

# Lecture 14: Outline

- The Uppaal model checker (part 2)
- Two case studies with Uppaal

# Model checking of temporal and timing system properties (reminder)

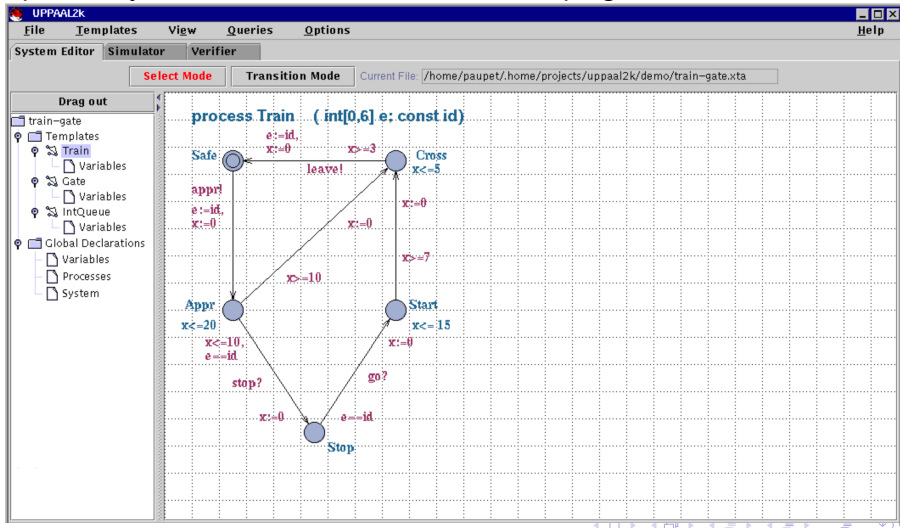
- One of the advantages of model checking is a possibility to check eventual reachability of certain desirable states
- Often, we need to enforce a time constraint on such eventual reachability. To reason about such properties, we need to introduce real time into our system models
- Statistical model checking allows us to verify more complex systems and properties, however with some statistical confidence instead of complete certainty

# The Uppaal tool (reminder)

- Uppaal – integrated tool environment for modelling, simulation and verification of real-time systems
- Developed jointly by Uppsala (Sweden) and Aalborg (Denmark) universities
- Uppaal consists of 3 main parts: a description language, a simulator, and a model-checker
- Recently, the Uppaal tool was extended to support statistical model checking (SMC), where properties are checked with some statistical confidence (probabilities)
- The exhaustive model checking of the system state space is replaced with simulating the system behaviour for some number of "system runs" and accumulating the statistical results

# The Uppaal graphical editor (reminder)

Allows modelling a system process as an automata. The circles and arrows represent system states and transitions. Time progress – within in a state



# The Uppaal simulator (reminder)

Provides visualisation of possible dynamic behaviours. Also shows traces (counter-examples) generated by the model checker

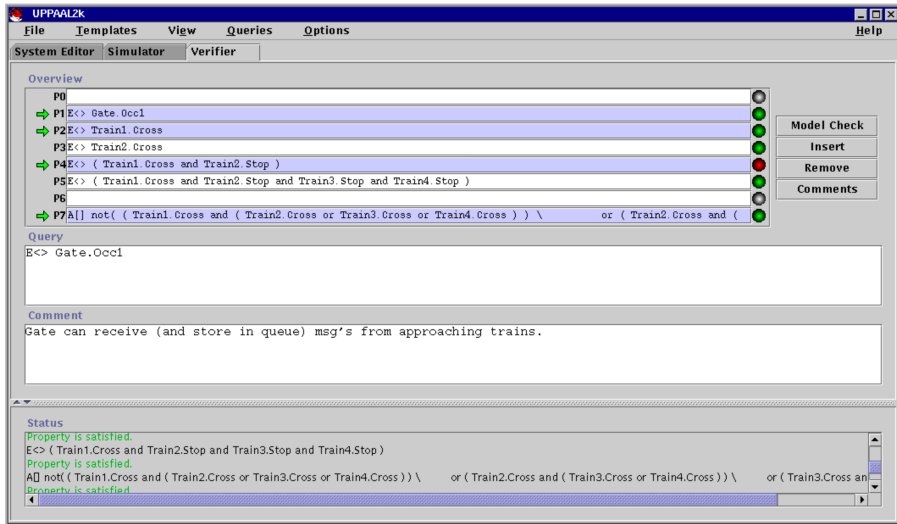
The screenshot displays the UPPAAL2K simulator interface. The top menu bar includes File, Templates, View, Queries, Options, and Help. The main window is divided into several panels:

- System Editor:** Contains tabs for System Editor, Simulator, and Verifier. The System Editor tab is active, showing a list of enabled transitions: (Train2.trans4, Gate.trans3), (Train3.trans4, Gate.trans3), and (Train4.trans3, Gate.trans4). Below this list are buttons for Next and Reset.
- Simulation Trace:** Displays a sequence of events: (Safe, Safe, Safe, Safe, s2, Start), (Train4.trans4, Gate.trans8), (Safe, Safe, Safe, Appr, s3, Start), (Gate.trans9, Queue.trans5), (Safe, Safe, Safe, Appr, Occ1, Start), (Train4.trans1), (Safe, Safe, Safe, Cross, Occ1, Start), (Train1.trans4, Gate.trans3), (Appr, Safe, Safe, Cross, Occ2, Start), (Train1.trans2, Gate.trans5), (Stop, Safe, Safe, Cross, s4, Start), and (Gate.trans10, Queue.trans5).
- Variables:** Lists the current state of variables:  $\text{el} = 1$ ,  $\text{Queue.len} = 4$ ,  $\text{Queue.list}[0] = 1$ ,  $\text{Queue.list}[1] = 0$ ,  $\text{Queue.list}[2] = 0$ ,  $\text{Queue.list}[3] = 0$ ,  $\text{Queue.list}[4] = 0$ ,  $\text{Queue.list}[5] = 2$ ,  $\text{Queue.i} = 0$ ,  $\text{Train1.x} \text{ in } [0, 5]$ ,  $\text{Train2.x} \text{ in } [10, \text{inf}]$ ,  $\text{Train3.x} \text{ in } [10, \text{inf}]$ ,  $\text{Train4.x} \text{ in } [0, 5]$ ,  $\text{Train2.x} - \text{Train1.x}$ ,  $\text{Train3.x} - \text{Train1.x}$ ,  $\text{Train4.x} - \text{Train1.x}$ ,  $\text{Train3.x} - \text{Train2.x}$ ,  $\text{Train4.x} - \text{Train2.x}$ , and  $\text{Train4.x} - \text{Train3.x}$ .
- State Transition Diagrams:** Four diagrams are shown, each representing a process:
  - process Train1:** A state machine with states Safe, Appr, Cross, and Stop. Transitions are labeled with events and guards. The initial state is Safe.
  - process Train2:** A state machine with states Safe, Appr, Cross, and Stop. Transitions are labeled with events and guards. The initial state is Safe.
  - process Gate:** A state machine with states Safe, Appr, Cross, and Stop. Transitions are labeled with events and guards. The initial state is Safe.
  - process Queue:** A state machine with states Safe, Appr, Cross, and Stop. Transitions are labeled with events and guards. The initial state is Safe.

At the bottom, there is a Trace File input field and buttons for Prev, Next, Replay, Open, Save, and Random. A speed slider is also present, ranging from Slow to Fast.

# The Uppaal model checker (reminder)

Allows to formulate and verify invariant/safety and reachability/liveness system properties by exploring the system state space



The screenshot shows the UPPAAL2K model checker interface. The main window is titled "UPPAAL2K" and has a menu bar with "File", "Templates", "View", "Queries", "Options", and "Help". Below the menu bar are three tabs: "System Editor", "Simulator", and "Verifier". The "Verifier" tab is active, showing a list of properties and their verification status.

**Overview**

Property	Status
P0	
→ P1 $E <> \text{Gate.Occ1}$	Green
→ P2 $E <> \text{Train1.Cross}$	Green
P3 $E <> \text{Train2.Cross}$	Green
→ P4 $E <> ( \text{Train1.Cross and Train2.Stop} )$	Red
P5 $E <> ( \text{Train1.Cross and Train2.Stop and Train3.Stop and Train4.Stop} )$	Green
P6	
→ P7 $A[] \text{not}( ( \text{Train1.Cross and } ( \text{Train2.Cross or Train3.Cross or Train4.Cross} ) ) \setminus \text{ or } ( \text{Train2.Cross and } ($	Green

**Query**

$E <> \text{Gate.Occ1}$

**Comment**

Gate can receive (and store in queue) msg's from approaching trains.

**Status**

Property is satisfied.  
 $E <> ( \text{Train1.Cross and Train2.Stop and Train3.Stop and Train4.Stop} )$   
Property is satisfied.  
 $A[] \text{not}( ( \text{Train1.Cross and } ( \text{Train2.Cross or Train3.Cross or Train4.Cross} ) ) \setminus \text{ or } ( \text{Train2.Cross and } ( \text{Train3.Cross or Train4.Cross} ) ) \setminus \text{ or } ( \text{Train3.Cross and } ($   
Property is satisfied.

# Case study: satellite data processing unit

- A real example: the European Space Agency (ESA) mission BepiColombo, sending an orbiter (spacecraft) to Mercury to collect scientific data
- Data processing unit (DPU): the core part of the orbiter, operating scientific instruments and producing scientific data
- Software written by the company Space Systems Finland
- Scientific research (including software modelling and verification) within the EU project Rodin
- Both Event-B (the Rodin platform) and Uppaal were used for that

# Bepi Colombo DPU: description

- DPU consists of the core software and software of scientific instruments
- The core software communicates with Bepi Colombo by sending telecommands (TCs), while the instrument software produces telemetrics (TMs) with scientific data
- Moreover, the spacecraft regularly produces housekeeping/diagnostics data (HKs) as special kind of TMs
- Both incoming TCs and outgoing TMs are stored in the respective (circular) buffers



# Bepi Colombo DPU: description

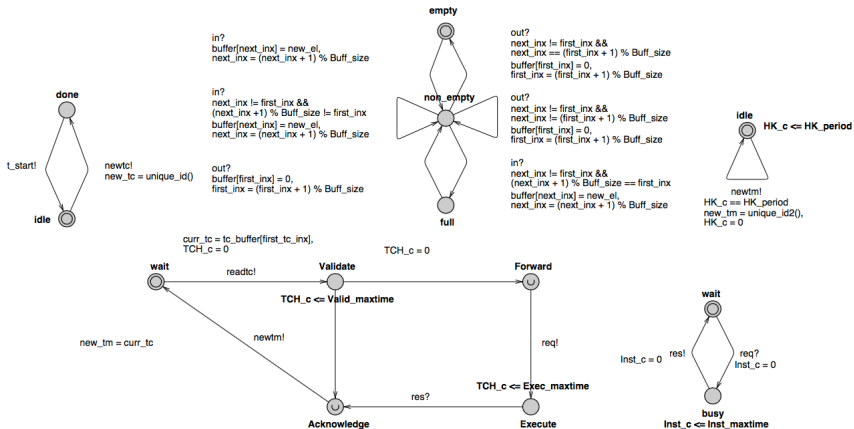
- The incoming TCs are put into the TC buffer (if it is not full)
- Each TC should be validated (checked for correct data according to the standard) before being handled
- Once validated, a TC is forwarded to the instrument software for execution (e.g., changing the operational mode, managing the memory, requesting a particular type of scientific data, etc.)
- A generated TM is put into the TM buffer (if it is not full) for sending back
- A separate housekeeping unit is responsible for regularly generating diagnostic reports and putting them into the TM buffer

The system parameters affecting the overall system performance:

- the size of circular buffers for storing TCs and TMs;
- the worst execution time for TC validation;
- the worst execution time for the instrument responding to the forwarded TC;
- the period of the process regularly generating housekeeping data returned as additional TM;
- the maximal delay before the generated TM is delivered
- ...

# Modelling the Bepi Colombo system in Uppaal

The Bepi Colombo system in the Uppaal notation:



**Fig. 2.** The BepiColombo process view model Uppaal notation.

## Annotations and their explanations:

- State transitions (arrows):  $x?$  and  $x!$  actions that are synchronised with other processes
- State transitions (arrows): conditions/guards enabling a transition
- State transitions (arrows):  $v = \text{expr}$  assignments to model variables
- States (circles): conditions or probabilistic rates (frequencies) on clock variables expressing time constraints for staying in that state
- ...

# Bepi Colombo DPU: properties to check

- For this system, we are interesting in two kinds of properties to verify (model check)
- First, we need to ensure that, for any received TC, the corresponding TM (indicating either success or failure) is eventually returned
- Second, we need to check that the worst execution time of TC handling does not exceed the pre-defined limit (depending on various system parameters such as the buffer size or the period of housekeeping data generation)

# Some examples of checked timing properties (reminder)

- $E<> p$ : there exists an execution path where  $p$  eventually holds;
- $A[] p$ : for all paths  $p$  always holds (invariant);
- $E[] p$ : there is an execution path where  $p$  always holds;
- $A<> p$ : for all paths  $p$  will eventually hold;
- $p \rightarrow q$ : whenever  $p$  holds,  $q$  will eventually hold.

# Bepi Colombo DPU: quantitative verification examples

- The first kind of properties can be expressed in Uppaal as, e.g.,:

`(new_tc==1) --> (last_tm==1)`

for the message id 1 (the property can be checked independently of concrete id values)

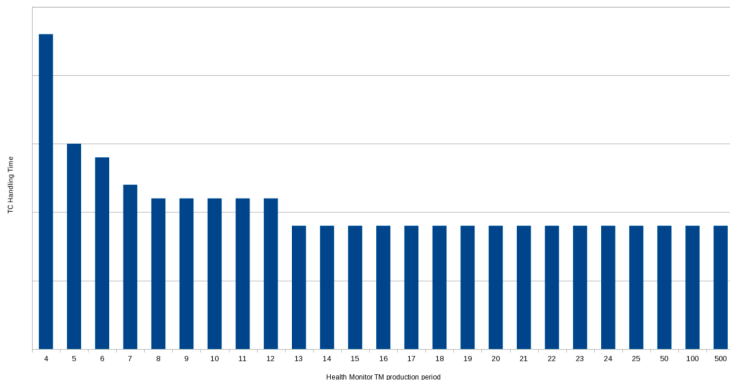
- The first kind of properties can be expressed in Uppaal as follows (using the values of clock variables):

`A[] (last_tm == 1 && Observer1.stop) imply  
(Observer1.Obs c < upper_bound)`

where `A[]` means "Always, for any execution path", while `Observer1` is a special process that starts the clock `Observer1.Obs c`, whenever a TC command with the id 1 is received, and stops it, once the corresponding TM is returned

# Quantitative assessment

Investigating how the HK production affects the overall TC handling time



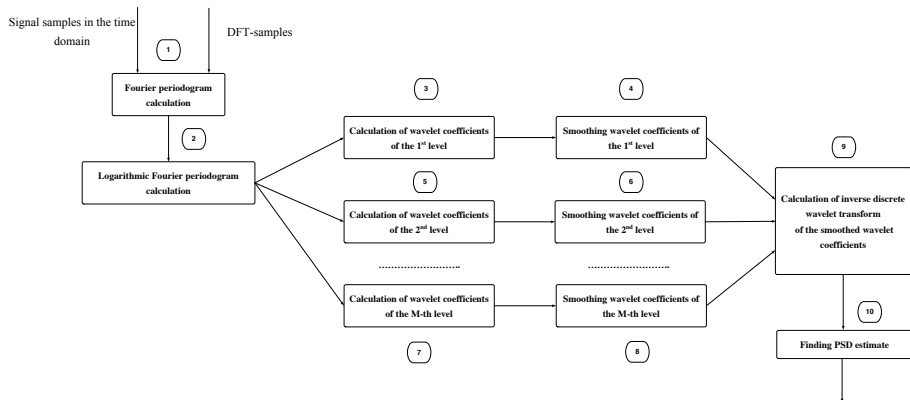
**Fig. 3.** The relationship between the period of TM production by *Health Monitor* (X-axis) and worst TC handling time (Y-axis). For periods less than 4, the system is potentially in livelock. X-axis numbers are the multiples of a base measuring constant – the TC decoding time.



# Another case study: signal processing in a distributed system

- Signal processing of acoustic signals in a dynamic distributed system with a number of data processing workers (servers), based on the pre-defined calculation algorithm
- Signal preprocessing from multi-channel observations (recorded sea sounds from reception hydrophones), filtering noises and other interferences
- Purpose of the project: to formally develop and quantitatively evaluate software architectures that most suitable for effective application of the proposed signal processing algorithm
- Again, both Event-B (the Rodin platform) and Uppaal (statistical version this time) were used for that

## Algorithm structure:



# Signal processing: system architecture

- One master component coordinating the overall execution of the algorithm
- Many worker components that are capable of executing specific calculations assigned by the master component
- Some algorithm phases involve (if possible) parallel computations by several workers. The optimal number of parallel computations that a specific phase can be split into is known beforehand
- The availability of workers is dynamic (i.e., some components can be busy/unavailable, they also can dynamically fail or recover)

# Signal processing: system parameters

System parameters as Uppaal textual declarations (in a separate file):

```
// Place global declarations here.

const int N = 10;           // number of components
typedef int[0,N-1] id_comp;

const int M = 16;           // number of possible parallel computations
typedef int [0,M-1] id_tasks;

const int TASK_TIME = 34;   // task calculation time
const int INIT_DELAY = 2;   // initial delay for receiving data
const int PERIOD = 5;       // monitoring period
const int MIN_DELAY = 1;    // minimal communication delay

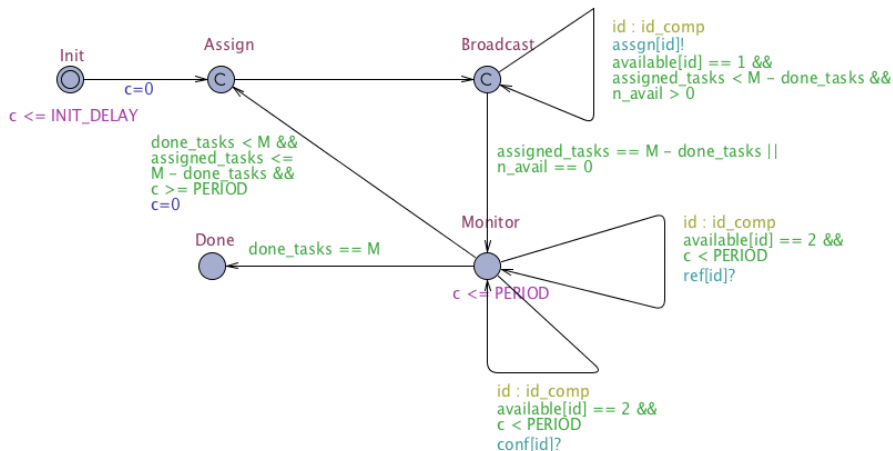
const int pw_aa = 95;       // probab. weight of component staying available
const int pw_au = 5;        // probab. weight of component becoming unavailable
const int pw_ua = 15;       // probab. weight of component becoming available
const int pw_uu = 85;       // probab. weight of component staying unavailable
const int pw_suc = 95;      // probab. weight of component finishing a given task
const int pw_ref = 5;       // probab. weight of component failing a given task

broadcast chan assign[N];   // channel for broadcasting task assignments
broadcast chan conf[N];     // channel for confirming task completion
broadcast chan ref[N];      // channel for refusing task execution

int[0,M] done_tasks = 0;    // number of completed tasks
int[0,M] assigned_tasks = 0; // number of assigned tasks
int[0,N] n_avail = 0;       // number of components available for task execution

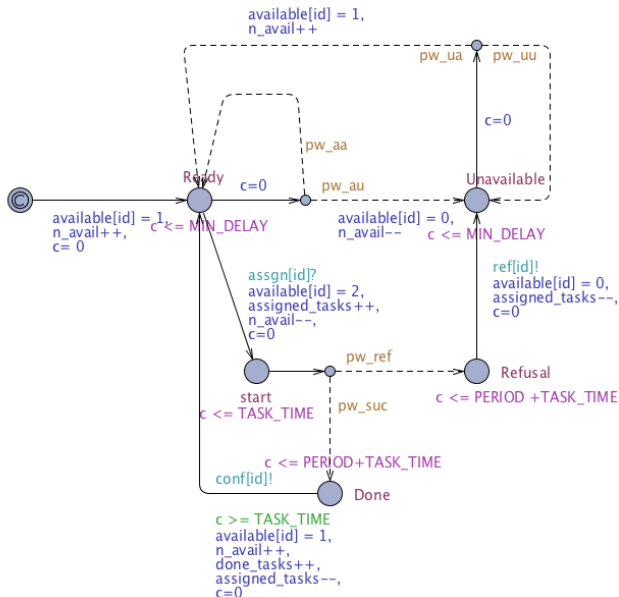
int[0,2] available[N] = {0,0,0,0,0,0,0,0,0,0}; // component availability status
```

# Signal processing: the Master component



Here `id : id_comp` is a component parameter (like ANY clause in Event-B)

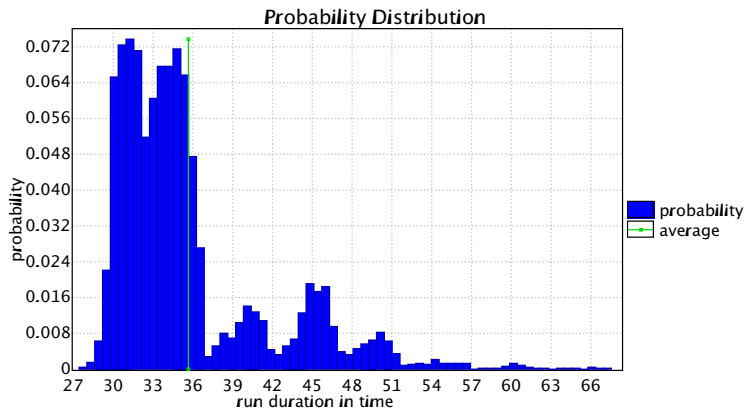
# Signal processing: the Worker component



# Signalling: quantitative verification examples

- A number of required time reachability properties, considering different value combinations for system parameters. All the verified properties are of the form:  
 $\text{Pr } [\leq \text{time\_bound}] (<> \text{Master.Done})$
- The result is the probability that the Master component eventually reaches the state Master.Done (i.e., the state where all the parallel task calculations are successfully completed) within the given time bound
- Moreover, the obtained results can be graphically plotted to show probability distribution or cumulative probability distribution for different values of the clock

Probability distribution (for the data size 20 and 10 worker components):



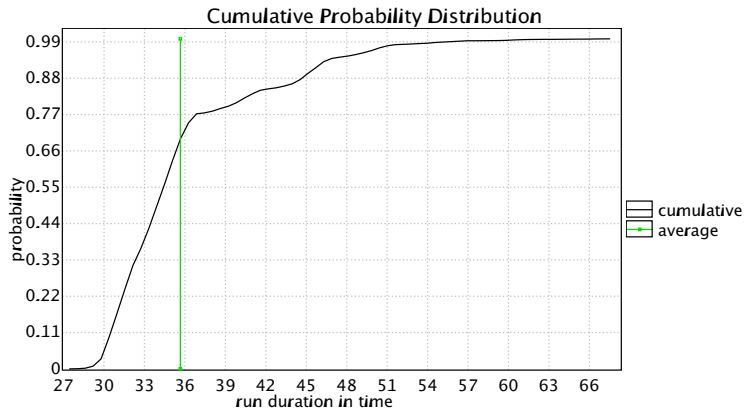
**Runs: 4612 in total, 4609 displayed, 3 remaining.**

**Probability sums: 0.99935 displayed, 0.000650477 remaining.**

**Minimum, maximum, average: 27.4211, 67.5414, 35.6614.**



Cumulative probability distribution (for the data size 20 and 10 worker components)::

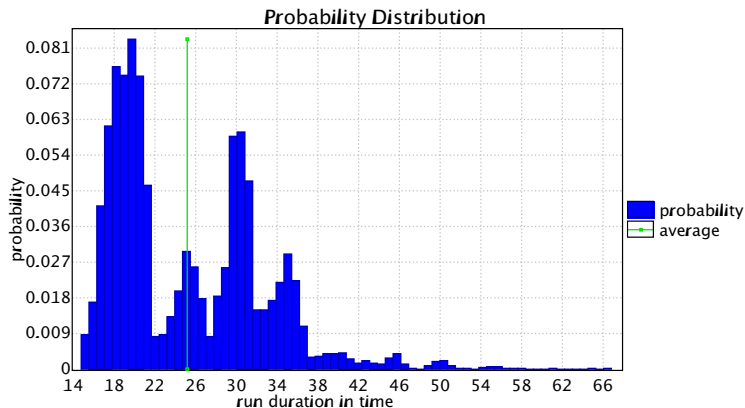


**Runs: 4612 in total, 4609 displayed, 3 remaining.**

**Probability sums: 0.99935 displayed, 0.000650477 remaining.**

**Minimum, maximum, average: 27.4211, 67.5414, 35.6614.**

Probability distribution (for the data size 20 and 16 worker components)::

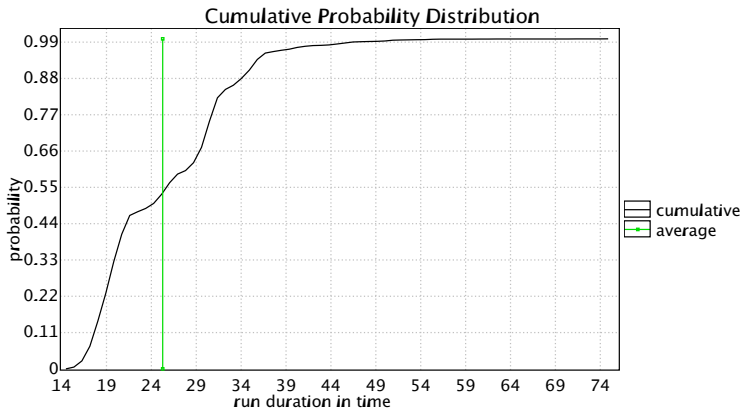


**Runs:** 4612 in total, 4612 displayed, 0 remaining.

**Probability sums:** 1 displayed, 0 remaining.

**Minimum, maximum, average:** 14.7551, 66.8134, 25.1675.

Cumulative probability distribution (for the data size 20 and 16 worker components)::

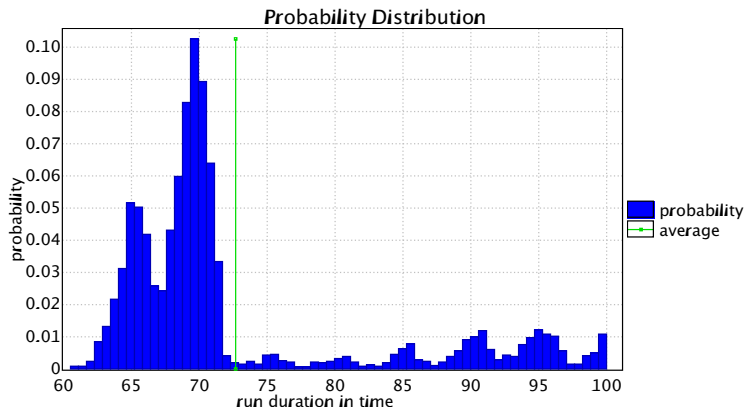


**Runs:** 4612 in total, 4612 displayed, 0 remaining.

**Probability sums:** 1 displayed, 0 remaining.

**Minimum, maximum, average:** 14.484, 74.8756, 25.2627.

Probability distribution (for the data size 40 and 10 worker components)::

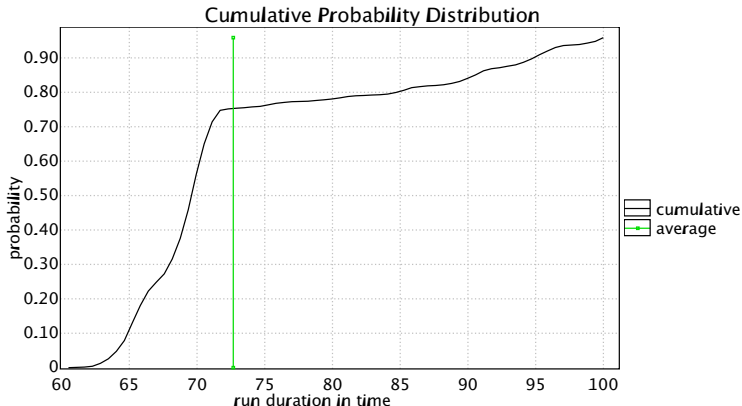


**Runs: 4612 in total, 4422 displayed, 190 remaining.**

**Probability sums: 0.958803 displayed, 0.0411969 remaining.**

**Minimum, maximum, average: 60.5077, 99.9952, 72.6749.**

Cumulative probability distribution (for the data size 40 and 10 worker components)::



**Runs:** 4612 in total, 4422 displayed, 190 remaining.

**Probability sums:** 0.958803 displayed, 0.0411969 remaining.

**Minimum, maximum, average:** 60.5077, 99.9952, 72.6749.

# Signal processing: analysis of the optimal system architecture

- Further human expertise needed to choose the best architecture from these generated basic analysis data
- Possible "rules of thumb" (heuristics): probability distribution should be similar to the normal probabilistic one, with average time being the only main peak. Or, cumulative distribution should reach 0.99 and plateau from there as soon as possible.
- Different artificial intelligence and machine learning techniques are also applicable here