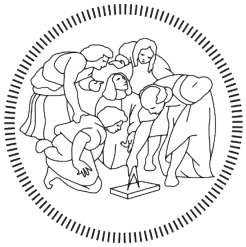


POLITECNICO
MILANO 1863

Formal Methods For Concurrent and Real-Time Systems

Homework Project 2022-2023

Formal Digital Twin Of A Lego Mindstorms Production Plant

Valeria Detomas - 10615309
Mauro Famà - 10631287
Giulia Gerometta - 10628256

July 19, 2023

Table of Contents

1 Introduction

This document presents a UPPAAL formal model of a Lego Mindstorms production plant.
Our model reinforces critical properties and analyzes the behavior of the plant under different configurations, both deterministic and stochastic, used for modeling and verifying real-time systems, involving two phases: exhaustive model checking and statistical model checking.
The production plant operates with a one-directional conveyor belt that carries workpieces flow through different processing stations. Each station can process only one piece at a time, and the processing duration may vary between stations. Our model handles the movement of the pieces with a controller that can direct them to a specific station or divert them to an alternative belt branch. Laser-based sensors detect the presence of pieces and prevent station overcrowding by blocking new workpieces from entering when the abovementioned station is busy.
In the stochastic version of the production plant, certain aspects of the system involve randomness or uncertainty. The processing time of each station follows a probabilistic distribution, more precisely, a normal distribution. Additionally, there is a probability of sensor malfunctions, where sensors may detect or fail to detect pieces incorrectly.

2 UPPAAL Model

2.1 Data Structures

We decided to manage the production plant relying on five pivotal arrays: `belt`, `out_sensors`, `in_sensors`, `preprocessing`, and `stations`.
Each of the above data structures represents a fundamental component of the production plant and tracks the state and positions of various elements within the system. The most critical data structure is the `belt` array, which represents the *Conveyor Belt*. It consists of a boolean array where each element corresponds to a specific `belt` slot. The binary value assigned to each array element denotes the presence or absence of a workpiece within the respective `belt` slot.
The remaining arrays, namely `stations`, `in_sensors`, `out_sensors`, and `preprocessing`, follow a similar pattern and have the same length as the *Conveyor Belt*. These arrays are paramount for accurately recording the positions and statuses of associated components. For instance, if a *Processing Station* occupies the *i-th* slot, the element at index i in the `stations` array will store the unique `id` of the *Processing Station*. The same principle applies to the other data structures, facilitating the identification and tracking of *In Sensor*s, *Out Sensor*s, and *Processing Station* preprocessing components. By utilizing these arrays, we establish a robust framework to effectively manage, monitor, and synchronize the state and locations of the elements within the production plant. Moreover, we use auxiliary boolean arrays, namely `status_stations`, `status_in_sensors`, and `status_preprocessing`, to determine the status of the corresponding elements they represent. These arrays have a length equal to the total number of components they represent, and each position corresponds to the `id` of the corresponding element. A value of 1 indicates that the production plant component is busy, while 0 denotes its availability. By monitoring these arrays, the system can effectively and comprehensively track the occupancy status of the elements.

2.2 Templates

Our UPPAAL model is based on four different templates.

2.2.1 Conveyor Belt

In our model, the *Conveyor Belt* template is responsible for the movement of the pieces and manages the switching policy of the controller.
It is a sort of engine whose main task is synchronizing the steps of the workpieces onto the belt as well as avoiding the overlap in the same belt position. In this template, we set a clock which is in charge of

regulating the speed of the belt; hence, every `x` seconds, it triggers the `step` channel. The `step` channel links the *Conveyor Belt* with the *In Sensor*, the laser-based sensor at the entrance of the stations, and with the *Processing Station*. For each step in the belt, we check whether or not the workpieces are in an ordinary or significant position (i.e., a piece is on a sensor, station, or gate).

2.2.2 Processing Station

As previously mentioned, the *Processing Station* is in charge of workpiece processing for a fixed amount of time, which may vary in each processing station.
We decided to include a PREPROCESSING stage, which comprehends the belt slots from the *In Sensor* to the *Processing Station* itself. We considered this stage since only one workpiece at a time can flow in this part of the belt. Indeed, if a workpiece is in the *Processing Station*, the previous one has to wait on the *In Sensor* until the *Processing Station* is free again. The *Processing Station*'s TA evolves as follows:

1. The *Processing Station* waits for the `enter_preprocessing` notification, fired by the *In Sensor*, which indicates that a workpiece is ready to enter the PREPROCESSING phase.
2. While in PREPROCESSING, the workpiece continues moving until it encounters the *Processing Station* in which it urgently enters and pauses there for the fixed processing time.
3. Once the *Processing Station* finishes the workpiece processing, meaning that the processing time elapses, there can be two different scenarios depending on the presence of a sensor guarding the end of the queue (i.e., *Out Sensor*):
   (a) If the *Processing Station* does not have an *Out Sensor* and the following position in the belt is free, it can directly release the piece.
   (b) Otherwise, the *Processing Station* has to check if the *Out Sensor* is free or occupied:
      i. If it is free and the next belt position is empty, the *Processing Station* will release the workpiece;
      ii. Else, the *Processing Station* enters the WAIT state, waiting for the *Out Sensor* free notification. At this point, there can be another *Out Sensor* to check. Because of the presence of the branch, a *Processing Station* may have to check the filling of two queues. In this instance, we assumed that a workpiece must wait in the *Processing Station* until both *Out Sensor*s are freed.

2.2.3 In Sensor

The *In Sensor* template contains the logic of the laser-based sensor placed at the entrance of each *Processing Station*. Since there is always a sensor monitoring the access to a *Processing Station*, we set the sensor `id` matching the *Processing Station* `id`. In the first transition of the automaton, we store the *In Sensor*'s position in the variable `in_sensor_pos`.
The *Conveyor Belt* triggers the *In Sensor* through the `step` channel, synchronizing the *In Sensor* with the movement of the belt.
The *In Sensor* checks if in its position there is a piece present on the belt or not. When the sensor detects a workpiece on the *Conveyor Belt* in the `in_sensor_pos`, the *In Sensor* status is updated on the global array `status_in_sensors`.
At this point, we wait for the *Conveyor Belt* to move again to notify the corresponding *Processing Station*, via the `enter_preprocessing` channel, that a piece is ready to enter the PREPROCESSING phase. The *In Sensor* stays in the BUSY state until that *Processing Station* triggers the `free_in_sensor` channel, announcing that the workpiece successfully left the *Processing Station*.

2.2.4 Out Sensor

The Out Sensor template models the laser-based sensor guarding the end of the queue. Identical as in the *In Sensor*, we store the *Out Sensor* position in the corresponding variable. The *Out Sensor* listens to the `check_outsensor` channel, triggered by the previous *Processing Station* when it needs to release a piece. The *Out Sensor*, in turn, triggers two different channels depending on the presence or not of a workpiece on top of it. The *Processing Station* receives busy if there is a piece, free otherwise.

2.3 Channels

In out system we leveraged several channels:

- `initialize`: It is a broadcast channel used for system initialization. It triggers the initialization process and sets the parameters for the system components.
- `step`: It is a broadcast channel used to synchronize the *Conveyor Belt* steps. It enables the components to advance together in a synchronized manner.
- `enter_preprocessing`: This channel is indexed by the number of *Processing Station*s in the system. It signals the entry of a workpiece into the PREPROCESSING stage.
- `free_in_sensor`: This channel is indexed by the number of *In Sensor*s in the system. It indicates the availability of an *In Sensor* to process a workpiece.
- `check_out_sensor`: This channel is indexed by the number of *Out Sensor*s in the system. The *Processing Station* triggers this channel to check the status of the *Out Sensor*.
- `out_sensor_busy`: This channel is indexed by the number of *Out Sensor*s in the system. It notifies the *Processing Station* that the *Out Sensor* is occupied and the *Processing Station* cannot release the workpiece.
- `out_sensor_free`: This channel is indexed by the number of *Out Sensor*s in the system. It notifies the *Processing Station* that the *Out Sensor* is free and the *Processing Station* can release the workpiece.

3 Stochastic Version

We implemented both stochastic features proposed:

1. The processing time may vary within a single *Processing Station* following a normal (Gaussian) distribution. We decided to add mean and variance as variables for each *Processing Station* which allowed us to compute the random processing time of the stations according to the normal distribution. We calculate the processing time every time the *Processing Station* changes from PREPROCESSING state to PROCESSING state using the function:

   $double\ random\_normal(double\ mean, double\ sd)$

   This UPPAAL function computes the pseudo-random number distributed according to normal distribution for a given mean and standard deviation. Since the function returns a *double*, we converted it to *integer* type using the *fint()* function. This approximation is acceptable because we check the delay without removing any relevant information. We use the processing time computed as an *invariant* in the PROCESSING state. The simulation is still well described using *integers*.

2. There is a certain probability of a sensor malfunctioning, meaning that a sensor may not work correctly every time the *Conveyor Belt* moves. This feature is based on the probability weight of the error, which is initialized differently for each sensor.
   In this context, there are four cases to take care of:

   (a) The *In Sensor* detects a piece that is not present. In this case, the system slows down because it blocks the workpieces behind. Indeed, they will wait for a clock cycle to move again.

   (b) The *In Sensor* fails to detect a piece that is on it. What happens is that the workpiece will keep going forward as the *Conveyor Belt* moves. The workpiece will enter the *Processing Station*, but it will not get processed, leaving the *Processing Station* immediately. Another odd case happens when right after the undetected workpiece, there is another piece that now gets detected. The second piece will move forward to the *Processing Station*, generating a not allowed queue in the preprocessing phase before the *Processing Station*. The latter workpiece will actually enter the preprocessing state of the timed automata of the *Processing*

Station right after the undetected piece sets the station status to one; hence the station starts processing the latter one. In the meantime, the detected workpiece (if no other piece behind it gets detected) will move forward without being processed by the *Processing Station*. The undetected workpiece does not get processed because we set the processing time of a station in the preprocessing phase; therefore, if the *In Sensor* does not detect a piece, that time remains zero.

(c) The *Out Sensor* detects a piece that is not there, resulting in a delay if a workpiece is in the previous *Processing Station*. In this case, the probability of the system's deadlock increases.

(d) The *Out Sensor* does not detect a piece when it is there, making the previous *Processing Station* release the workpiece. For this reason, the probability of exceeding the maximum length of the queue increases.

4 Design Assumptions

The `shift()` function is responsible for the actual movement logic of the *Conveyor Belt* in terms of the step-by-step advancement of a single workpiece on it. This function moves the pieces every time the *Conveyor Belt* clock overheads a specific duration. The `shift()` goes through the `belt` array from the last position to the first one, checking whether there is an empty slot after an occupied one (i.e., *belt[i] == 1 AND belt[i + 1] == 0*). Therefore, the function swaps only the *1*s followed by the *0*s in order to reduce the number of operations performed. Moreover, `shift()` controls if a piece is free to move based on its position, sensor status, and station status. For example, if a piece is currently in a *Processing Station*, the station status is *1* until the workpiece does not exit it. Once the station finishes the processing phase, it sets its status to *0* releasing the workpiece.
During the implementation of the system, we made the following assumptions:

1. The number of *In Sensor*s corresponds to the number of *Processing Station*s, while the number of *Out Sensor*s is less or equal to the number of *Processing Station*s.

2. We placed processing stations and the corresponding laser-based sensors on valid cells:

   (a) The *In Sensor* of a station must precede the *Processing Station*.

   (b) Between an *In Sensor* and a *Processing Station*, there is always at least one preprocessing slot.

3. If a workpiece passes the *In Sensor*, the *In Sensor* remains busy for all the preprocessing and processing phases until the *Processing Station* releases the workpiece. Hence, we assumed that the workpieces right before it are blocked.

4. Referring to the belt configuration in Figure 4.1, we assumed that *Station 1* does not release the workpiece until both *Out Sensor 2* and *Out Sensor 3* are free.

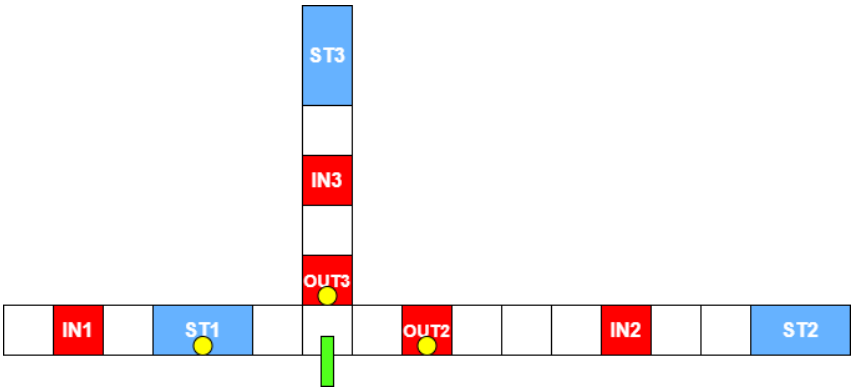


Figure 4.1: Belt configuration with controller

5 Evaluation and Results

5.1 Properties

We verified all the mandatory properties, namely:

1. **It never happens that a station holds more than one piece**.
   In order to verify this property, we added a counter that is increased before the `PROCESSING` state in the *Processing Station* and decreased right after. This counter must be less or equal to *1* to verify the latter property. We expressed this feature in TCTL logic as:

   $\forall\Box(\forall(i : id_s)Processing\_Station(i).processing\_counter <= 1)$

2. **It never happens that two pieces occupy the same belt slot**. Every time the `shift()` function fires, we update the `belt_count` variable as the sum of *1*s in the `belt` array. If the `belt_count` is always equal to the number of workpieces, it never occurs that two pieces are in the same slot, which would result in `belt_count` strictly less than the number of workpieces in the `belt`.

   $\forall\Box(\neg(Conveyor\_Belt.INIT) \implies belt\_count == NUM\_PIECES)$

3. **No queue ever exceeds the maximum allowed length**. We define the queue as the number of pieces occupying the slots between the *In Sensor* and the *Out Sensor*. We set each maximum span of the queue when we initialize the plant. While the current length of the queue is updated run-time every time the *Conveyor Belt* moves.

   $\forall\Box(\forall(i : id_s)len\_queue[i] <= max\_len\_queue[i])$

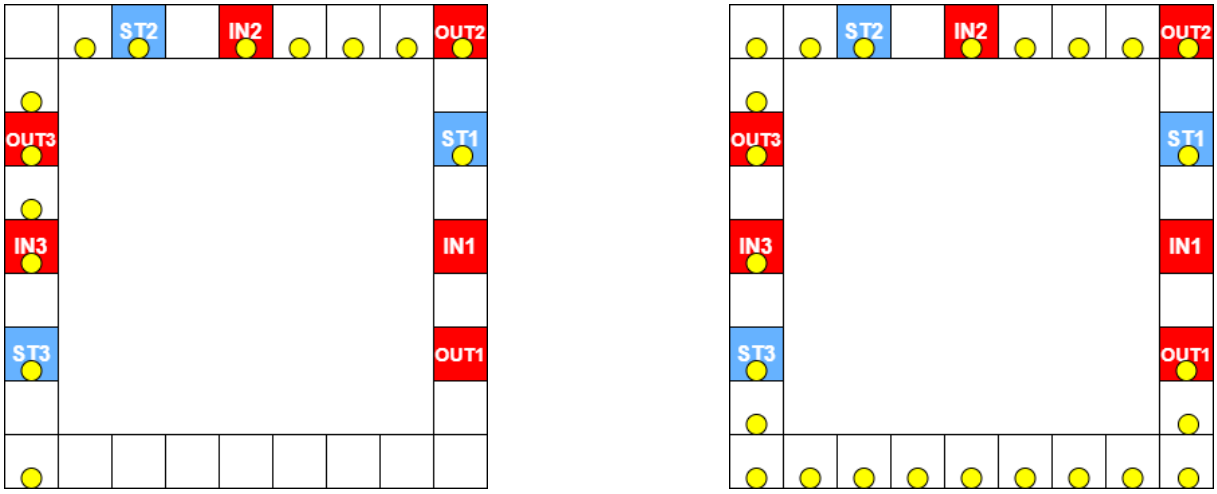
4. **The plant never incurs in deadlock**.
   This property can be easily verified using UPPAAL notation.

   $\forall\Box(\neg deadlock)$

After assessing properties verification on the model present in the project presentation, we designed three different non-trivial configurations to analyze the system behavior and highlight pivotal behavioral aspects of the plant.

5.1.1 Scenario 1 - Bottleneck Station

In the plant configuration represented in Figure 5.5a, *station2* (i.e., ST2) processing time is ten times higher than the rest of the stations. In this case, the whole system will wait for the processing completion of the workpiece in the bottleneck station (i.e., ST2). If the other stations' queues are large enough, the system will not deadlock. Indeed, the workpieces will wait until *station2* frees to start the subsequent piece processing. If the number of workpieces in the system is less or equal to the sum of the lengths of the queues, the system will not deadlock, and the workpieces will wait, stocking in the different queues. In opposition, if the number of pieces exceeds the maximum allowed, every workpiece exiting a processing station will find the following out sensor (i.e., the sensor guarding the queue) busy; hence, every processing station cannot release the workpiece causing a deadlock, represented in. Moreover, if the out sensor is too close to the *In Sensor* and the preceding processing station has a low processing time, the *fast* station may check the out sensor status before it is set to busy since the previously processed workpiece has not already reached it. This configuration may result in a deadlock; hence, placing the out sensors at an appropriate distance from the processing station, keeping in mind each station's processing time, is paramount.



(a) Working bottleneck station configuration

(b) Deadlock bottleneck station configuration

Figure 5.1: Bottleneck station scenario representation

5.1.2 Scenario 2 - Conveyor Belt with Branch

Other than simply demonstrating the corrected functioning of the system with the presence of a controller (the green bar in Figure 5.2), thus with a second branch on the *Conveyor Belt*, we show that we support a property that increases the robustness of our system. If we locate a processing station (*station 1* in Figure 5.2) before the bifurcation, it has to check two possible out sensors (*out sensor 2* and *out sensor 3*) before releasing a workpiece. The latter station will perform a double-check when its processing time elapses. Since we can not predict the future state of the controller, we will verify the worst-case scenario. If at least one of the two queues is full, the piece will remain in the processing station, even if the gate would not lead to the part into the already saturated queue. In this manner, we ensure that the system will not deadlock even if the processing time of the latter stations is really high. This feature could slow down the system but at the expense of increased robustness and error resistance. In our example, we have placed three processing stations, two of which are on the branches: the last-mentioned stations have much smaller tail capacities than the station on the principal belt. With a large number of pieces, we can see how the station waits to free a workpiece to ensure that the smaller queue can always add it to its waiting pieces. In particular, the station1 queue is larger than the station2 queue.

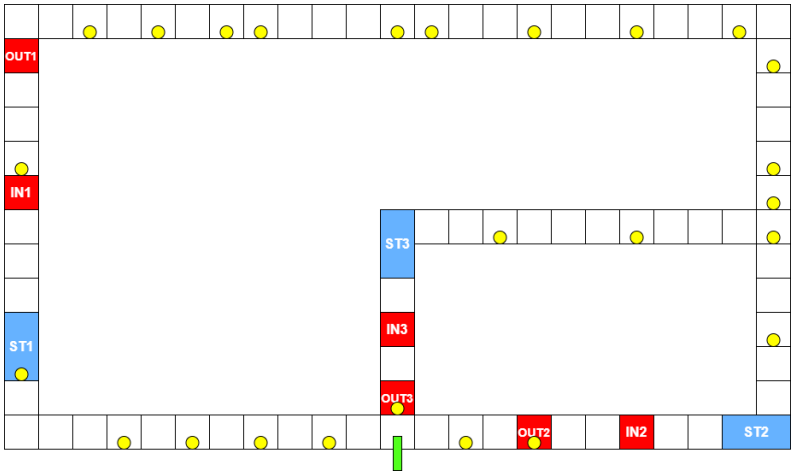


Figure 5.2: Conveyor Belt with Branch scenario representation

5.1.3 Scenario 3 - Infinite Loop

In this configuration, we ensured a system failure. There is only one processing station, with a number of workflow pieces greater than the maximum length of the queue as represented in Figure 5.3. What happens is that there is an infinite loop. The first workpiece enters the processing station, while the other pieces wait for the station to free again. The station cannot release the piece because the sensor guarding the end of the queue detects another workpiece notifying the station. The station will keep doing this check while the output sensor persists to send the same notification. All the pieces are stuck, resulting in an infinite loop. We checked that the following property is never satisfied to assess the infinite loop behavior of this configuration:

$$Processing\_Station(0).PROCESSING \implies Processing\_Station(0).AFTER\_PROCESSING$$

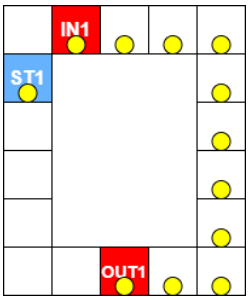


Figure 5.3: Infinite Loop scenario representation

5.2 SMC Version

In the stochastic version of the project, we calculated the probabilities of the properties we tested during the first project phase, checking them within a specific time-bound. We first calculated how much time the first piece on the queue (after the initialization process) takes to reach again its initial position, which approximately estimates a loop duration of a single workpiece. Afterward, we appropriately set the time-bound to ensure that a piece performs at least three or four times the whole belt. Since UPPAAL can verify the deadlock only in a deterministic environment, we implemented the query ourselves. To verify the probability that the system does not incur a deadlock, we check that the automata do not get stuck waiting for a signal from another automaton and vice-versa. In our case, first, we verify that the station is not stuck in the FREE state waiting for `enter_preprocessing[id]` signal fired from the sensor, while the sensor is in the BUSY state waiting for `free_in_sensor[id]` triggered by the corresponding station. If these two are waiting for each other, the system deadlocks. Another situation in which the system can deadlock is when both the station and the sensor guarding the queue placed at the exit of that station are in the state WAIT_CHECK. Again, in this case, both the sensor and the station are waiting for a signal from each other; thus, the system will deadlock. To verify the probability of the other properties, we decided to check the same queries written for the non-stochastic version. As in the previous version, we first verified the properties of the model present in the project presentation and then designed three different configurations, changing sensors' failure probabilities and the processing time distribution of each station.

5.2.1 Scenario 1 - Bottleneck Station

This scenario has the analogous configuration of the first non-stochastic scenario described in the previous section. The processing stations have a probabilistic processing time, but we maintained station 2 as the bottleneck station with processing times higher than the ones of the other stations. We decided to have a high malfunction probability in out sensors, due to this setting, the bottleneck station queue will exceed the maximum length allowed. This malfunction scenario causes the break of the constraint that the queues of the pieces never exceed the maximum allowed length. Antithetically, the deterministic version always respects the queue constraint. Note that even in this case, the system will not deadlock. However,

the workpieces will populate the belt until the slot right after the bottleneck station. We report in Figure 5.4 the behavior over time of the queue length of the bottleneck station.

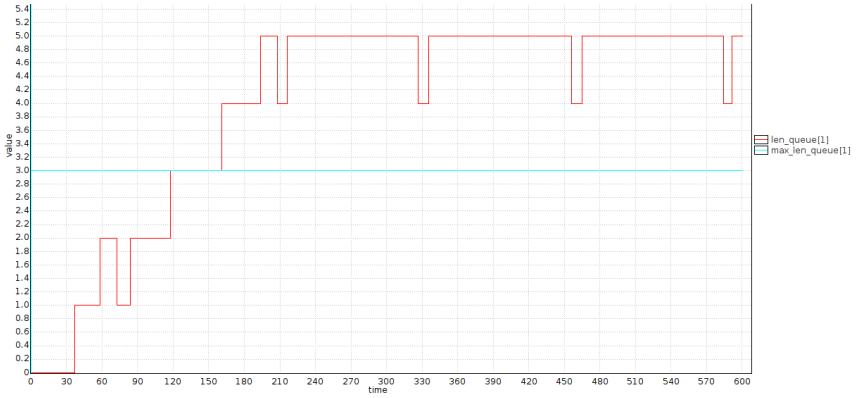
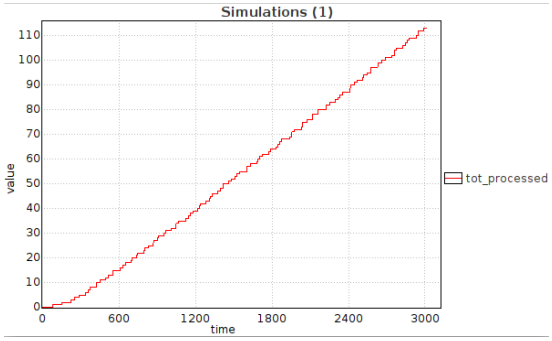


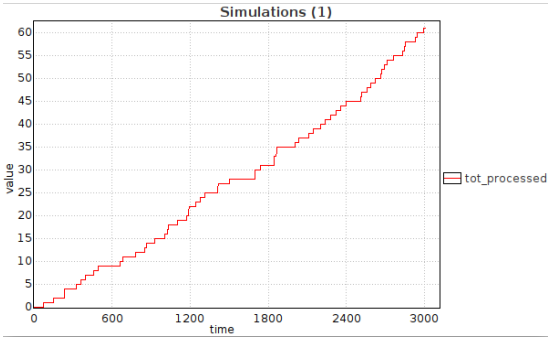
Figure 5.4: Bottleneck station queue length over time in Scenario 1

5.2.2 Scenario 2 - Different sensor malfunction probability

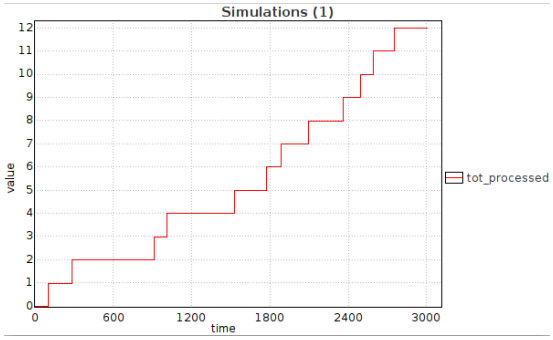
The more we increase the error probability, namely the malfunctioning of the *In Sensor*s, the fewer pieces the system process over time. In the following images, we report the number of pieces processed over time with increasing error probability, respectively 1% (nearly always functioning), 50% (not working about half of the times), and 90% (nearly always not working). We update a counter after the AFTER_PROCESSING stage to count the number of pieces.
The belt configuration we used to perform this analysis is the same as Scenario 2 in the deterministic version, where six stations are present and the belt has a branch. We can conclude that the correct functioning of the input sensor is critical in our system, especially the proper detection of a workpiece, indeed, the increasing number of false negatives degrades the overall performance.



(a) 1% error probability



(b) 50% error probability



(c) 90% error probability

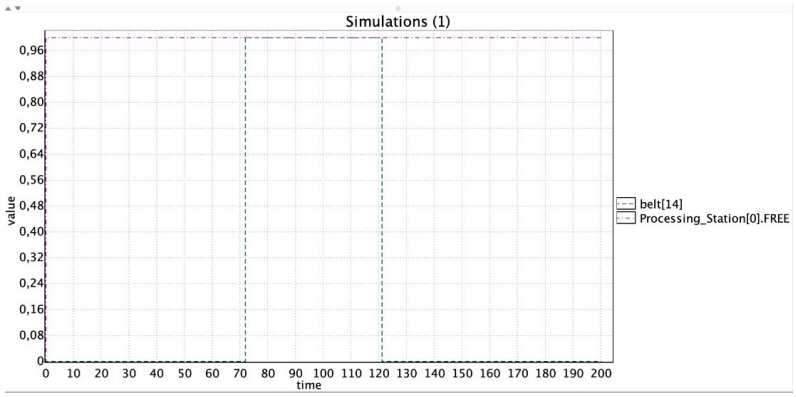
Figure 5.5: Number of pieces processed over time with a certain error probability

5.2.3 Scenario 3 - False-negative workpiece detection

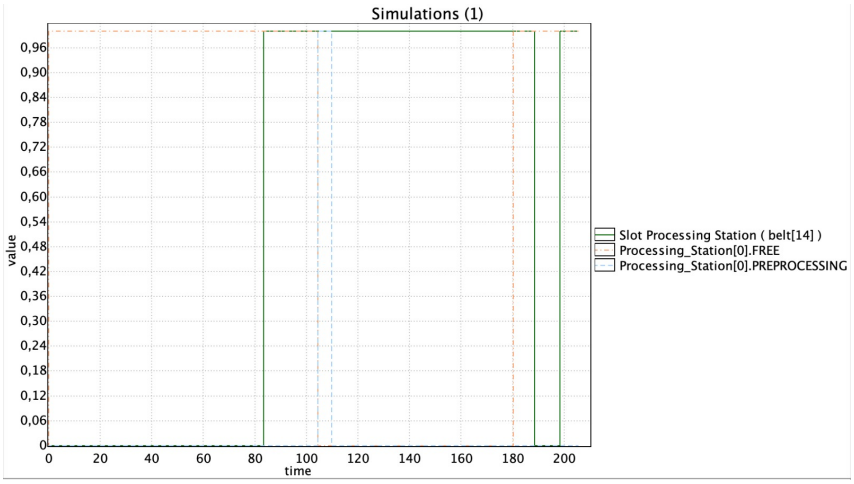
When a malfunctioning *In Sensor* does not detect a workpiece, the latter proceeds toward the preprocessing phase. Assuming that the malfunction probability is low, there are two cases to take care of: the processing station is empty, or it is processing a piece in the instant of the false-negative. In the first case, the workpiece will go through the preprocessing phase without being detected and passes the processing station without being processed. As a consequence, not all the workpieces are processed, in contrast to the project specifications. In the case of an occupied station, the workpiece will pass through the preprocessing even if it shouldn't, being invisible to the system. This piece will wait in the slot right before the processing station until it releases the processed workpiece. The "invisible" workpiece passes through the processing station without activating it. As in the previous case, the system will not process all the workpieces. Moreover, the processing station will release pieces without checking the *Out Sensor* status; thus, the system may exceed the maximum allowed queue length. Note that, even with those kinds of malfunctions, the conveyor belt still works correctly since the workpieces will not overlap waiting for an empty next slot. We expressed this this feature as:

$$Pr[\leq TAU] <> \forall(i : id\_s)\ (Processing\_Station(i).FREE || Processing\_Station(i).PREPROCESSING) \text{ and } belt[Processing\_Station(i).pos\_curr\_station] == 1)$$

In Figure 5.6a, we see that even if the not-detected workpiece enters the station, the processing station status stays free. While in Figure 5.6b, we can see that when the undetected workpiece enters the statio, the station is free, but as the *In Sensor* detects a piece, the station is in the PREPROCESSING state.



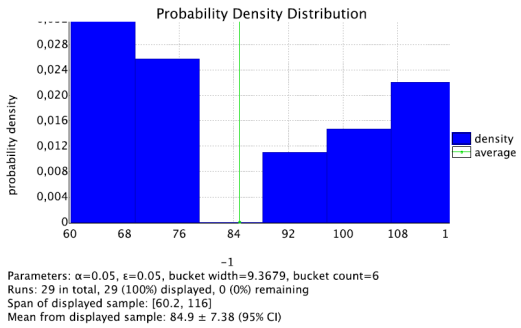
(a) One not-detected workpiece



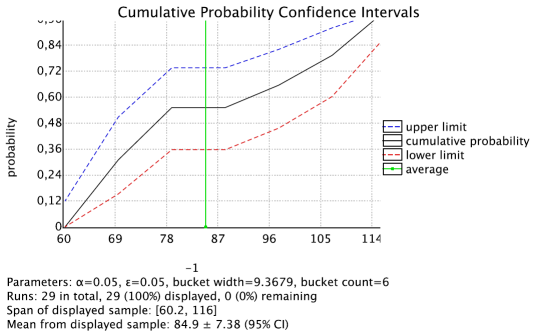
(b) Detected workpiece after an undetected one

Figure 5.6: Station behavior as the undetected workpiece passes through it

In Figure 5.7, we reported the probability distribution of the *In Sensor* failure to detect a workpiece, the probability weight of the error is set at 60%.



(a) Probability Density Distribution



(b) Cumulative Probability Confidence Intervals

Figure 5.7: Probability distribution of undetected workpieces

6 Conclusion

In conclusion, we designed our production plant model to be both flexible and robust, handling belt layouts with a maximum of two branches of the belt. Moreover, we challenged our system with boundary configurations highlighting our system's still correct and solid behavior. While in the stochastic version, we investigated pivotal metrics (i.e., queue length over time, number of pieces processed over time) to emphasize the correctness of our system.