POLITECNICO
MILANO 1863

Formal Methods For Concurrent and Real-Time Systems

Model Checking of Warehouse Robotics with UPPAAL

Homework Project

Authors
Omitted for Privacy.

Instructors
Prof. Pierluigi San Pietro
Dr. Livia Lestingi

A.Y. 2021-2022

Abstract

An UPPAAL model for an automated warehouse is presented in this report. This document shows how the model has been built and a series of interesting properties about it. The main focus of the model is to create a realistic simulation of a warehouse towards different layouts of the grid, arrangements of pods to be transported and the presence of robots which are responsible for dispatching goods to their destination on the shop floor.

1 High Level Model Description

The model is based on 4 main components:

- *Robot*
- *Semaphore*
- *TaskManager*
- *Human*

Robot

The job of a robot is to take pods to the delivery point and then back to their initial position. In this model, the robot interacts with other components in order to find a path to its destination by running a pathfinding algorithm.

Semaphore

The semaphore is in charge of deciding which is the next robot that is allowed to move. It is needed to synchronize the robots to prevent them from moving in the same cell of the grid simultaneously.

TaskManager

The TaskManager receives new tasks to be added to the queue and assigns them to the corresponding pods. Moreover, it releases tasks to the robots in the claiming phase.

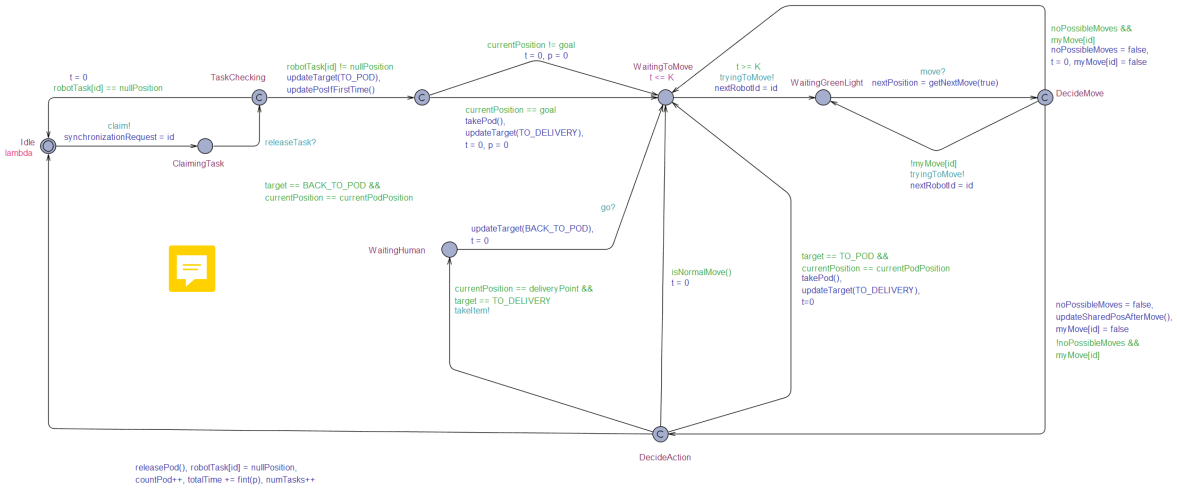
Human

The human is responsible for taking the item from the pod when the robot is at the delivery point and notify the robot when the pod can be returned to its initial position.

2 Component Description

2.1 Robot

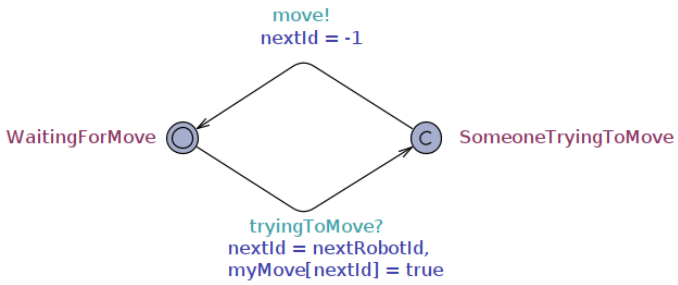
The `Robot` template contains the core of the logic of our model. Its instances represent the robots that are moving in the warehouse and that complete the tasks.



The robot lifecycle has been modeled as follows:

1. When a robot is free (i.e. does not have tasks to complete) after an amout of time sampled from an exponential distribution it tries to claim a task by interacting with the `TaskManager` using the `claim` channel
2. When a task is assigned to the robot, the corresponding pod is assigned in the global `robotTask` array to its robot and the permission to move is asked by the robot to the `Semaphore` with the `tryingToMove` channel
3. If the permission to move is granted (synchronization is guaranteed by the `move` channel), the next move to be performed is computed with the pathfinding algorithm, which will be illustrated in one of the following sections
4. At this point the robot is in the `DecideMove` state, that includes two possibilities:
   - If there are no possible moves for the robot, it returns to the `WaitingToMove` state. The robot will then be allowed to try to move again after K time units
   - If the pathfinding was successful, the robot will be able to actually move to the computed cell and the global `robotPositions` array is changed accordingly
5. After the actual move, the robot is in the `DecideAction` state, and there are four possible scenarios that can happen:
   - The robot was returning the pod back to its initial position: this means that the task has been completed and the robot can go back to the initial `Idle` state where it will try to claim other tasks. After releasing the pod `taken` is set as false
   - The robot is going to pick up the pod before bringing it to the `Human` and the current position is equal to that of the pod itself, therefore the robot will take the pod and will change its target to the delivery point. The pod now set `taken` as true
   - The robot has reached the delivery point to give the pod to the `Human`: in this case the robot has to communicate with the human in order to make him/her pick up the pod. There is a waiting time in the `WaitingHuman` state; after that, the robot will be able to update its target to return the pod to the correct initial position
   - If the situation is not any of the above cases, it means that it is a so-called *normal move*: the robot is going to the pod or to the delivery point but does not have to perform any specific action, it only has to move to the next cell in the path to the target

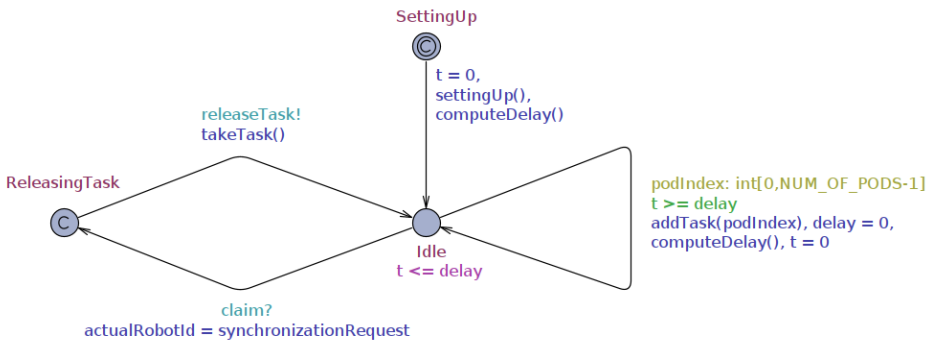
2.2 Semaphore



The `Semaphore` template is a very simple finite state machine that has the only task of synchronizing the moving phase of the various robots in the warehouse and make it atomic, avoiding the possibility that two robots try to simultaneously move to the same position. When a robot is about to move, a message through the `tryingToMove` channel is sent. The `Semaphore` will set as true the element in the `myMove` global array that corresponds to the synchronized robot: this will give the signal that the movement can be safely performed.

2.3 TaskManager

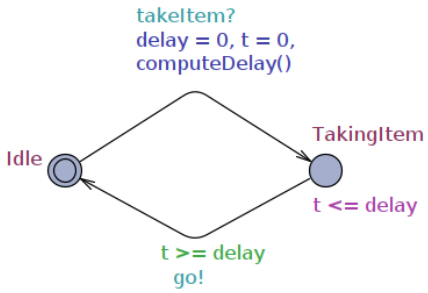
As previously mentioned, the `TaskManager` is the entity which collects new tasks when they are added to the queue, and allocates them to the robots.



The `TaskManager` works as follows:

1. In the initial state (and then every time a new task is added) a random delay for adding new tasks is computed
2. When a robot claims a task, if the queue is not empty, the task is assigned and removed from the queue

2.4 Human



The job of the `Human` is to take the item that is placed on the pod. This operations will take a certain amount of time, that is randomly computed every time a robot delivers a pod to the `Human`. Once the delay period has passed, a message on the `go` channel is sent to the robot to notify it that the pod can be returned to its initial position.

2.5 Channels

- `claim` is the channel used by the robots to get a new task from the TaskManager (if available)
- `releaseTask` is used by the TaskManager to notify robots that a task has been assigned (if available)
- `tryingToMove` is the first of the two channels exploited by the robots to communicate with the semaphore and avoid collisions with other robots
- `move` is the actual channel that is listened by the robots and that triggers the actual movement of the robots themselves, that can be now safely performed
- `takeItem` is used by robots to make the human pick up the pod

- `go` is the channel that sends a signal to the robot to return the pod to its initial position, since the human has successfully taken the item

2.6 Design Choices

Floor layout

The floor (grid) layout has not been implemented as a dedicated template. As a matter of fact, the only important things to consider regarding the grid are the initial positions of the pods and the current position of the all the robots. Therefore, our grid representation is "implicit": given the position of pods, robots and the dimension of the grid, the layout is known without allocating any other variable. This simplification allows us to avoid to allocate a matrix to represent the grid as a global variable, thus saving time when checking properties. Moreover, the optional feature that states that the model should support any rectangular-like grid layout has been implemented: it is enough to change the number of rows/columns, the initial positions of the pods and the entry/delivery point in the global declarations, and the system will "adapt" to the new floor layout.

Decentralized control for robots

Each `Robot` computes its own path to get to the current target without a centralized control unit. This choice is due mainly to scalability reasons: in a real-world scenario, the process of adding new robots to the warehouse can be easier if each robot can "decide" itself the path to follow, and not all the responsibility is on a central unit that could be saturated and/or encounter synchronization issues.

Pathfinding

Given that we decided to design our system as to accept any grid-like layout, we couldn't make any assumption on how to generate the path from a robot to its goal. To avoid any "soft deadlocks" (i.e. when a robot is stuck or unable to reach its goal indefinitely) we needed to generate the complete path by exploring the entire grid. We decided to use a greedy DFS (Depth-First Search) algorithm. We have wrote the pseudo-code for the pathfinding in algorithm 1. We tried to calculate the path and save it inside an array, but that slowed down significantly the checking due to the significant number of variables added. Furthermore, when we tried to declare the array containing the path as **meta** all robots started to skip multiple positions in their path, resulting in an incorrect system. So, the best thing was to let every robot recalculate the path before each movement, then forget everything except the immediate next move. Unfortunately this resulted in a long checking time, but it was far more manageable than any other option that we tried before.

```
push(currentRobotPosition);
while (searching){
    if(!stackIsEmpty()){
        curPos = lastElementInStack();
        if(isEqual(curPos, goal)) {
            copyStack();
            return true;
        } else {
            directions = getGreedyBestDirections(curPos);
            for(direction : directions){
                if(!visited(direction)){
                    push(direction);
                    added = true;
                    break;
                }
            }
        if(!added) { pop(curPos); }
        added = false;
        }
    } else { searching = false; }
}
return false;
```

Listing 1: Pathfinding pseudo-code

Statistical Model Checking

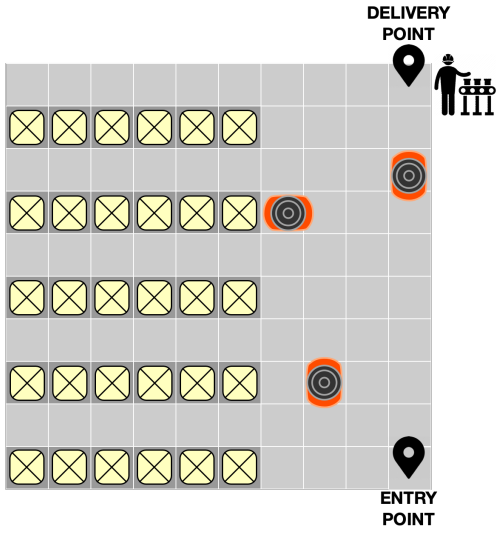
In this project, all the (optional) stochastic features have been implemented. This means that:

- The time elapsed between the generation of two tasks is a sample of a Normal distribution
- The human reacts to the arrival of the robot at the delivery point in a time interval that is described by a Normal distribution (not predictable)
- The robots claim new tasks with a delay described by an Exponential distribution

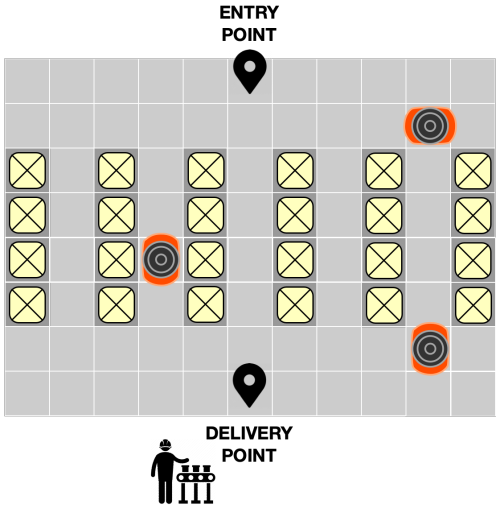
Regarding the delays that must follow a Normal distribution, a dedicated function `Normal()` has been added to the model. In this case, the result has been converted to the integer data type by using the `fint()` function, in order to have shorter property checking times. This is acceptable, since having delays represented by floating point data types doesn't add anything significant in order to obtain a realistic warehouse simulation, that can be well described using integer delays.

3 Grid Layouts

Our system can accept any rectangular grid layout that the user can think of. We decided to design it in this way because constraining it to accept only some specific layouts is extremely limiting for a real system. We decided to use two layouts to test our model. The first one (1a) is the same that professor Lestingi proposed in the presentation of the homework. We felt that its design was challenging enough for our pathfinding and at the same time was close enough to a real case scenario. The second layout (1b) instead was created to explore the limit of our system in an easier setting for all the robots, but still providing a plausible scenario.



(a) Config. 1



(b) Config. 2

The first layout (1a) is contained in the .xml file named **homework**, while the second (1b) in **homework2**. In both of the files multiple properties are described and are ready to be checked. In each file 2 test configuration are available: HIGH and LOW. The HIGH configuration will return an high probability for the fundamental porperty that has to be checked, while the LOW configuration will return a low probability.

4 Properties

In this section we are going to show the proprieties that we have checked over the system to be sure that it behaves as expected. The properties we discuss here can be found as *queries* in the *Verifier* section of UPPAAL. The properties shown here include the mandatory one, in addition to some properties that aim at measuring the performance of the system.

As stated in previous sections, the proposed model includes random delays, therefore *stochastic model checking* (SMC) has been applied. Regarding the statistical parameters, we decided to set the value of ϵ to 0.025 in order to find a trade-off between the *uncertainty* of the results, and the time needed to check the properties. The other statistical parameters were maintained with the values suggested by UPPAAL.

It is important to highlight that the complexity of the system has made impossible to set an high *time horizon* (i.e. the number of "passes" considered when checking properties). Therefore, we decided to set our time horizon to 10^4 steps, which is a trade-off, but still significant enough to evaluate the properties in a consistent way.
In the following table the parameters used for every configuration are shown.

Configuration	Robot speed	Human μ	Human σ	TaskManager μ	TaskManager σ
Homework_LOW	1	11	1.5	64	2.5
Homework_HIGH	2	18	1.5	42	2.5
Homework2_LOW	2	13	1.5	38	2.5
Homework2_HIGH	4	20	1.5	35	2.5

4.1 Robots perform a task on an average time between min and max

Our first property checks how long it takes a robot to complete a task. In particular, we checked that every robot perform a task in a bounded average time and we searched the boundaries for the two configurations of each layout of the grid. These are the two properties which have been checked:

$$(\boldsymbol{\psi}): \quad \forall\Box\ (\ (r0.totalTime/r0.countPod) < max\ \wedge\ (r1.totalTime/r1.countPod) < max \wedge (r2.totalTime/r2.countPod) < max\ )$$

$$(\boldsymbol{\lambda}): \quad \forall\Diamond\ (\ min < (r0.totalTime/r0.countPod)\ \wedge\ min < (r1.totalTime/r1.countPod) \wedge min < (r2.totalTime/r2.countPod)\ )$$

These are the best accurate results we found considering a reasonable amount of time to check the properties (at most 10 minutes, considering $\epsilon = 0.075$ for the HOMEWORK layout).

Configuration	max	min	probability max	probability min	moves
Homework_LOW	95	35	$0.83 \leq \mathbb{P}(\Box_{\leq 10^4}\psi) \leq 0.98$	$\mathbb{P}(\Diamond_{\leq 10^4}\lambda) \geq 0.85$	$35 \leq n \leq 95$
Homework_HIGH	130	35	$0.83 \leq \mathbb{P}(\Box_{\leq 10^4}\psi) \leq 0.98$	$\mathbb{P}(\Diamond_{\leq 10^4}\lambda) \geq 0.84$	$17 \leq n \leq 65$
Homework2_LOW	70	45	$0.91 \leq \mathbb{P}(\Box_{\leq 10^4}\psi) \leq 0.96$	$\mathbb{P}(\Diamond_{\leq 10^4}\lambda) \geq 0.95$	$22 \leq n \leq 35$
Homework2_HIGH	145	90	$0.89 \leq \mathbb{P}(\Box_{\leq 10^4}\psi) \leq 0.94$	$\mathbb{P}(\Diamond_{\leq 10^4}\lambda) \geq 0.95$	$22 \leq n \leq 35$

The results of the layout HOMEWORK are not so precise, as we had to increase the uncertainty parameter. Nevertheless, we are able to extract some conclusions after normalizing the number of moves performed by every robot using their speed. As we expected, given the more xomplex layout, the HOMEWORK configuration is significantly slower than HOMEWORK2. In particular for the LOW setup HOMEWORK goes 2 times slower than HOMEWORK2, while for the HIGH setup only 1.33 times. As we anticipated, the floor layout is the main factor affecting the performance of our system.

4.2 Maximum difference of tasks executed between robots

The second property checks the maximum difference of tasks executed between the robots. This property verifies not only the fact that our model is consistent but also measures the efficiency of the entire system in terms of distribution of the tasks to the robots. As a matter of fact, it's important that the system proves to be able to distribute the tasks among robots as equally as possible, in order to maximize efficiency.

$$(\boldsymbol{\xi}): \quad \forall \Diamond \ (\ abs(r0.numTasks - r1.numTasks) > taskDifference \vee abs(r0.numTasks - r2.numTasks) > taskDifference \vee abs(r1.numTasks - r2.numTasks) > taskDifference) \ )$$

configuration	taskDifference	probability
Homework_LOW	28	$\mathbb{P}(\Diamond_{\leq 10^4} \xi) \leq 0.05$
Homework_HIGH	15	$\mathbb{P}(\Diamond_{\leq 10^4} \xi) \leq 0.05$
Homework2_LOW	23	$\mathbb{P}(\Diamond_{\leq 10^4} \xi) \leq 0.05$
Homework2_HIGH	15	$\mathbb{P}(\Diamond_{\leq 10^4} \xi) \leq 0.05$

The number of passes considered is, as always, 10^4, so we can calculate the total number of tasks to review the distribution of tasks.

configuration	Total generated tasks
Homework_LOW	~ 156
Homework_HIGH	~ 238
Homework2_LOW	~ 263
Homework2_HIGH	~ 285

The maximum difference in the number of tasks completed by the robots is 28 which correspond in the worst case to 18% of the total task generated for Homework.LOW and the minimum is 15 which correspond in the best case to 5% of total tasks generated in Homework2.HIGH.
In conclusion, we can say that the distribution of tasks is sufficiently fair.

4.3 Mandatory Property

The mandatory property for the project aims to verify that it never happens that tasks are lost, i.e. the task queue is never full. Obviously, as SMC features were included in the model, what we can verify is the probability of losing incoming tasks.

The property has been checked by exploiting the TaskManager, since it handles the task queue and distribution of the tasks to the robots. In particular, when the queue is full and another task is generated, the TaskManager sets his `exploded` variable to true. The mandatory property is expressed as follows:

$$(\boldsymbol{\gamma}): \quad \forall \Diamond \ taskManager.exploded == true$$

In this case, we wanted to show two configurations: one in which the probability of losing tasks is high, and one in which it is unlikely for this event to happen.

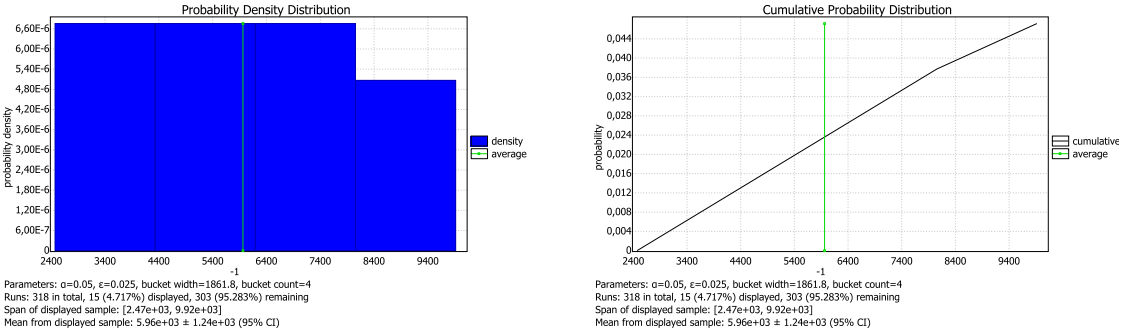
configuration	probability
Homework_LOW	$0.025 \leq \mathbb{P}(\Diamond_{\leq 10^4} \gamma) \leq 0.075$
Homework_HIGH	$0.935 \leq \mathbb{P}(\Diamond_{\leq 10^4} \gamma) \leq 0.985$
Homework2_LOW	$0.02 \leq \mathbb{P}(\Diamond_{\leq 10^4} \gamma) \leq 0.07$
Homework2_HIGH	$0.93 \leq \mathbb{P}(\Diamond_{\leq 10^4} \gamma) \leq 0.98$

All the probabilities are very polarized ($\geq 95\%$ or $\leq 5\%$), so we can see immediately if a configuration works well or not. We have created two configurations for each layout: the configuration LOW, described in the previous sections, is more *robust*, therefore it can work in most situations without any problem, while the configuration HIGH has been built to show a non-efficient behaviour, as it accumulates tasks in the queue of the TaskManager.

For this mandatory property we retrieved some probabilistic diagrams, which describe the trend of the system.

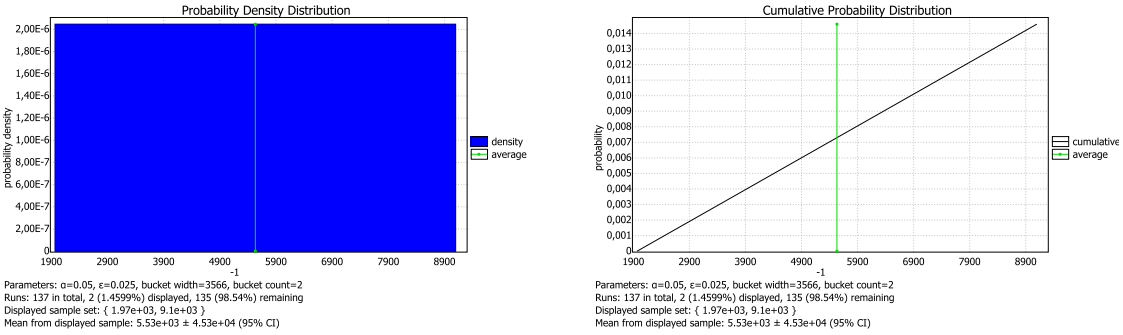
4.3.1 Homework_LOW and Homework2_LOW

For these two configurations, we can verify that the efficiency is very high because the graph cannot describe well when the queue is full due to the fact the queue is filled extremely rarely.



(a) PDD (b) CPD

Figure 1: Probability distributions of the homework with the configuration LOW

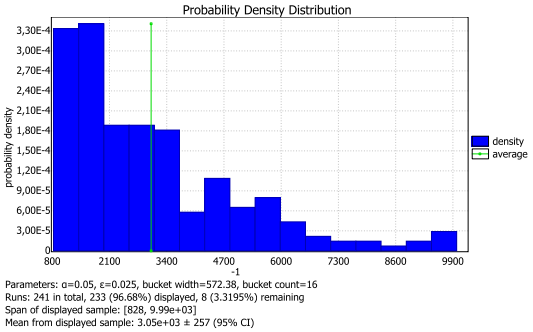


(a) PDD (b) CPD

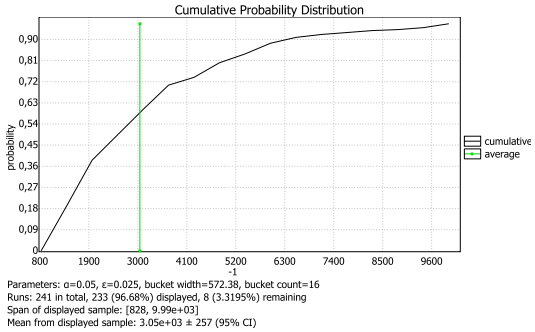
Figure 2: Probability distributions of the homework2 with the configuration LOW

4.3.2 Homework_HIGH and Homework2_HIGH

In the configurations HIGH, we can find that the graphs can describe pretty well what happened. In particular, its say that it's more probable that the homework system is "broken" in the first 4000 steps and the function is likely a logarithm function.
For the homework2 system, its indicate that the system works well at the beginning before 3000/4000 steps but in the middle of the runs it is overworked and between 4000 and 9000 steps it's probable that the system becomes "broken". On the contrary, after 9000 steps the probability of the system to be "broken" decreasing.

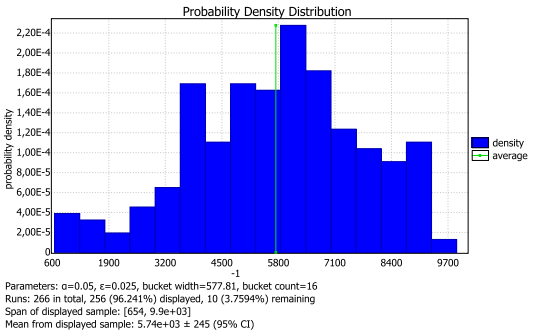


(a) PDD

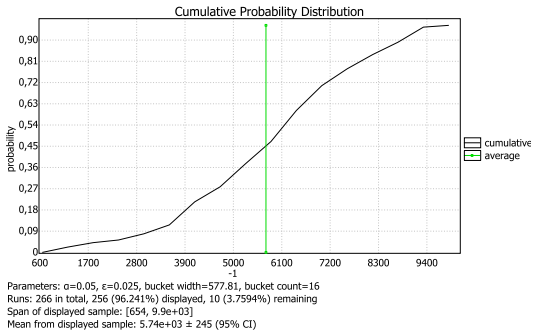


(b) CPD

Figure 3: Probability distributions of the homework with the configuration HIGH



(a) PDD



(b) CPD

Figure 4: Probability distributions of the homework2 with the configuration HIGH

5 Conclusions

To summarize, this warehouse model has been built to be as general as possible, as it can handle any grid layout. Most of the efforts has been spent on designing the pathfinding algorithm, that leads to a system that, as shown by the proven properties, has a fair task distribution. Moreover, the first configuration, given its narrow corridors, is extremely more complex to navigate for robots. Instead, the second configuration, facilitating the pathfinding, has shown better resilience to more extreme parameters.

References

[1] Alexandre David. *UPPAAL SMC Tutorial*. URL: https://www.it.uu.se/research/group/darts/papers/texts/uppaal-smc-tutorial.pdf.

Model Checking of Warehouse Robotics

Project Report
Formal Methods for Concurrent and Real-Time Systems

Omitted for privacy.

POLITECNICO
MILANO 1863

June 27th, 2021

Title:	Model Checking of Warehouse Robotics
Authors:	Omitted for privacy. Omitted for privacy.
Course:	Formal Methods for Concurrent and Real-Time Systems [088882]
Ref. professors:	Livia Lestingi Pierluigi San Pietro
Version:	1.1
Date:	2021-06-27
Source:	
Copyright:	

Table of Contents

List of Figures

1 Design

1.1 Warehouse layout

We design the warehouse as a rectangular grid of arbitrary size, divided in a west and east side, each of which holds an equal number of rows of pods, spaced one cell from each other. Each of the two sides has a (possibly different) fixed number of pods per row. Between the two sides, there is a central highway spanning the entire warehouse in height and two columns in width. At most one of the two sides of pod rows can be completely missing (i.e. having *zero* pods per row): in such case the highway is positioned completely west or completely east of the warehouse.

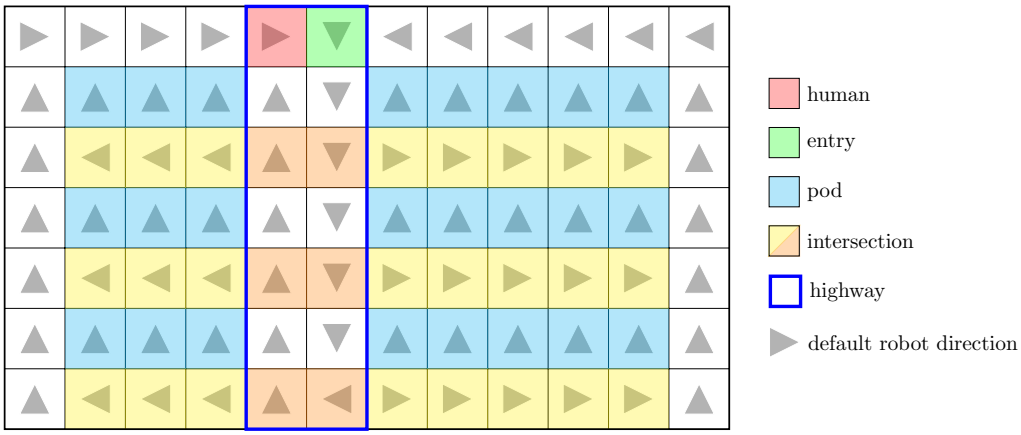


Figure 1.1: Warehouse layout

The *human operator* can be positioned on an arbitrary cell of the highway, while the *entry point* can be positioned on an arbitrary cell of the warehouse that is not already occupied by a pod or by the human operator.

The robots in the warehouse move from cell to cell according to the default directions defined for each cell, except for *intersection* cells where they can turn and move in a different direction in order to reach a pod. The robot behavior and movement is described more in detail in section 2.2.4.

The above image highlights one of the possible layouts of the warehouse as well as the default directions that bots will travel. In this case, we have 3 pod rows, 3 pods-per-row on the west side and 5 pods-per-row on the east side. These parameters (further described in section 2.1) allow defining a rectangular warehouse of arbitrary size with arbitrary west-to-east highway placement.

1.2 Stochastic features

We model the following stochastic features, where the normal distributions were implemented directly as transitions on the timed automata following the UPPAAL SMC Tutorial[1].

- Time elapsed between two tasks with a normal distribution: $\mathcal{N}(\mu_T, \sigma_T)$

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

- Delay for a robot to claim a new task with an exponential distribution: $\mathcal{X}(\lambda)$
- Human operator processing time with a normal distribution: $\mathcal{N}(\mu_H, \sigma_H)$

1.3 Design assumptions

The following assumptions were made at the design stage and need to be satisfied in order for the model to be simulated correctly and meaningfully:

1. The sum of the total number of robots plus the capacity of the task queue is strictly less than the number of pods in the warehouse. It does not make sense to model a system in which the number of pods is lower than or equal to the number of available bots plus the queue capacity: such a system would never be able to fail under any time parameter assignment.
2. Pods with idle bots underneath cannot be assigned to any robot as a new task. This implies that no robot can be assigned a task corresponding to the same pod twice in a row.
3. Entry point and human operator are placed on valid cells, that is: human on one of the cells of the highway and entry on any other cell that is not a pod.

2 UPPAAL Model

2.1 Global Declarations

2.1.1 System parameters

Parameter	Description
`N_BOTS`	Number of robots in the warehouse
`N_POD_ROWS`	Number of rows of pods in the warehouse
`N_PODS_PER_ROW_W`	Number of pods per single row on the warehouse West side
`N_PODS_PER_ROW_E`	Number of pods per single row on the warehouse East side
`QUEUE_CAPACITY`	Capacity of the tasks' queue
`TASK_GEN_MEAN`	Task generation mean time (μ_T)
`TASK_GEN_VAR`	Task generation time variance (σ_T)
`HUMAN_MEAN`	Human operator mean reaction time (μ_H)
`HUMAN_VAR`	Human operator reaction time variance (σ_H)
`BOT_IDLE_EXP_RATE`	Robot task claim delay (λ of exponential distribution)
`BOT_STEP_TIME`	Time needed for a robot to move between adjacent tiles
`ENTRY_POS`	Entry point position (coordinates)
`HUMAN_POS`	Human operator position (coordinates)
`TAU`	Upper time bound for verification

2.1.2 Variables and channels

We model the warehouse as a 2D matrix `int map[][]`, whose size is calculated based on the system parameters `N_POD_ROWS` and `N_PODS_PER_ROW_{W,E}`: each cell holds information about its kind (pod/human/entry/intersection), whether a robot is present or not, and the default directions for bots to follow. The task queue is a simple array `int tasks[]` used as a ring buffer. Finally, we keep track of the availability of each pod through another global array `bool pod_free[]`.

For synchronization purposes we define the following channels, which are *all* `urgent broadcast`:

- `init_done`: only used for initialization to signal all templates to start
- `asap`: used to fire a particular transition as soon as possible
- `step`: channel used to synchronize the movement of the bots when one or more bots are stuck one behind each other in a queue
- `delivery_ready`: used by the `Bot` template to notify the human that it's ready for delivery
- `human_done`: used by the `Human` template to release the currently delivering

2.2 Templates

2.2.1 Initializer

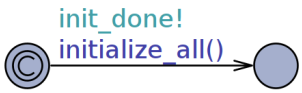


Figure 2.1: Initializer Timed Automaton

This is a dummy timed automaton which only has one transition which executes immediately as the initial state is committed. The `initialize_all()` function generates the map and initializes global variables. All the other timed automaton of the system are signaled to start through the `init_done` channel.

2.2.2 TaskGenerator

This template models the incoming requests for the packages to the warehouse. It takes a mean and variance as parameter to describe the normal distribution for the task generation time: $\mathcal{N}(\mu_T, \sigma_T)$.

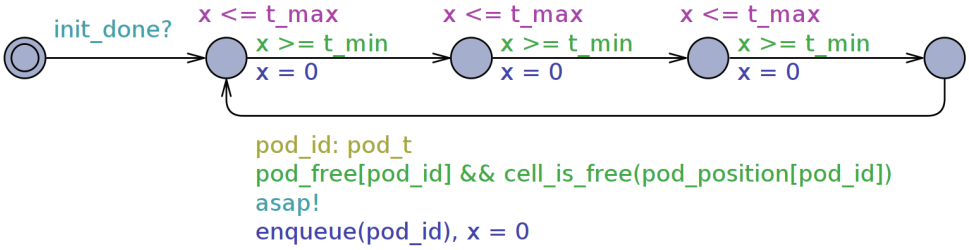


Figure 2.2: TaskGenerator Timed Automaton

It is composed of a single cycle of 4 edges, 3 of which model the normal distribution. The constants `t_min` and `t_max` are defined as:

$$\texttt{t\_min} = \frac{\mu_T - \sigma_T}{3} \qquad \texttt{t\_max} = \frac{\mu_T + \sigma_T}{3}$$

The last edge that closes the cycle performs the actual generation of a task by selecting a free pod that has no robot below it and enqueueing the task into the global `queue` through the `enqueue()` helper function. Task generation is done as soon as possible after the initial delay exploiting the `asap` urgent broadcast channel.

2.2.3 Human

This template models the behaviour of the human operator. It takes a mean and variance as parameter to describe the normal distribution for the operator processing time: $\mathcal{N}(\mu_H, \sigma_H)$.

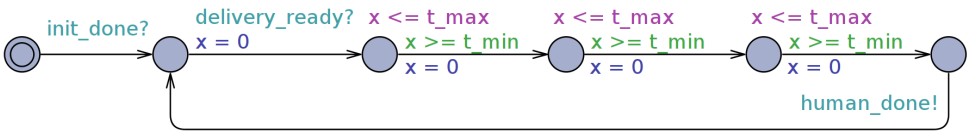


Figure 2.3: Human Timed Automaton

It is composed of a single cycle of states. The first transition waits for a robot to show itself through the `delivery_ready` channel. Again, the following 3 edges model the normal distribution in the same way as for `TaskGenerator`. The last edge that closes the cycle releases the delivering robot through the `human_done` channel.

2.2.4 Bot

This template models a single robot and is instantiated `N_BOTS` times. The only template parameter is the robot's ID.

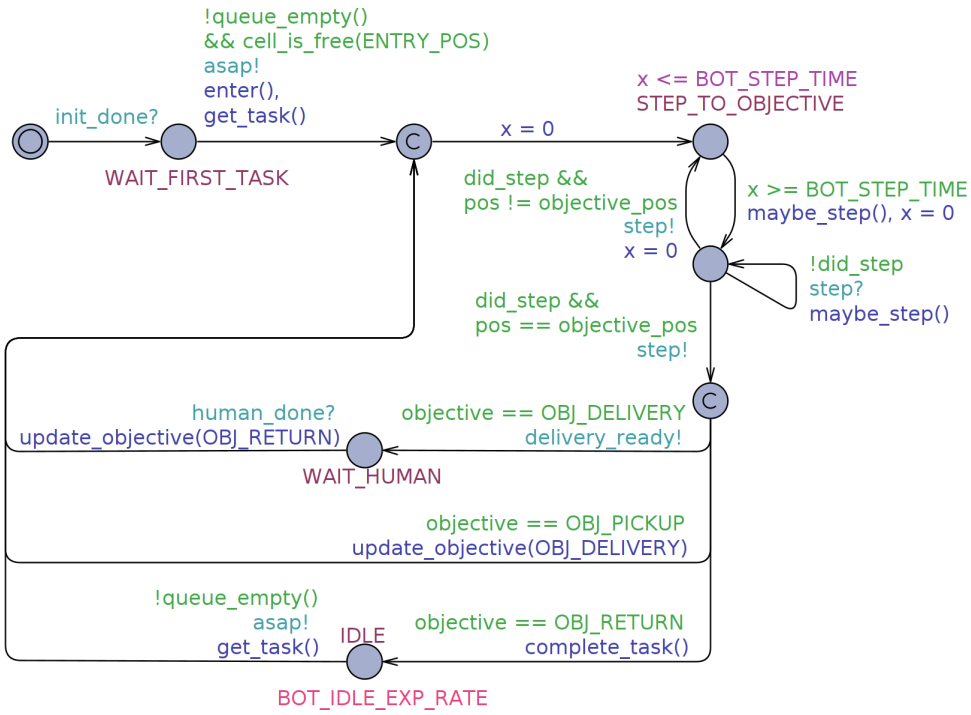


Figure 2.4: Bot Timed Automaton

At a high level, after entering the warehouse, each robot continuously cycles among four *objectives*[2]: ***pickup***, ***delivery***, ***return***, ***idle***. When not ***idle***, the behavior of a robot is based on its current *objective*

[2]We merely chose the word *objective* here because in UPPAAL `state` is a reserved word

and *objective position* (coordinates in the map). The initial objective of a robot right after entering the warehouse is ***pickup***. The behavior of a robot depending on the objective is as follows:

- ***Pickup*** or ***return***: the *objective position* corresponds to the assigned pod. To reach it, the robot follows the default directions for all cells *except* intersections (see layout in figure 1.1), in which case it checks based on its position whether it needs to enter the aisle below the wanted pod's row *or* go up because it just reached the cell right under the pod *or* follow the default direction. After ***pickup*** the robot changes objective to ***delivery***, while after ***return*** the robot becomes ***idle***.
- ***Delivery***: the *objective position* corresponds to the human. The robot merely follows the default direction for each cell of the warehouse (see layout in figure 1.1). These directions ensure that robots always reach the highway regardless of their starting position, and then cycle on the highway until the human is reached. After reaching the human, the robot signals that the pod has been delivered through the `delivery_ready` channel and waits on the `human_done` channel. After this, the objective changes to ***return***.
- ***Idle***: the robot stays under the returned pod for a minimum amount of time determined by an exponential probability distribution (with $\lambda =$ `BOT_IDLE_EXP_TIME`), then retrieves a new task from the queue as soon as it is available.

The actual movement logic, in terms of step-by-step movements through the cells of the grid of the warehouse, is implemented in the `maybe_step()` function, which tries to take a single step based on the robot's current position, *objective* and *objective position*. If the movement is successful, the robot advances one cell, otherwise this means that another robot is currently occupying the target cell: in this case the robot implicitly waits for any other robot movement through the `step` channel until the target cell becomes free.

2.3 Conflicts between robots

It may happen that more than one robot needs to move to the same cell at the same time. Whenever a robot needs to move, it first checks if the target cell is free, and if so it moves to it and marks it as occupied. This conflict is implicitly solved by the atomicity of transitions provided by UPPAAL. Since timed automata transitions are executed atomically (even if happening at the same instant of time), there will always be only one robot moving to the target cell, and the conflict is avoided altogether.

It may also happen that one or more robots get stuck behind each other in line. In such case, the `step` urgent broadcast channel makes all robots in the line move forward as soon as possible, without wasting another `BOT_STEP_TIME` period per robot.

Finally, all ***idle*** robots try (at the same time) to pick the next available task from the queue if it is not empty, which raises another possible conflict. The atomicity of this operation is again guaranteed by the atomicity of transitions provided by UPPAAL, so there cannot be multiple bots picking the same task off the queue.

3 Analysis and results

We first performed a multi-parameter analysis to find meaningful system configurations, then extracted two non-trivial configurations to analyze the system under the two requested scenarios: one with high efficiency and one with low efficiency.

3.1 Multi-parameter analysis

We analyzed the warehouse efficiency in terms of the probability of losing any task (i.e. exceeding the task queue capacity) through the following query:

```
Pr [<= TAU] (<> tasks_lost > 0)
```

We built an ad-hoc Python test suite[3] to vary 3 independent parameters in different ranges, verifying each different configuration using the `verifyta`[4] command line tool bundled with UPPAAL 4.1.25[5], plotting the results using Matplotlib[6]. We created two multi-parameter tests, configured according to the following table, and ran them with default SMC parameters except for *probability uncertainty* set to $\varepsilon = 0.01$.

System parameter	Test A value	Test B value
N_BOTS	5	$\in [3, 10] \subset \mathbb{N}$
N_POD_ROWS	5	5
N_PODS_PER_ROW_W	$\in [0, 10] \subset \mathbb{N}$	5
N_PODS_PER_ROW_E	$10 -$ N_PODS_PER_ROW_W	5
QUEUE_CAPACITY	$\in \{1, 5, 10, 15, 20\}$	$\in [1, 10] \subset \mathbb{N}$
TASK_GEN_MEAN (μ_T)	$\in [10, 20] \subset \mathbb{N}$	$\in [5, 20] \subset \mathbb{N}$
TASK_GEN_VAR (σ_T)	5	1
HUMAN_MEAN (μ_H)	2	2
HUMAN_VAR (σ_H)	1	1
BOT_IDLE_EXP_RATE (λ)	3	3
BOT_STEP_TIME	1	1
ENTRY_POS	top of highway east column	top of highway east column
ENTRY_POS	bottom of highway east column	bottom of highway east column
TAU	10000	10000

The main purpose of **Test A** is to assess the effect of different highway placements: we keep the number of pods and pod rows fixed and vary the position of the highway from East to West of the warehouse.

The main purpose of **Test B** is to assess the impact of the queue capacity under different task generation frequencies and with different workloads (task generation mean time) and different numbers of robots.

[3] See README.md at
[4] https://docs.uppaal.org/toolsandapi/verifyta
[5] https://uppaal.org/downloads/other/#uppaal-41
[6] https://matplotlib.org

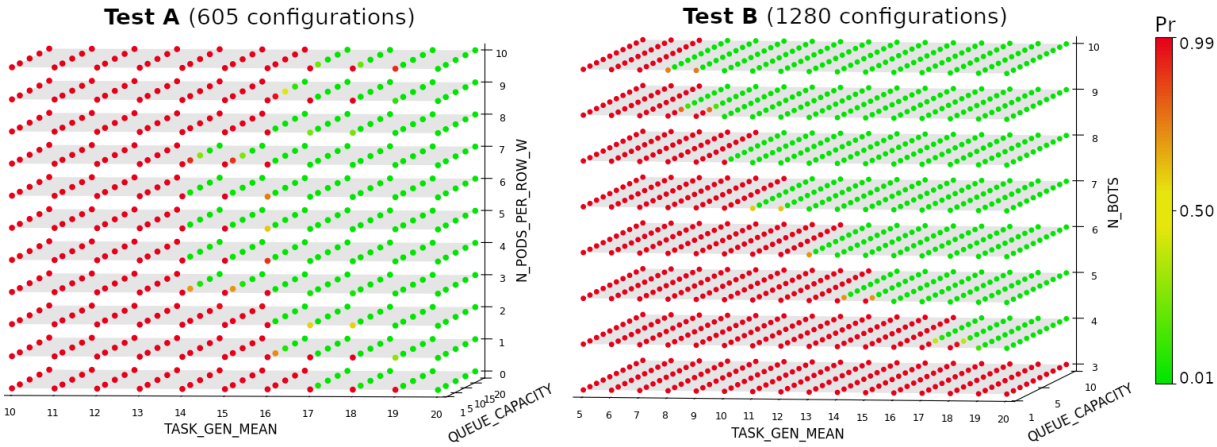


Figure 3.1: 4D plots of probability of losing any task - one data point per configuration

From the results of **Test A**, we conclude that (unsurprisingly) a centered highway placement is better than a lateral one. From the results of **Test B**, we observe that *the capacity of the task queue only marginally affects the efficiency of the warehouse*. The probability of not losing tasks does not scale linearly along with the queue capacity. Fixed other system parameters, there seems to be a clear breakpoint for the queue capacity before which losing any task is very likely and after which is very unlikely. Choosing a capacity that is above this breakpoint is thus unneeded.

From the above graph for **Test B**, we can in fact see that for a warehouse of 50 pods with a centered highway, if the bots are capable of completing tasks in a timely manner, a queue capacity of 4 is enough to not lose tasks even in the most stressful system configurations, otherwise *higher queue capacities* do not solve the problem; the queue will eventually fill up and tasks will start getting lost. However, it is worth noting that what we are testing here is continuous operation of the warehouse. In a real-life scenario where the warehouse may *not* be continuously operational 24 hours a day, a large enough queue (based on the maximum time of continuous operation per day) could actually solve the problem.

3.2 First scenario: efficient warehouse

From the analysis performed in the previous section, we extract a relevant system configuration where the warehouse rarely loses tasks. Here we have a warehouse with 50 total pods distributed evenly between West and East and 10 robots.

System and SMC parameters are configured as follows (non-default SMC values in bold):

System parameter	Value
N_BOTS	10
N_POD_ROWS	5
N_PODS_PER_ROW_W	5
N_PODS_PER_ROW_E	5
QUEUE_CAPACITY	5
TASK_GEN_MEAN (μ_T)	8
TASK_GEN_VAR (σ_T)	5
HUMAN_MEAN (μ_H)	2
HUMAN_VAR (σ_H)	1
BOT_IDLE_EXP_RATE (λ)	3
BOT_STEP_TIME	1
ENTRY_POS	$\{0, 7\}$ (highway north-east)
HUMAN_POS	$\{10, 7\}$ (highway south-east)
TAU	10000

SMC parameter	Value
$\pm\delta$	0.01
α	**0.01**
β	**0.01**
ε	**0.01**
u_0	0.9
u_1	1.1
Trace resolution	4096
Discretization step	0.01

We calculated the efficiency of the warehouse (as probability of losing any task) again through the following query to get an exact result:

```
Pr [<= TAU] (<> tasks_lost > 0)
```

Which yields $\Pr \in [0, 0.0199955]$ with a confidence of 99% (over 228 runs).

We finally calculated the expected maximum number of tasks waiting in the queue at any given time through the following query, where `tasks_in_queue` is a global counter updated when enqueueing/dequeueing:

```
E [<= TAU; 1000] (max: tasks_in_queue)
```

Which yields the following probability distribution:

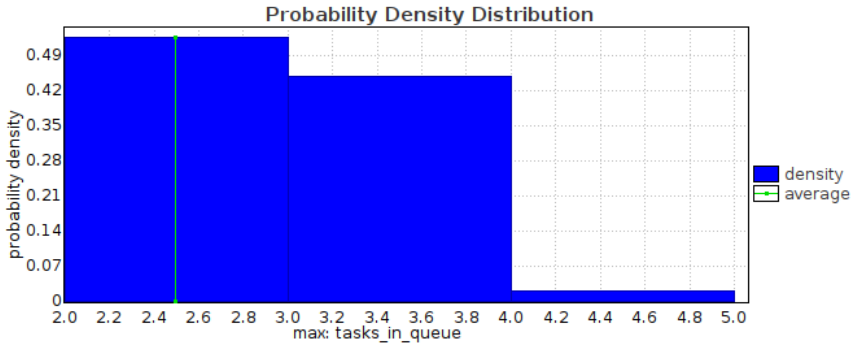


Figure 3.2: Scenario 1: probability distribution of maximum tasks in queue

The average maximum number of tasks in the queue over 1000 runs is 2.502 ± 0.045. The queue rarely gets filled up to 4 tasks, and it does not seem to be ever holding 5 tasks.

3.3 Second scenario: inefficient warehouse

Always from the analysis performed in the previous section, we extract a second relevant data point where the warehouse almost certainly loses tasks.

Here's the full list of system and SMC parameters. There are only 3 system parameters which are different from the previous scenario, with the new values highlighted in bold:

System parameter	Value
`N_BOTS`	$10 \rightarrow \mathbf{7}$
`N_POD_ROWS`	5
`N_PODS_PER_ROW_W`	5
`N_PODS_PER_ROW_E`	5
`QUEUE_CAPACITY`	$5 \rightarrow \mathbf{10}$
`TASK_GEN_MEAN` (μ_T)	$8 \rightarrow \mathbf{10}$
`TASK_GEN_VAR` (σ_T)	5
`HUMAN_MEAN` (μ_H)	2
`HUMAN_VAR` (σ_H)	1
`BOT_IDLE_EXP_RATE` (λ)	3
`BOT_STEP_TIME`	1
`ENTRY_POS`	$\{0, 7\}$ (highway north-east)
`HUMAN_POS`	$\{10, 7\}$ (highway south-east)
`TAU`	10000

SMC parameter	Value
$\pm\delta$	0.01
α	0.01
β	0.01
ε	0.01
u_0	0.9
u_1	1.1
Trace resolution	4096
Discretization step	0.01

As for the previous scenario, we calculated the warehouse efficiency again with the same query:

```
Pr [<= TAU] (<> tasks_lost > 0)
```

Which now yields $\Pr \in [0.980005, 1]$ with a confidence of 99% (over 228 runs).

In this case UPPAAL also produces probability distribution graphs, so we exported the cumulative probability distribution of exceeding the queue capacity and starting to lose tasks over time:

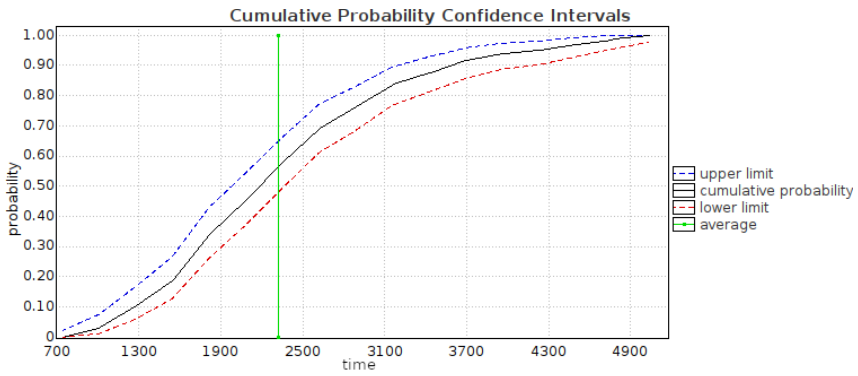


Figure 3.3: Scenario 2: cumulative probability distribution of losing any task over time

Given enough time, roughly half of our simulation upper bound (`TAU`), the system becomes almost certainly ($\Pr \approx 1$) unable to handle the workload and starts losing tasks.

Finally, taking a look at the probability distribution of the total number of completed tasks versus the total number of lost tasks using the following two queries:

```
E [<= TAU; 1000] (max: tasks_completed)
E [<= TAU; 1000] (max: tasks_lost)
```

We obtain the following results:

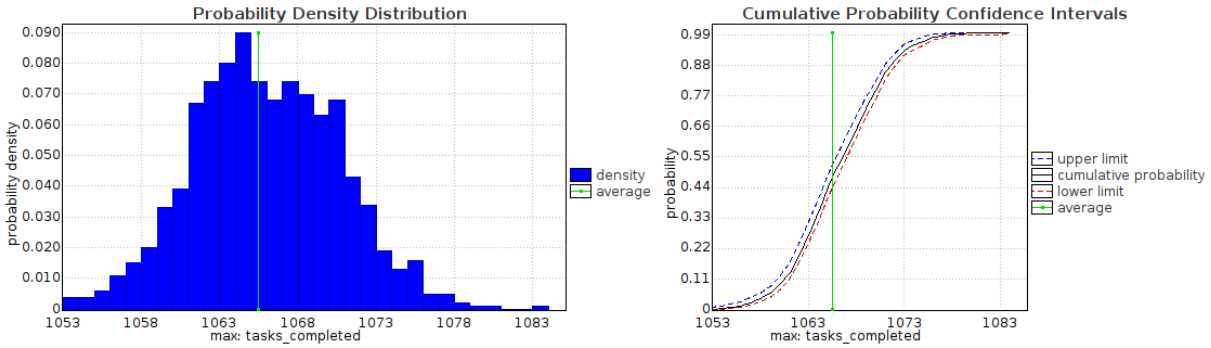


Figure 3.4: Scenario 2: probability distribution of maximum number of completed tasks

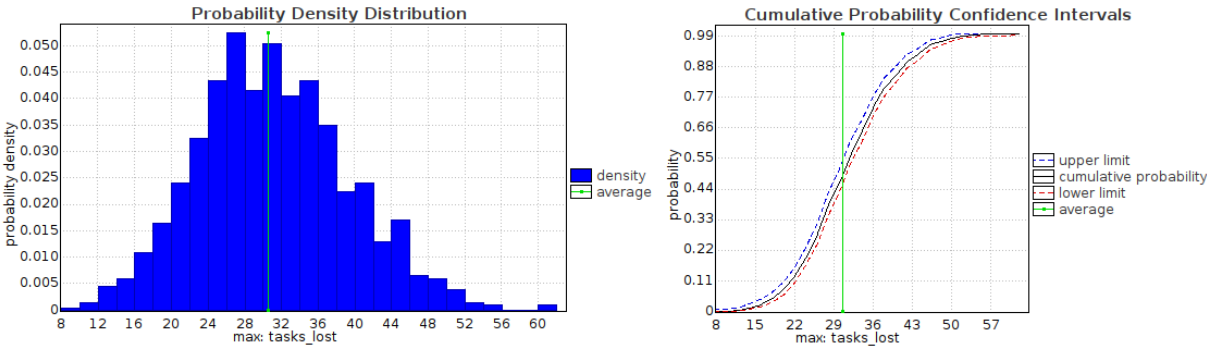


Figure 3.5: Scenario 2: probability distribution of maximum number of lost tasks

With an average of 1065.490 ± 0.381 tasks completed and 30.587 ± 0.672 tasks lost over 1000 runs.