POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Formal Digital Twin of a LEGO Mindstorms Production Plant

Project Report
Formal Methods for Concurrent
and Real-Time Systems

Authors:

Filippo Scaramuzza - 10853428
Riccardo Strina - 10670377
Pietro Lodi - 10669653
Luca Gerin - 10666665

Academic Year: 2022-23

Contents

1 Introduction

1.1. Objective

The industrial system under analysis is composed of a network of belts that carries pieces into stations where they are processed. The flow is controlled with the aid of sensors on the belts which detect the presence of pieces in specific locations and if needed stop the pieces from moving into already occupied stations, by controlling gates on the belt. The topology of the system includes belts' paths that can intersect, and decisions about what direction to send the moving pieces is in charge of the system itself.

We will take care of modeling the system in order to create a digital twin counterpart for it. The aim is to use this digital twin to perform simulation and verification of some relevant properties, possibly along the whole life-cycle of the real industrial plant of interest. This will allow us to evaluate the system's performance and reliability.

1.2. Method

In order to create and analyze the digital twin, the system is represented using timed automata with the aid of Uppaal.
Uppaal is a tool with the objective to support the modeling, simulation, and verification of real-time systems with nontrivial timing behaviors, such as delays, timeouts, and deadlines.
Moreover, the tool has some extensions that allow to additionally consider probabilistic behaviors. This feature of Statistical Model Checking will be utilized in order to extend the scope of the system including possible faulty behaviors of its components.

2 Design

2.1. General Plant Design

2.1.1. Belts

We have modeled the belts as a two-dimensional array of *Integers* representing all of the slots for the pieces on the belts (i.e. the places where the pieces can be located along the belts). In particular, each element `[i][j]` of the two-dimensional array holds a counter of the number of pieces that are on the `j`-th slot of the `i`-th belt (ideally, in a correct working state this counter is always ≤ 1). This modeling choice, to consider also the erroneous case in which there is more than one piece in the same slot, allows to have a more concrete understanding of the behavior of the belt while performing simulation and facilitates the checking of some properties, although it must be noted that it may increase the number of states for verification compared to other approaches such as employing a mono-dimensional array or using a boolean type instead.

2.1.2. Stations and Head/Tail Sensors

Each Station in the plant is characterized by a given processing time and an input and output belt, from which it respectively picks up and drops pieces once they have been processed after the processing time is elapsed. The logic behind the stations' behavior is aided by the presence of sensors, divided into head sensors (always present before the entrance of a station) and tail sensors (optionally located at the beginning of a belt, the distance to the next head sensor define the length of that belt's queue).
The sensors, apart from signaling the presence of pieces in determined positions, have two roles. The first one consists in blocking pieces from entering the stations already busy processing a piece. This is modeled by a *Boolean* array, where each element represents whether the physical gate on the belt indexed `i` is opened or closed, i.e. if pieces are allowed to pass or not. A `True` value means that the gate on belt `i` is open.
The second function is to prevent stations from dropping pieces on the output belt if the elements queuing up on the belt have exceeded the defined queue size. This is also modeled by a *Boolean* array, where each element `i` tells if the station has to wait or not, i.e. if pieces can be dropped on the output belt or not.

2.1.3. Belts Intersections

Belts intersections are modeled in two ways. A belt splitting into two poses the problem of directing each incoming piece in one direction or another. This issue is tackled by the *Flow Controller*, whose job is to direct the piece's flow in the two belts, according to different configurable policies. Two belts whose flow is conveyed into one are instead handled by the *Merger* component, which joins the pieces on the same belt avoiding conflicts.

2.1.4. Plant and Property Configurations

To make the configuration less cumbersome and the property verification more effective, we defined several data structures that serve as configuration data for the different templates of the system that will be employed. In particular, it is possible to configure: the conveyor belt transitional speed, the processing time of each station, the number of pieces circulating in the plant, the maximum length (if any) of the queue upstream of each station, and the branch-switching policy.

We set the plant configuration throughout the whole implementation by following the enumeration of the components shown in Fig. 2.1. In particular:

- The processing stations are numbered with ids from 0 to 5
- The belts are numbered with ids from 0 to 6
- The slots on each belt are numbered with indexes starting from 0
- The head sensors are identified with ids referring to the processing machine they precede
- The tail sensors are numbered with ids from 0 to 4
- There are just one Flow controller and one Merger

We designed the whole system to be able to support different plant configurations, varying in the number of stations, belts, sensors, and so on.

2.2. Stochastic Features

We modeled the following stochastic features:

- The stations' processing time is distributed according to a normal distribution with customizable mean and variance.
- The fact that a certain sensor is working properly or not is modeled through a parametric error probability weight assigned to each sensor.

These features are designed to incorporate uncertainty into the model, capturing behaviors that closely resemble those of the real system.

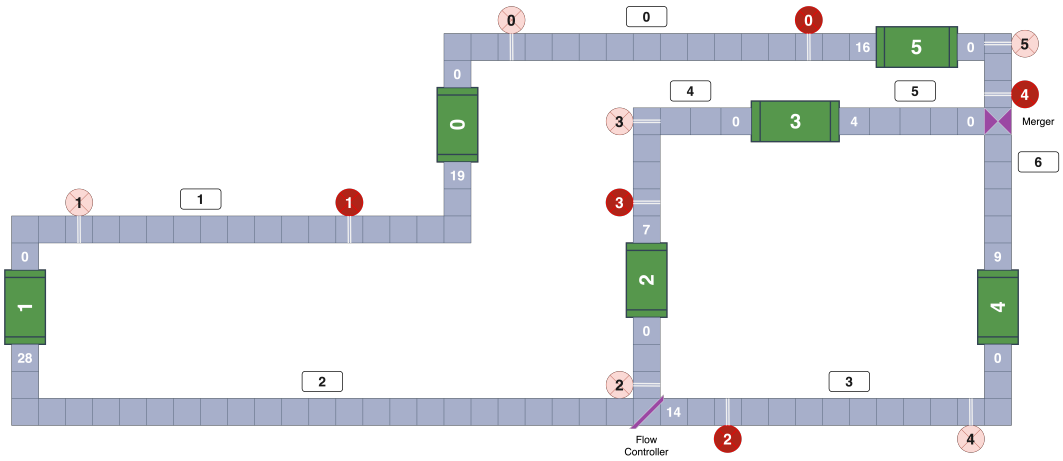

Figure 2.1: System representation.

2.3. Design Assumptions

The following assumptions were made during the design stage and have been verified through implementation:

- Stations become busy as soon as a piece passes through the Head Sensors that guards the Stations' entrance.
- The Flow Controller and Merge component are positioned individually on specific positions of separate belts. Upon activation, the Flow Controller adheres to its policies, either retaining pieces on Belt 2 or transferring them to the initial position of the other belt (Belt 3). Conversely, the Merge component extracts pieces from the final slot of Belt 5 and transfers them onto Belt 6.
- Configurations are made with respect to the enumerations in Fig. 2.1.

3 UPPAAL Model

3.1. Global Declarations

3.1.1. System Parameters

Table 3.1: Constant system parameters.

Parameter	Description
HEAD_SENSORS	number of head sensors
TAIL_SENSORS	number of tail sensors
BELTS	number of belts
STATIONS	number of stations
BELT_SPEED	amount of units of time needed by the belt to move
MAX_SLOTS	maximum number of slots in a belt

3.1.2. Configurations Data Structures

Belt Configuration

The belt configuration's data structure is defined as follows:

```
typedef struct {
    pos_t head, tail, length; // typedef int[-1, MAX_SLOTS] pos_t;
    int pieces;
} belt_config_t;
```

Where `head` is the position on the belt of the Head Sensor, `tail` is the position on the belt of the Tail Sensor, `length` is the number of slots the belt is made of, and `pieces` is the total number of pieces to be placed on this belt during initialization. The reason for incorporating a multitude of parameters into this component is due to its pivotal role in our modeling process.

Station Configuration

The Station configuration's data structure is defined as follows:

```
typedef struct {
    int processing_time;  //  Units of time to process a piece
    belt_t input, output; // typedef int[0, BELTS-1] belt_t;
} station_config_t;
```

Where `processing_time` is the time taken by the station to process a piece, `input` and `output` are respectively the input and output belts ids.

Sensors Configuration

```
typedef struct {
    belt_t belt;        // typedef int[0, BELTS-1] belt_t;
    pos_t pos;          // typedef int[-1, MAX_SLOTS] pos_t;
} sensor_config_t;
```

Where `belt` is the belt onto which the sensor is placed and `pos` is the specific slot of the belt where the Sensor is.

3.1.3. Functional Data Structures and Synchronization

As mentioned before, the belt is defined as a two-dimensional array:

```
int[0,2] belts[BELTS][MAX_SLOTS];
```

The range `[0, 2]` enables the UPPAAL verifier to examine properties concerning the number of pieces in each belt slot at any given time. By allowing up to two pieces in a single slot, we have accounted for the scenario we aim to prevent but have precisely modeled.

The faculty of the head gates to block or allow the passing through of pieces is modeled employing a *Boolean* array:

```
bool gate[BELTS];
```

A piece is allowed to flow after the sensor on belt `i` when the value at the index corresponding to the belt is set to *True*.

Similarly, the Tail Sensors blocking the stations from pushing pieces to the output belts are modeled as a *Boolean* array:

```
bool wait[BELTS];
```

A station is allowed to load a piece onto the belt `i` when the value at the index of the output belt is set to *False*.

To allow synchronization between components, in addition to shared variables, we have defined the following channels:

- `init_done` used to notify the templates that the initialization phase is over
- `tick_components` used by the Belt template to assert that its clock has reached `BELT_SPEED`
- `move` sent by the belt, after `tick_components`, to notify everyone it hash performed one move
- `free[station_id]` used to synchronize a particular Station and its related Head sensor about the status of a particular station

3.2. Templates

3.2.1. Station

The Station template takes care of the processing of pieces. Upon receiving the `tick_components` signal, all the stations will look at the first slot of their input belt; if a piece is there it will be loaded and the station will start that piece's processing and stay in a processing state for `processing_time`. When the processing is over it will wait for the move signal to be received and only if the wait variable is set to `false` load the processed piece onto the output belt. Before returning to the initial state, it will update the Head Sensor by signaling to it that the station is ready to receive a new piece.

3.2.2. Belt

The Belt template is in charge of the movement of all the pieces on the belt. The speed of the belt is modeled with a clock. Each time this clock reaches the value of `BELT_SPEED`, the `tick_components` signal is sent to notify components about the imminent movement of the belt, allowing them to perform those operations that need to be performed just before the movement happens. Then the belts move all the pieces on them, and fire the `move` signal to notify that the pieces have been moved, allowing the components to update their states.

3.2.3. Head Sensor

The Head Sensor template ensures that the stations do not receive more than one piece for processing simultaneously. It monitors both the presence of a piece on the belt and the status of the adjacent station: when the station is idle and ready to receive a piece, the sensor detects the entry of a new piece and transitions to a different state, closing the gate on the corresponding belt to effectively preventing additional pieces from entering. The gate remains closed until the station signals its availability once again.

3.2.4. Tail Sensor

The Tail Sensor is responsible for efficiently handling the queue on its corresponding belt. When the sensor detects the presence of a piece in front of it, it signals the station preceding it to refrain from placing a processed piece on the belt. The sensor's primary function is to communicate this unavailability to the preceding station. Only after the piece has moved past the sensor's position will the preceding station be permitted to load a processed piece onto the belt.

3.2.5. Flow Controller

The Flow Controller template manages the outgoing intersection of the belts, where the direction for each piece needs to be determined. Based on the policy schedule, a piece can either remain on its current belt or be

directed to a different position. This decision is made upon receiving the `tick_components` signal.

3.2.6. Merger

The Merger template handles the incoming intersection of the belts, where the flow from one belt merges with another. Whenever the position before the merger on the receiving belt is unoccupied, ensuring no conflicts in the desired placement of the piece, the merger will load any piece that has reached position 0 on the incoming belt. This mechanism ensures smooth integration of the incoming flow into the target belt. This operation is performed upon receiving the `tick_components` signal.

3.2.7. Initializer

The Initializer template is responsible for placing pieces on the belt at the start of the simulation. Once this task is complete, it signals all other templates that the initialization process is finished by firing the `init_done` signal.

4 Analysis and Results

4.1. Properties

TCTL formulas provide a means to verify the model and determine if the desired properties hold true. The verification process can be conducted on both the deterministic and stochastic versions of the model.

The verification time for the deterministic model has a strong dependency on the processing time of the stations and the pieces loaded onto the belt. In fact, with low processing time, we were able to verify configurations with a lot of pieces, while high processing time made it difficult to efficiently run the verification even with fewer pieces present in the system. In particular, we noticed that with the second configuration, we were able to verify the properties very quickly, even with 30 work-pieces in the system.

The configurations used for the deterministic version are the following:

Configuration Description	Station's Processing Time	Belt's Speed	Flow Controller's Policy Schedule
1: Slow stations, fast belts	[6, 6, 3, 3, 6, 6]	1	Round-Robin
2: Fast stations, slow belts	[2, 2, 1, 1, 2, 2]	6	Round-Robin
3: Random values config	[3, 5, 4, 4, 8, 6]	3	Round-Robin

Utilizing the stochastic version, we successfully verified the properties across four distinct configurations, each involving a significantly higher number of work-pieces on the belt. This expanded scope allowed us to thoroughly test the system's behavior and assess the properties under diverse scenarios.

The properties listed below were essential and mandatory for us to verify:

1. the model never incurs into a deadlock state
2. the belt never holds more than one piece on each slot
3. stations cannot process more than one piece at a time
4. the queues' max length is never exceeded

All of these are found under the tab *Verifier* of Uppaal's project files.

Configuration Description	Station's Processing Time (Mean, Variance)	Belt's Speed higher means slower	Sensor's Error (Prob %)
1: Fast belt, slow stations, good sensors	[(6, 2), (6, 1), (4, 1), (4, 1), (7, 2), (7, 2)]	1	2
2: Slow belt, fast stations, good sensors	[(3, 1), (3, 1), (2, 1), (2, 1), (4, 2), (3, 1)]	6	2
3: Fast belt, slow stations, bad sensors	[(6, 2), (6, 1), (4, 1), (4, 1), (7, 2), (7, 2)]	1	10
4: Slow belt, fast stations, bad sensors	[(3, 1), (3, 1), (2, 1), (2, 1), (4, 2), (3, 1)]	6	10

4.1.1. Not deadlock

This property ensures that the model, across all possible paths, does not encounter a deadlock. The objective of the deterministic version was to demonstrate the model's resilience against deadlocks.

$$\forall \, \Box \, \neg \text{deadlock}$$

In contrast, in the stochastic version, it is not feasible to entirely prevent deadlocks due to the possibility of sensors malfunctioning. Consequently, the property examines the probability of the model not encountering a deadlock within a specified time-bound. This approach acknowledges the inherent uncertainty and assesses the likelihood of deadlocks occurring based on the stochastic nature of the system.

$$(\boldsymbol{\psi}) : \forall \Box (\text{Initializer.init\_over} \Rightarrow \exists i : \text{station\_t}\, \text{Station}(i).\text{waiting} \wedge$$
$$\text{Belt.check\_deadlock}(\text{Station}(i).\text{input}()) \vee \exists j : \text{station\_t}\, \text{Station}(j).\text{error} \vee \text{FlowController.error})$$

Configuration	Probability
1: Slow stations, fast belts, good sensors	$\mathbb{P}(\Box_{\leq 10^4} \psi) \leq 0.0499441$
2: Fast stations, slow belts, good sensors	$\mathbb{P}(\Box_{\leq 10^4} \psi) \leq 0.0499441$
3: Slow stations, fast belts, bad sensors	$\mathbb{P}(\Box_{\leq 10^4} \psi) \leq 0.0499441$
4: Fast stations, slow belts, bad sensors	$\mathbb{P}(\Box_{\leq 10^4} \psi) \leq 0.0499441$

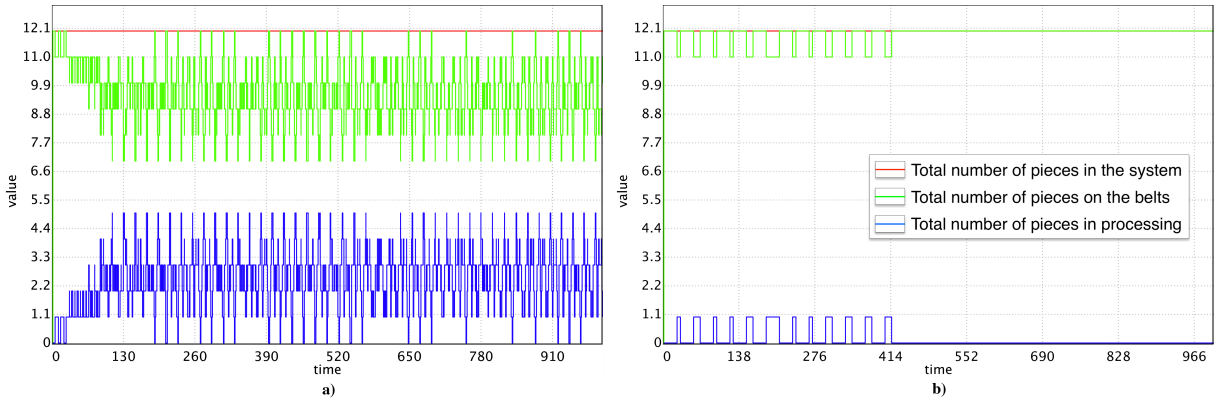

Figure 4.1: Work-pieces' flow simulations with Uppaal `simulate` with $\tau = 1000$ **a)** deterministic version, no deadlock reached **b)** stochastic version, deadlock reached.

4.1.2. Stations process one piece at a time

This property verifies that the stations can only accommodate one piece at a time during the processing phase. By design, this requirement is always met, as each station is capable of processing only one piece at a time.

Consequently, this property examines whether the total number of pieces, considering both the piece currently being processed and the one on the belt, is consistently equal to the number of pieces initially loaded onto the belt. This validation ensures that the system maintains the expected balance between the number of loaded pieces and the pieces undergoing processing.

$$(\boldsymbol{\psi}) : \forall\, \Box\, (\text{Initializer.init\_over} \Rightarrow (\text{sum}(x : \text{station\_t})\ \neg\text{Station}(x).\text{waiting}) +$$
$$(\text{sum}(i : \text{belt\_t})\text{sum}(j : \text{belt\_pos\_t})\text{belts}[i][j]) = \text{sum}(k : \text{belt\_t})\text{belt\_conf}[k].\text{pieces})$$

Configuration	Probability
1: Slow stations, fast belts, good sensors	$\mathbb{P}(\Box_{\leq 10^4}\psi) \leq 0.0499441$
2: Fast stations, slow belts, good sensors	$\mathbb{P}(\Box_{\leq 10^4}\psi) = 0.0412821 \pm 0.0356734$
3: Slow stations, fast belts, bad sensors	$\mathbb{P}(\Box_{\leq 10^4}\psi) = 0.0499441$
4: Fast stations, slow belts, bad sensors	$\mathbb{P}(\Box_{\leq 10^4}\psi) = 0.0306514 \pm 0.0303669$

4.1.3. Never more than one piece on a slot

This property verifies that no two pieces occupy the same slot on a belt. This is accomplished by checking that the number in the matrix representing the belt remains lower than 2. This approach effectively ensures the property, as the movement of the pieces is managed by incrementing or decrementing the previous value in the array.
In the deterministic version, it is imperative that this property holds true for all paths without exception.

$$(\boldsymbol{\psi}) : \forall\, \Box\, (\text{Initializer.init\_over} \Rightarrow \forall\, (i : \text{belt\_t})\ \forall\, (j : \text{belt\_pos\_t})\ 0\ \leq \text{belts}[i][j] < 2)$$

In the stochastic version, due to the nature of the sensors, this property examines the probability of the situation where two pieces occupy the same slot on a belt.

Configuration	Probability
1: Slow stations, fast belts, good sensors	$\mathbb{P}(\Box_{\leq 10^4}\psi) \leq 0.0499441$
2: Fast stations, slow belts, good sensors	$\mathbb{P}(\Box_{\leq 10^4}\psi) \leq 0.0499441$
3: Slow stations, fast belts, bad sensors	$\mathbb{P}(\Box_{\leq 10^4}\psi) \leq 0.0499441$
4: Fast stations, slow belts, bad sensors	$\mathbb{P}(\Box_{\leq 10^4}\psi) \leq 0.0499441$

4.1.4. Length of the queue never exceeded

This property verifies that each belt never exceeds its allowed queue length, ensuring that the number of pieces on a belt remains within the limits defined by the specified queue lengths. In the deterministic version, this behavior is strictly enforced for all paths without exception.

$$(\boldsymbol{\psi}) : \forall\, \Box\, (\text{Initializer.init\_over} \Rightarrow \forall\, (i : \text{belt\_t})((\text{sum}(j : \text{belt\_pos\_t})\text{belts}[i][j]) \leq \text{Belt.max\_queue}(i)))$$

In the stochastic version, we assess the probability of the unwanted situation where a belt contains more pieces than its allowed queue length.

Configuration	Probability
1: Slow stations, fast belts, good sensors	$\mathbb{P}(\Box_{\leq 10^4}\psi) \leq 0.0499441$
2: Fast stations, slow belts, good sensors	$\mathbb{P}(\Box_{\leq 10^4}\psi) \leq 0.0499441$
3: Slow stations, fast belts, bad sensors	$\mathbb{P}(\Box_{\leq 10^4}\psi) \leq 0.0499441$
4: Fast stations, slow belts, bad sensors	$\mathbb{P}(\Box_{\leq 10^4}\psi) = 0.0412821 \pm 0.0356734$

4.1.5. No pieces between a station and its head sensor during processing

This additional property verifies that no pieces are present between a station that is actively processing and its corresponding guarding sensor. By ensuring that no pieces are positioned between the processing station and the guarding sensor, we confirm the proper operation of the head sensor, as it effectively detects and prevents any unwanted pieces from entering the processing area.

$$(\psi) : \forall\, \square\, (\text{Initializer.init\_over} \implies (\forall(i : \text{station\_t}) \neg \text{Station}(i).\text{waiting} \implies ((\text{sum}(j : \text{belt\_pos\_t})(j < \text{HeadSensor}(i).pos())?\text{belts}[\text{Station}(i).\text{input}()][j] : 0) = 0))$$

Configuration	Probability
1: Slow stations, fast belts, good sensors	$\mathbb{P}(\square_{\leq 10^4}\psi) = 0.45086 \pm 0.0496008$
2: Fast stations, slow belts, good sensors	$\mathbb{P}(\square_{\leq 10^4}\psi) = 0.0599014 \pm 0.0397112$
3: Slow stations, fast belts, bad sensors	$\mathbb{P}(\square_{\leq 10^4}\psi) = 0.0306514 \pm 0.0303669$
4: Fast stations, slow belts, bad sensors	$\mathbb{P}(\square_{\leq 10^4}\psi) = 0.0641003 \pm 0.040315$

4.2. Simulations

In this chapter, we will present the results of simulations that were run to investigate how the configurable variables interact with the system. We wrote a Python script that automatically tests for deadlock probability with different combinations using the Uppaal `verifyta` command line utility.

At first, we simulated the system while dynamically changing the stochastic parameters: stations' mean processing time (`PROCESSING_TIME_MEAN`), variance (`PROCESSING_TIME_VARIANCE`), and sensor's error probability (`SENSOR_ERROR`). The result is shown in Fig. 4.2.a.

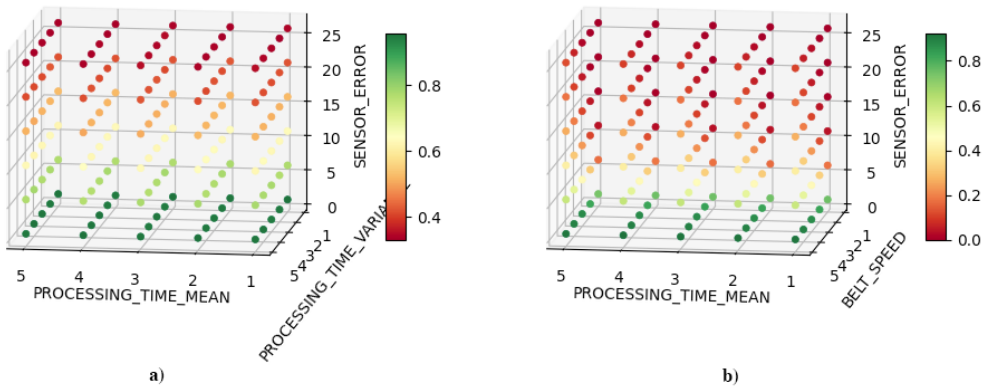

Figure 4.2: **a)** Faulty behaviors w.r.t. stochastic features ran with $\tau = 50$. **b)** Faulty behaviors w.r.t. stochastic features and belt speed ran with $\tau = 50$.

The results showed that the deadlock probability depends mostly on the `SENSOR_ERROR` parameter. To further investigate other configurable parameters we simulated the system while dynamically changing the following parameters: stations' mean processing time (`PROCESSING_TIME_MEAN`), belts' speed (`BELT_SPEED`), and sensor's error probability (`SENSOR_ERROR`). The result is shown in Fig. 4.2.b.

Again, we noticed how also in this case the processing time doesn't affect the deadlock probability while the belt speed and sensor error appear to be relevant, making the system less prone to deadlock with slower belts and better sensors.

5 | Conclusions

The implemented digital twin of the industrial system provided valuable insights into its performance, reliability, and adherence to key properties throughout its life cycle. The analysis allowed for the identification of potential issues and the evaluation of system performance under various scenarios.

The successful verification of the properties in the deterministic version allowed us to conclude that the system we modeled indeed represents the ideal one when considering optimal working conditions.

However, the introduction of stochastic features, reflecting real-world uncertainties, revealed potential errors. Notably, the probability of encountering undesired behaviors, such as deadlocks, increased exponentially with the sensor error rate. This aligns with the design of the system, where sensor failures could rapidly lead to system failures due to frequent employment of the sensors. When a sensor malfunctions, pieces may be moved to incorrect locations or stations could suffer from "starvation" as no new pieces are allowed to enter despite the station being idle.

To enhance system reliability, several improvements can be considered. For example, redundant sensors could be introduced to mitigate the impact of sensor failures or allow stations to explicitly signal the sensors to open the gates could ensure a more efficient flow of pieces avoiding the problem of starvation.