POLITECNICO
MILANO 1863

Formal Methods for Concurrent and Real-Time Systems

A.Y. 20/21

MODEL CHECKING OF WAREHOUSE ROBOTICS

Authors

Arianna Casaroli
Davide Porta
Martino Manzolini

Instructors

Prof. Pierluigi San Pietro
Dott.sa Livia Lestingi

1. High Level Model Description

The project consists in creating a model to simulate an automated warehouse using UPPAAL. In our model there are a certain number of robots and a controller.

The controller has the role to publish new tasks from a stack, removing those that are claimed by robots and handle the requests for claiming.

The robot, accordingly, can claim a task and move around the grid to pick, transport or release a pod.

Both robots and controller start in an idle status until the first tasks are generated. When a task is generated and randomly assigned to a pod, the first robot (currently free from tasks) that will open a request to claim it, will receive the task from the controller. Once the task is claimed the controller removes it from the stack meanwhile the robot will start to move itself to reach the pod and to pick it up with the objective to bring it in front of the delivery point. At the delivery point, according to the specifications of the homework, a human will have to withdraw the package, so that the robot can once more pick up the pod and crawl back to the original position of the pod to let it down. Once the robot is taken back, the robot returns in idle status waiting for a new task to claim.

2. Initial assumptions

- The space through which robots and pods move is simplified and is a grid.
- Robots can only move vertically and horizontally, not diagonally.
- Only one robot or pod can occupy one cell at once.
- Robots never stay in idle below a pod.
- Robots can move below pod that are waiting to be picked up.
- Tasks are taken with FIFO policy and don't have a deadline for completion.
- Human operator doesn't require a queue for waiting tasks.
- Human's work is represented inside the component "robot", and not by another independent component.
- It never happens that an incoming task is lost.

3. Component description

3.1. Controller

The Controller lifecycle it is divided in 2 distinguished states:

Idle: this is the initial state where the control starts by immediately taking the transition, and so the following functions:

- *initPodList()* which assigns to each pod the coordinates x and y of its position in the grid
- *initRequestsList()* which initialize the state of every robot task request to "refused" (value -1)

- *initDelay()* which is used to define a new randomic delay for the task spawn, according to a normal distribution characterized by the mean "meanT" and the standard deviation "devT".
- *initRobot()* which set the variable RobotReady to "true".

POD_LIST_INIT: this is the state where the control of the publishes, claims and requests of tasks are handled through three self-loops:

- *publishTask():* this function first creates a list of pods that are checked to be free from robots and then, after one of them is randomly picked, the pod state will be updated to let the system know that now it has been assigned to the task. As consequence, the list of free pods will be decreased by one. Instead, the list of published tasks is then increased by one slot and the state of this new request (which can be requested by a robot or not yet, depending on the robot availability at the moment) is updated. The last check is done in the case the queue of tasks is already full which will cause the increment of a variable overflow counter.

- *handleRequests():* this function checks for all robots, if a robot request has not been handled yet. After this check, If the maximum number of tasks that can be managed at the same time it has not yet been reached, and the request status of a certain task is different from that of the same task in the list of tasks, the function increases a counter that will be used to check if the number of current requests has exceeded the maximum of tasks that can be solved simultaneously. Once the checks are passed, if the request has been accepted, we increment the task queue and update the start of the queue. The same function also takes care of decreasing the task counter or updating the status of a request with a refusal if the number is excessive.

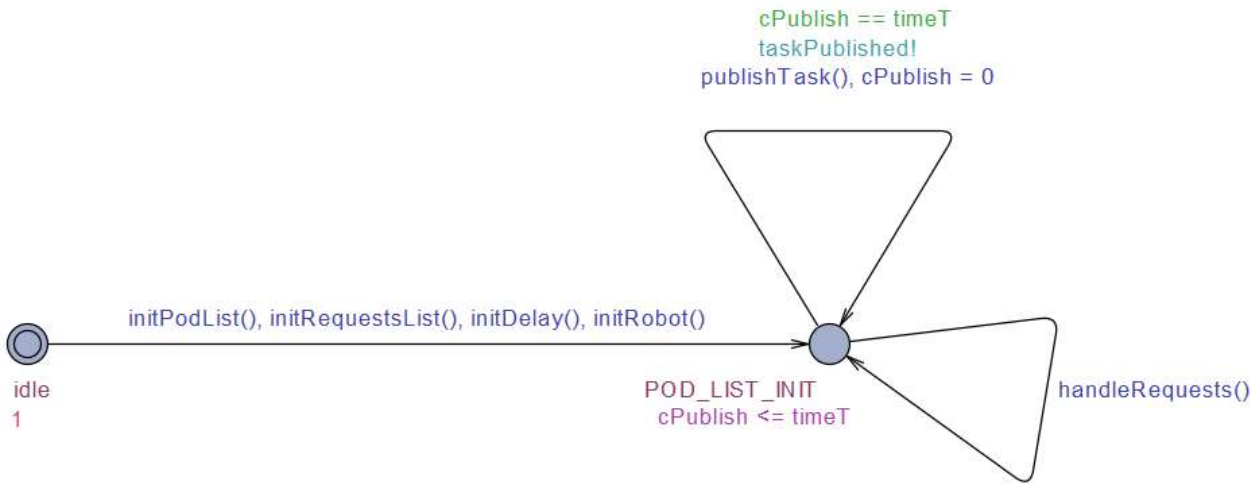

Figure 1: The Controller template

3.2. Robot

The robot, which has a more complex structure than the controller, is composed by 6 different states:

Idle, this is the initial state where the robot just starts and then directly takes the transition RobotReady to end up in the second state called *"Ready".*

Ready, in this state are published new notification about the publishing of new tasks and are checked the status of requests through the following functions:

- *newTaskNotification()*: The robot tries to ask for the first available task. To do so, it's done a check on all the tasks in the task queue to find to skip both the empty slots and those which have already been requested by a robot. Once that task is found, the function will set the robot to request that task (so also the task will be signed as "requested"). Last step is to update, in the list of requests, the status of the task requested by the robot.
- *checkRequestStatus():* if a certain request is accepted saves the ID of the pod that has been assigned to it.
- *takeTask():* the role of this function is first to set the current destination with the coordinates of the targeted pod position and then to assign that pod to the robot itself.

Task_accepted, in this status the robot roams around the grid aiming to reach the position specified in the destination previously mentioned. This is possible through:

- *move():* through the coordinates of the destination and the calculation of a variable delta to understand the direction, the robot will move in the grid trying to avoid other robots which are also "walking" around. . If an "obstacle" is found in the way, according to which direction the robot is moving (up-down or left-right), it will choose the other way. If a robot gets stuck by other two robots on both coordinates that needs to crawl to achieve the destination, it will choose another random direction in order to avoid deadlocks. In this function the robot can also pass below that are "resting" if it is not carrying any pod.
- *liftpod():* this function sets the pod as "not assignable" but doesn't "block the floor" as the robot is carrying it.

Pod_position_reached, this status represents when the robot finally achieved the destination and picked up the pod and now, he has to move it to bring it to the delivery point:

- *movePod():* the structure and the functionalities of this function are pretty much the same of the function *move()* explain in the previous paragraph but, since in this case the robot is carrying the pod on itself, it is not allowed to pass below other pods in idle.
- *initHumanReaction():* This function is used to simulate the behave of the human operator which takes in charge the pods at the delivery point. To do so is defined a random time for the human activity, according to a normal distribution characterized by the mean meanH and the standard deviation devH.

Delivery_point_reached, here is when the delivery point has been reached, the task handled by the "*human operators"*, and the variables of the destination are updated in order to bring the pod back in its original position.

Returning_Pod, this status is the following action of the one before: the robots is now moving towards its final destination to return the pod. The last two functions used are:

- *movePod():* it's the same function explained previously where the robot roams without passing below pods in idle.
- *releasePod():* the robot that was transporting the pod, it releases it so the robot is set in "waiting" status. Also, the state of the pod gets updated and the new destination of the robot is set to be the entry point: the original place where it started its journey.

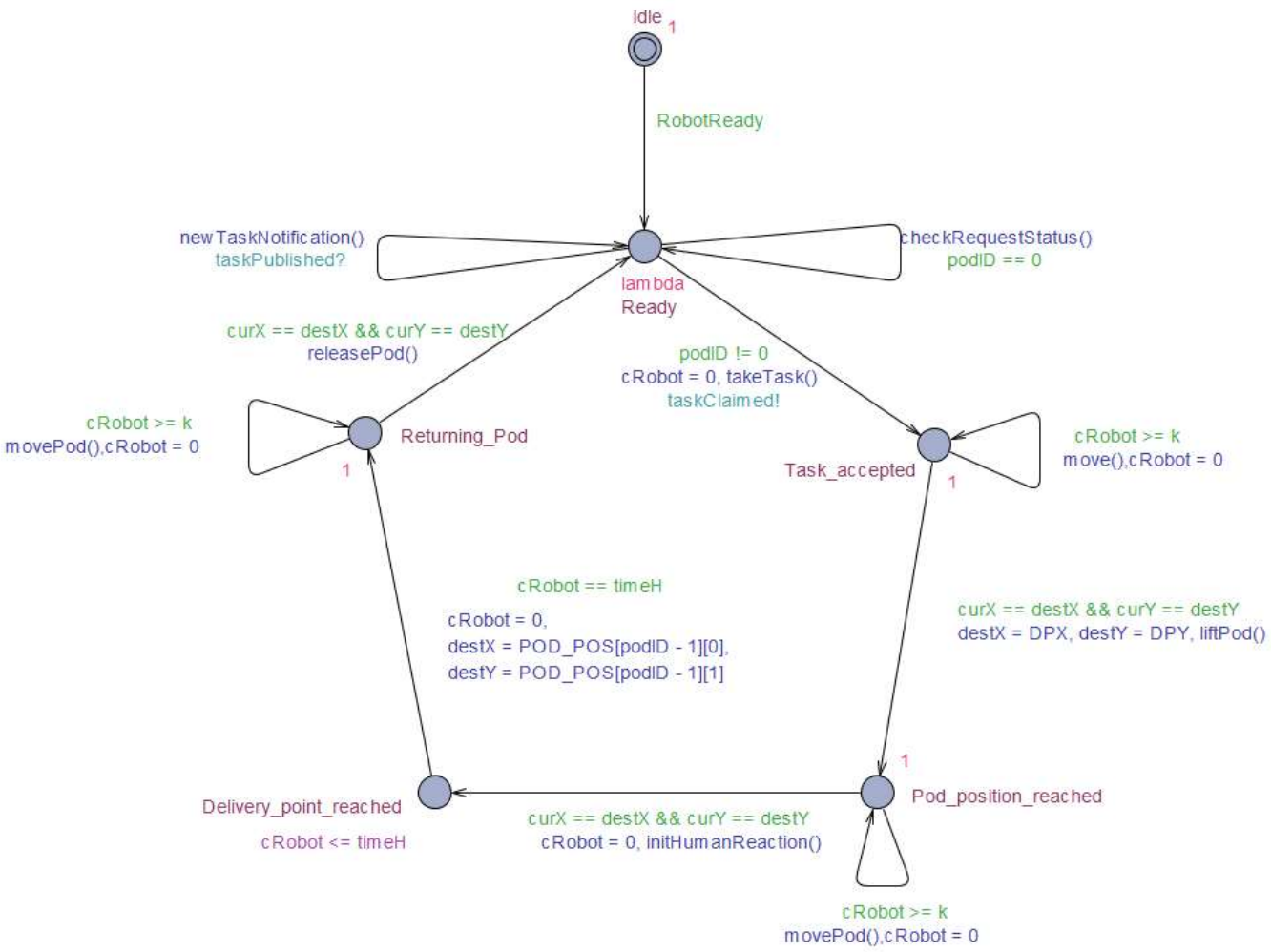

Figure 2: The Robot Template

3.3. Design choices

Random approach has been used in different functions of this project such as, for example, in the publishing of task when a robot is picked for the assignment of it. Another example can be found in the random delay for the task spawn, still in the template of the controller. The choice of using such random approaches is to simulate, as much as possible, a realistic environment where such activities are not deterministic and predictable.

No human template has been inserted in the project, but the action of the human operator has been directly included inside the robot scheme. This choice has been done trying to simplify the model since the only interaction that the operator should have had it would have been the one with the robot.

A **FIFO task queue** has been used when the controller pushes new requests and when robots request for tasks, in order to complete all the tasks in a reasonable similar time to each other. To do that we employed an array which memorize each time the position of the start and the end of the queue, so that the robot will begin to check the availability of tasks starting from the initial part of such queue.

A **Congestion handling strategy** has been applied whenever the robot is not allowed to proceed in the direction to reach its destination position, to avoid a state in which all robots cannot move, we introduced a mechanism which allows to make a movement even if this move doesn't allow the robot to reach the destination. The movement that the robot will do in this case is to "go back", according to the planned trajectory of his path, by one cell. The only purpose of this is to untangle the robots from a situation of deadlock.

4. Properties and scenarios

In this paragraph of the document, we present the properties and scenarios that we used for the verification of the expected behavior of the system that we have modeled. The properties we discuss here can be found as queries in the Verifier tab of the project. We created 4 different "scenarios", by setting different parameters for each scenario, in order to study different situation of the same property. Two of these scenarios are studied to respect the properties that will be explained further in this document, and two of them are supposed not to respect them.

Here follows the table where the different value of the parameters is summarized:

4.1. Scenarios comparing table and short description

Parameter	Scenario 1	Scenario 2	Scenario 3	Scenario 4
X, Y	10, 10	12, 9	7, 13	14, 9
MAX_R	6	6	7	5
MAX_T	6	9	9	9
MAX_P	12	15	16	14
EPX, EPY	9,0	11, 0	0, 0	13, 0
DPX, DPY	9,9	11, 8	6, 12	0, 8
meanH	15	5	12	15
devH	0.15	0.25	0.25	0.12
K	1	3	2	2
Lambda	1	1	2	1
meanT	45	70	58	100
devT	0.2	0.12	0.20	0.15
Respect the properties	Yes	No	Yes	No

Table 1: Parameters of the scenarios

Here are briefly explained the parameters in the previous table:

- X, Y: Floor length (x) and height (y)
- MAX_R: maximum robot fleet number
- MAX_T: maximum number of tasks at the same time
- MAX_P: maximum number of pods
- EPX, EPY: entry point coordinates
- DPX, DPY: delivery point coordinates
- meanH: μT for human operator
- devH: σT for human operator
- K: time units for a robot to move
- Lambda: lambda delay before a robot claims the next task
- meanT: μT for a task
- devT: σT for a task

Scenario 1: this scenario meets the requirements and has a low probability to reach the maximum number of possible tasks.

Scenario 2: this scenario has been designed to violate the task number property, even with a higher delay of spawn higher (meanT2 > meanT1) than the previous scenario. The percentage is determined by the last query.

Scenario 3: In this case the properties are not violated, like in scenario 1, but here we can see that a larger fleet of robot (7 instead of 6) allows the machine to move slower (2 time units instead of 1).

Scenario 4: This set of parameters describes a system similar to scenario 2 as it violates the properties by exceeding the maximum number of tasks. However, in this case, we have a larger task queue and a smaller fleet of faster robots (k = 2, instead of 3).

4.2. Properties

4.2.1. Robot reaching the return_pod state

This property concerns the possibility for a robot (robot0 in particular), to eventually reach the return_pod state, in CTL formula:

$$\forall \Diamond \; robot0.Returning\_Pod$$

In other words, this is the range of probability of completing all the previous actions to pick up the pod, deliver it to the human operator, take it back to its original position and drop it. In order to guarantee significant values, this property is checked with 100 runs:

Uppaal formula	***Scenario 1***	***Scenario 2***	***Scenario 3***	***Scenario 4***
Pr[<=5000;1000](<>robot0.Returning_Pod)	≥ 0.997	≥ 0.997	≥ 0.997	[0.9025, 0.9370]

Table 2: Probability of Property 1

For scenarios 1,2 and 3 we consider the property verified since the probability is ≥ 0.997, which means the robot always reach the return-pod state with this time bound. The chances instead are not certainly verified in scenario.

4.2.2. Maximum number of tasks possible in the queue

This property considers the event of eventually reaching the maximum number of tasks possible in the queue by the controller, after which will intervene the overflow. We can define this property using the CTL formulation:

$$\forall \Diamond \; controller.nTasks == controller.maxT$$

This check is done with 50 runs, a number sufficient to provide consistent values.

Uppaal formula	***Scenario 1***	***Scenario 2***	***Scenario 3***	***Scenario 4***
Pr[<=5000;1000](<> controller.nTasks == controller.maxT)	[0.0347, 0.0620]	*[0.6664, 0.7244]*	[0.0237, 0.0472]	[0.9517, 0.9755]

Table 3: Probability of Property 2

For scenarios 2 and 4 the chances of eventually reaching the maximum number of tasks allowed in the queue is significative, in particular the 2nd one. In scenario 1 and scenario 3, instead, the possibility is low and almost negligible.

4.2.3. Excess in the number of tasks in the queue

Consequently to the previous property of the controller, with this last one we analyze the range of probabilities of eventually exceeding the number of tasks in the queue, in CTL formula:

$$\forall \Diamond controller.overflowCounter > 0$$

In other words, is the probability of using the task overflow when they become too many to be handled in the queue. Checked over 50 runs.

Uppaal formula	***Scenario 1***	***Scenario 2***	***Scenario 3***	***Scenario 4***
Pr[<=5000;1000](<>controller.overflowCounter > 0)	[0.0022, 0.013]	[0.4745, 0.5374]	[0.0076, 0.0233]	[0.9223, 0.9530]

Table 4: Probability of Property 3

In scenarios 1 and 3 the probability of exceeding the maximum number of tasks in the queue is virtually impossible, meanwhile in scenario 2 the probability increases greatly. Almost certain is the violation for the scenario.

4.2.4. Graphs of probability

Despite the graphs obtained from the verification of the properties can be several, here we present only three of them to illustrate a sample of three different scenarios each in a different property.

If we consider for example the *Scenario 1,* verifying the third property, the outcome of the probability as reported in the *Table 4. I*n this case the probability of exceeding the maximum number of tasks in the queue is very low, so the diagram generated by *"verifier tool"* is the following

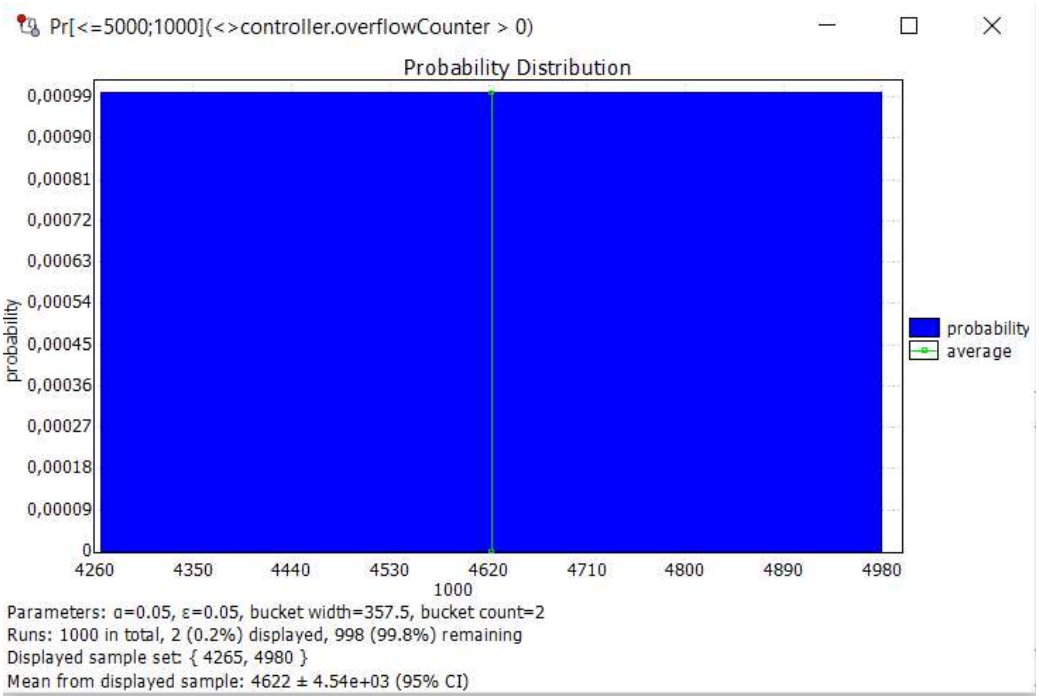

Figure 3: Probability Graph 1

If we consider the *Scenario 2,* which is in violation of the first property, as we can see from *Table 2*, in the verification of such property has a probability ≥ 0.97. This is shown in this Probability Distribution graph where the X axis indicates the time that it took for the robot to reach the state "*Returning_Pod*" and the Y axis the probability (the sum of each *column* height should be indeed 1). It is possible to see how the maximum time reached is 310 (and not the total 5000 analyzed) because in the rest of the time the values are negligible.

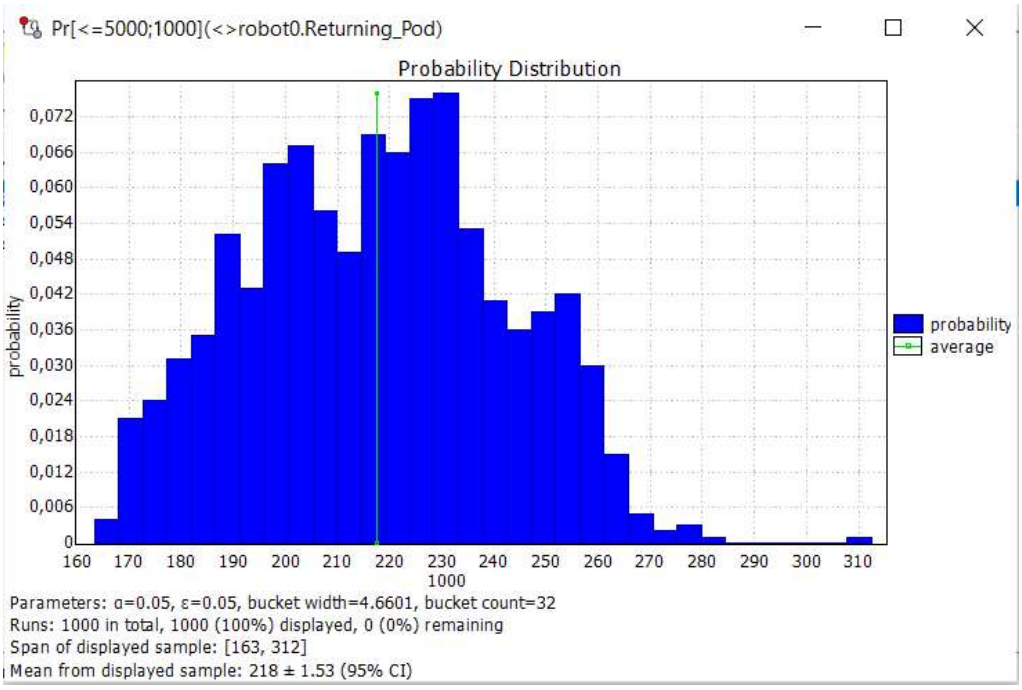

Figure 4: Probability Graph 2

To also consider a case where the probability is made by a range of value, we can take in account the *Scenario 4* in the verification of the second property which, according to what appears in *Table 3*, generate the following graph of Probability Distribution.

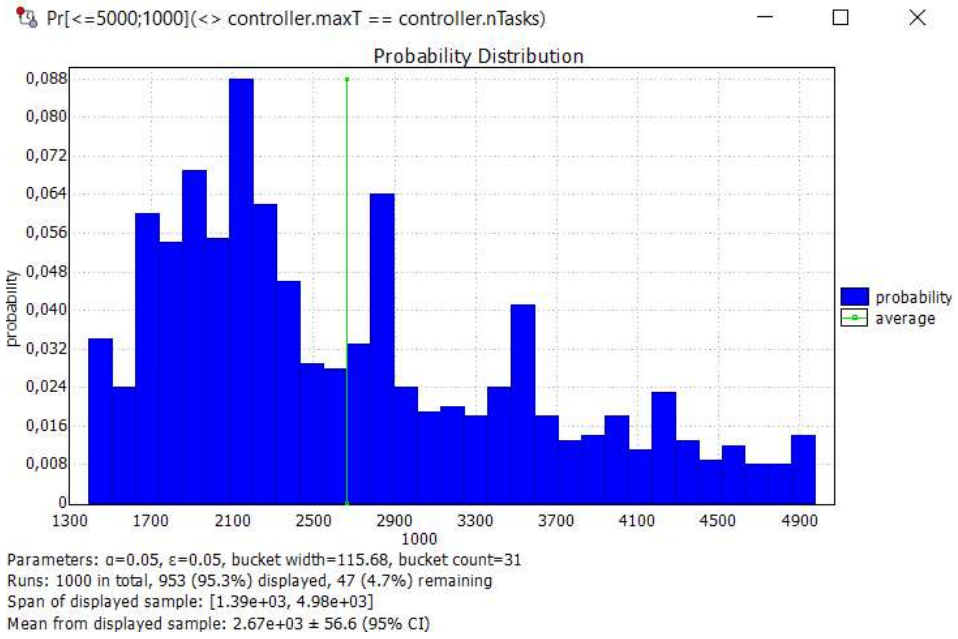

Figure 5: Probability Graph 3

4.2.5. Comment about the number of runs and tau chosen

For all three properties we have chosen a tau equal to 5000 through empirical experiments. We've spot how with values more less inferior than 5000 the "time horizon" imposed on the verifier was too close to notice the differences, and so we decided for an acceptable wider time span.

Talking about the number of "runs" chosen in the three different properties, we have chosen 1000 runs, again through empirical considerations, which resulted to be enough to show significant probability ranges for proof of property infringement.

5. Comment about testing

In order to test the different set of parameters, it is needed to uncomment the corresponding snippets of code:

- "Config N" values in "Declarations
- "ROBOT_POS[X][Y]" in "Declarations"
- POD_POS[X][Y] config in the initPodList() function (Controller -> Declarations)
- Check that the number of robot declared in the "System declaration" is correct