Model Checking of Warehouse Robotics

Project Report
Formal Methods for Concurrent and Real-Time Systems

Leonardo Barilani, Marco Bonelli

POLITECNICO
MILANO 1863

June 27th, 2021

Title:	Model Checking of Warehouse Robotics	
Authors:	Leonardo Barilani	10528015
	Marco Bonelli	10522665
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:	https://github.com/leonardobarilani/warehouse-model-checking	
Copyright:	© 2021 Leonardo Barilani, Marco Bonelli	

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.

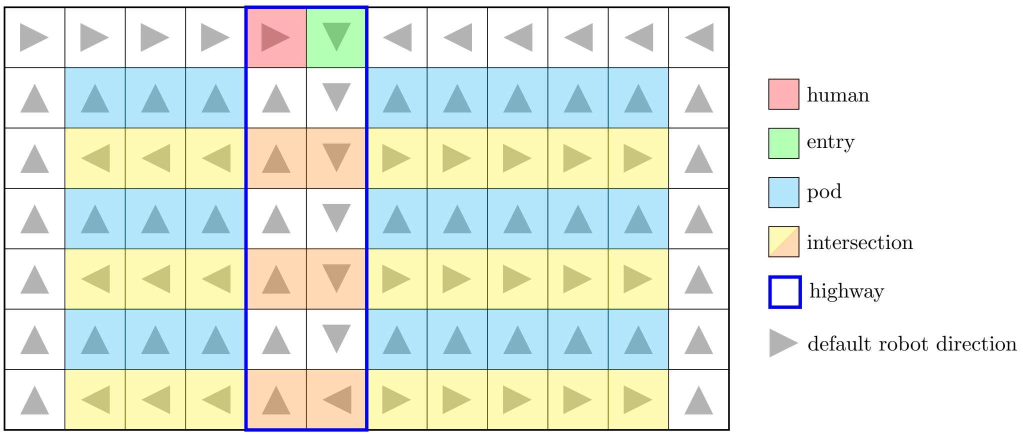

Figure 1.1: Warehouse layout

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

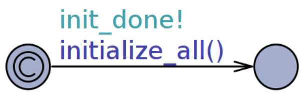

Figure 2.1: Initializer Timed Automaton

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: $\mathcal{N}(\mu_T, \sigma_T)$.

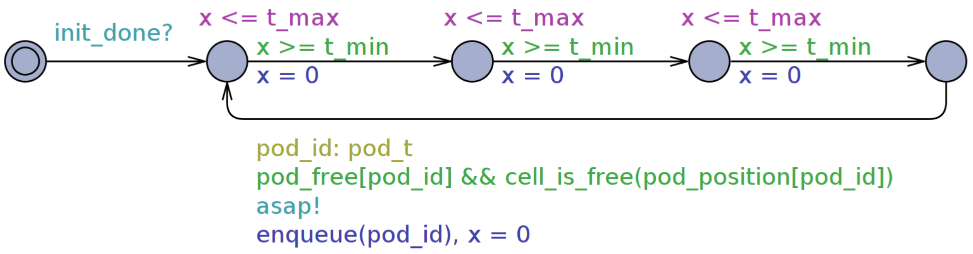

Figure 2.2: TaskGenerator Timed Automaton

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: $\mathcal{N}(\mu_H, \sigma_H)$.

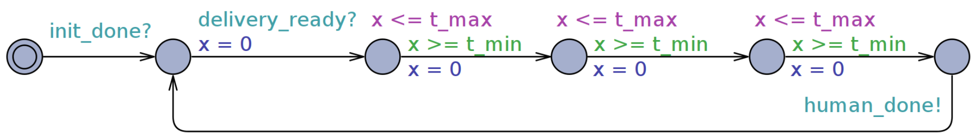

Figure 2.3: Human Timed Automaton

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.

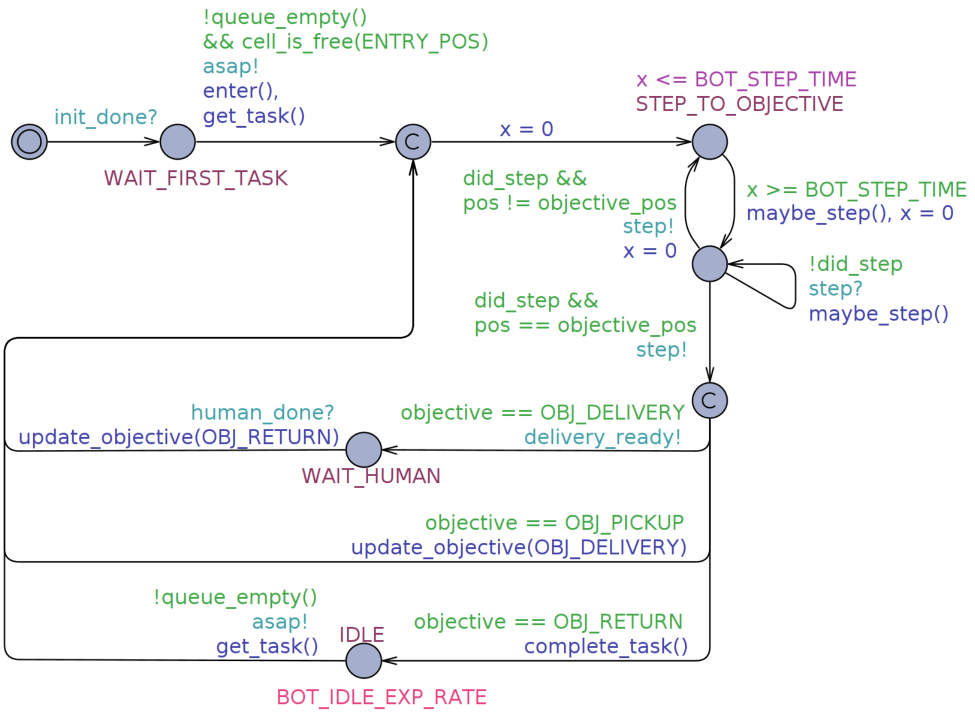

Figure 2.4: Bot Timed Automaton

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 https://github.com/leonardobarilani/warehouse-model-checking
[4]https://docs.uppaal.org/toolsandapi/verifyta
[5]https://uppaal.org/downloads/other/#uppaal-41
[6]https://matplotlib.org

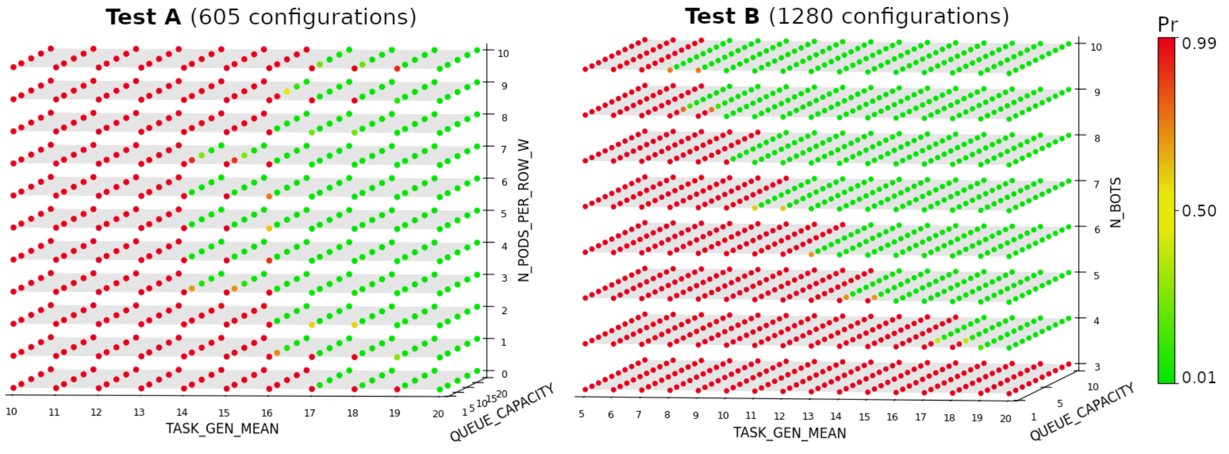

Figure 3.1: 4D plots of probability of losing any task - one data point per configuration

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:

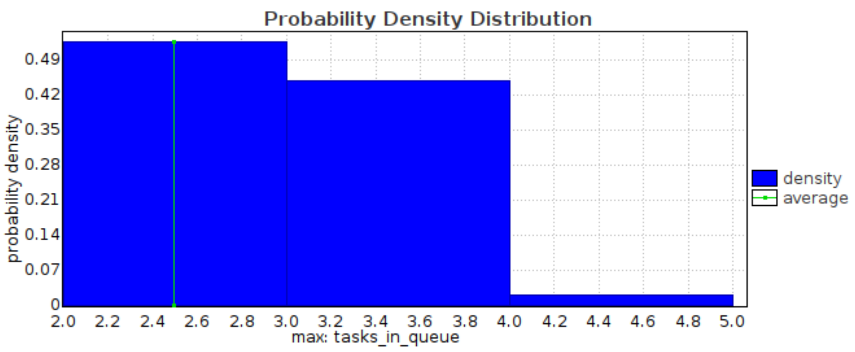

Figure 3.2: Scenario 1: probability distribution of maximum tasks in queue

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	$10 \rightarrow$ **7**
N_POD_ROWS	5
N_PODS_PER_ROW_W	5
N_PODS_PER_ROW_E	5
QUEUE_CAPACITY	$5 \rightarrow$ **10**
TASK_GEN_MEAN (μ_T)	$8 \rightarrow$ **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:

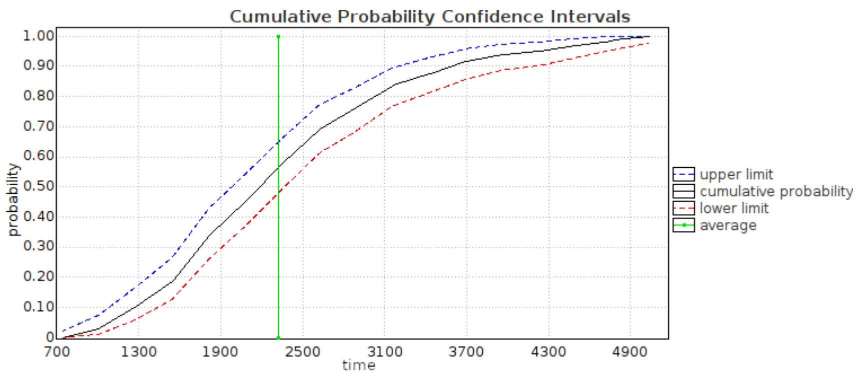

Figure 3.3: Scenario 2: cumulative probability distribution of losing any task over time

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:

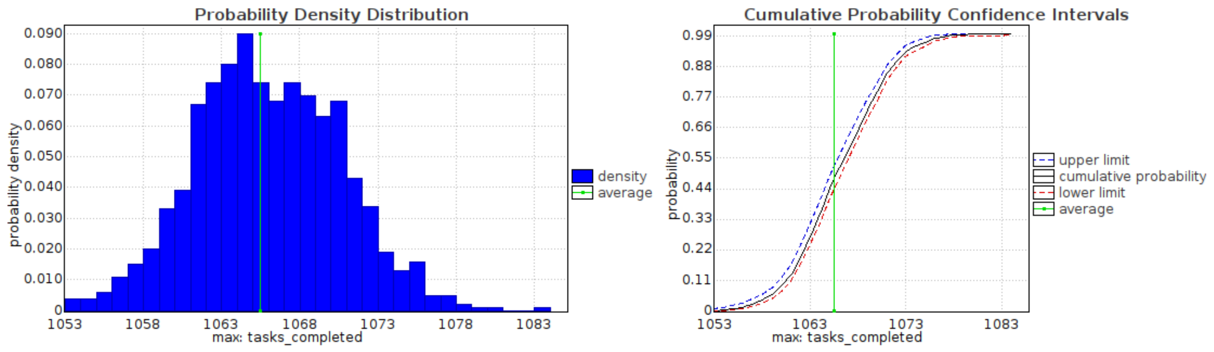

Figure 3.4: Scenario 2: probability distribution of maximum number of completed tasks

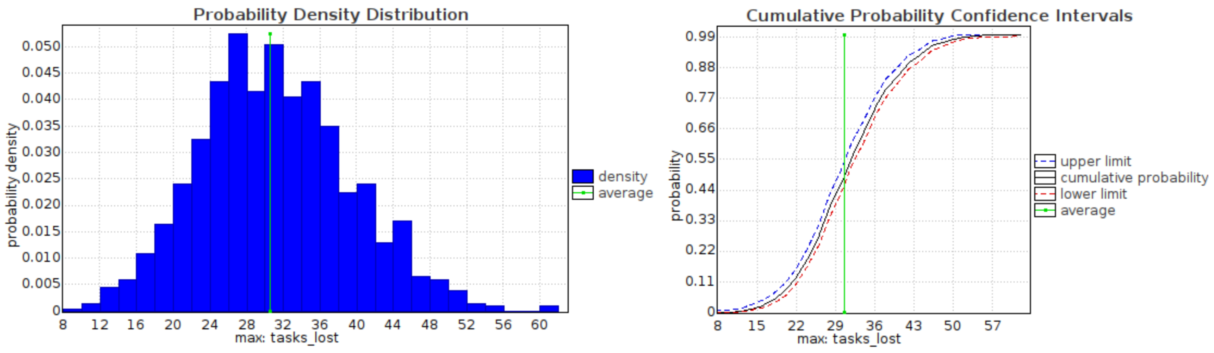

Figure 3.5: Scenario 2: probability distribution of maximum number of lost tasks

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