Modeling and Verification of ROS Systems Using Stochastic Timed Automata

Peter Backeman[1] and Cristina Seceleanu[1]

Mälardalen University, Västerås, Sweden
{peter.backeman, cristina.seceleanu}@mdu.se

Abstract. Robotic systems often operate under real-time constraints, requiring timely responses to sensor inputs. Early consideration of such requirements during design is advantageous. The Robot Operating System (ROS) provides a mature framework for system setup and communication, with ROS2 offering real-time capabilities. However, determining the maximum reaction time within a ROS network is intricate due to complex variable processing and scheduling, especially with periodic and event-triggered tasks. In this report, we propose a model of ROS-based structural designs with timed automata semantics, facilitating real-time behavior analysis. We extend this model to incorporate non-deterministic execution time and probabilistic loads, employing statistical model checking (SMC) for scalability and accuracy. We compare against previous work to confirm the validity of our approach.

1 Introduction

Many robotic systems are subject to real-time constraints that should be obeyed. One such constraint can be that a system must react to a certain sensor input within a time bound (e.g., door opening when light sensor is activated). Ensuring such requirements already at early design stages, e.g., on the architecture level, is beneficial. Formal methods, e.g., model checking [3], is one such approach which is powerful. It can help establish both liveness and safety properties subject to timing constraints using timed automata.

When designing a robotic system there are many aspects to take into consideration. Computation and communication issues must be decided upon and implemented. To alleviate this, one can use a pre-existing framework that already solves many challenges. The Robot Operating System (ROS) provides a framework for setting up nodes and their communication. However, to establish properties of a ROS network, e.g., finding an upper bound for the maximum reaction time, is complex as it is affected by the run-times of the involved tasks and how these are scheduled. Furthermore, tasks can be both periodic (e.g., scheduled every second) or triggered by another function publishing new data. In this technical report, we present a model of ROS-based structural designs, and assign semantics to allow model checking with respect to real-time behavior to help establish such properties. In particular we focus on bounding an end-to-end reaction time. We assign semantics in the form of timed automata (TA) [1]

templates, yielding a precise definition of the underlying behaviour that uses ROS communication. We begin with basic semantics and validate it against previous work [14]. This is extended with non-deterministic execution times and probabilistic loads. For scalability and richness of modeling purposes, we employ the statistical model checking [9] technique, where properties are guaranteed to a specific degree of confidence, by using the UPPAAL SMC [8] tool, to verify properties of the model. We present the following contributions:

- Introduction of a pattern-based TA semantics of ROS networks, covering both a deterministic and probabilistic running times and loads.
- Validation of our base semantics towards previous simulation-based work.
- Application of UPPAAL/UPPAAL SMC to find maximum reaction times.

The report is structured as follows: after the preliminaries in Sec. 2, we present our formalization of ROS systems in Sec. 3, followed by our TA semantics in Sec. 4 and its validation in Sec. 5. Afterwards, we show how the semantics can be extended in several directions in Sec. 6. Finally, related work is presented in Sec. 7, and our conclusions and future work in Sec. 8.

2 Preliminaries

In this section we briefly present scheduling of, and communication between, tasks in the robot operating system and summarize timed automata.

2.1 Robot Operating System

The Robot Operating System (ROS) is intended to provide developers with a set of open-source software frameworks, libraries, and tools to create applications for robots. The platform offers services for a heterogeneous computer cluster, such as hardware abstraction, device control, implementation of functionalities, message-passing between processes, and package management [12]. The operating system's version ROS1 underwent a major revision and became ROS2 [11], bringing many improvements, most notably the Data Distribution Service (DDS) support. DDS acts as middle-ware for inter-node communication, using the quality-of-service profile to provide real-time communication, scalability, performance enhancement, and security benefits not found native in ROS1.

The ROS2 platform has already been used in designing the communication architecture of collaborative and intelligent automation systems [6], or of self-driving cars that require safe and reliable real-time behaviour [13]. Most such robotic systems are subject to real-time constraints that, if not met, might result in issues of various severity degrees, from the application failing to perform correctly to a lowered performance of the overall system. Verifying if such undesired issues occur in a ROS-based robotic system, already at a structural design level, is very desirable. To achieve this, the basic, high-level communication and computation paradigms of ROS need to be given formal semantics, to be amenable to analysis, e.g., via model checking.

Scheduling A ROS network consists of *nodes* and *topics*. A node is a component which has one or more *tasks* (i.e., callbacks) which can be scheduled for execution. When a task is scheduled, a *job* is instantiated and put in the scheduling queue. When a job has finished, it can *publish* its value to a topic, or store it locally on the node. The ROS scheduling works by at each *polling point*, i.e., beginning of a *processing window*, pick one job from each task (which has queued job) and schedule them according to priority (with timers always having higher priority than subscribers). Then each of the tasks are executed and afterwards a new polling point is reached. If at any polling point, there are no jobs queued, then the scheduler idles until a task becomes scheduled. The process is illustrated in Fig. 1.

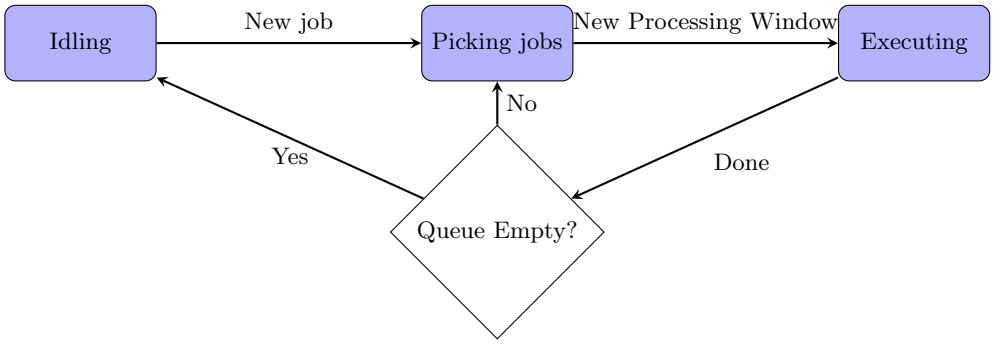


Fig. 1. Scheduler in ROS.

2.2 Timed Automata

A Timed Automaton (TA), as used in the model-checker UPPAAL [8] , is defined as a tuple, $\langle L, l_0, C, A, V, E, I\rangle$, where L is the set of finite locations, l_0 is the initial location, V is the set of data variables, C is the set of *clocks*, $A = \Sigma \cup \tau$ is the set of *actions*, where Σ is the finite set of *synchronizing actions*($c!$ denotes the send action, and $c?$ the receiving action) partitioned into inputs and outputs, $\Sigma = \Sigma_{\mathrm{i}} \cup \Sigma_{\mathrm{o}}$, and $\tau \notin \Sigma$ denotes internal or empty actions without synchronization, $E \subseteq L \times B(C,V) \times A \times 2^C \times L$ is the set of *edges*, where $B(C,V)$ is the set of *guards* over C and V, that is, conjunctive formulas of clock constraints ($B(C)$), of the form $x \bowtie n$ or $x - y \bowtie n$, where $x, y \in C$, $n \in \mathbb{N}$, $\bowtie \in \{<, \leq, =, \geq, >\}$, and non-clock constraints over V ($B(V)$), and $I : L \longrightarrow B_{dc}(C)$ is a function that assigns *invariants* to locations, where $B_{dc}(C) \subseteq B(C)$ is the set of downward-closed clock constraints with $\bowtie \in \{<, \leq, =\}$. Invariants bound the time that can be spent in locations, ensuring the progress of TA's execution. An edge from location l to location l' is denoted by $l \xrightarrow{a,g,r,u} l$, where a is an action, g is the guard of the edge, r is the clock reset set, that is, the clocks that are set to 0 over the edge and u is an update action. Variables are initialized

with a value of zero and an update action updates the value of zero, one or more variables. A location can be marked as *urgent* or *committed*, indicating that time cannot progress in such locations. The latter is more restrictive, indicating that the next edge to be traversed needs to start from a *committed* location.

The semantics of TA is a *labeled transition system*. The states of the labeled transition system are pairs (l, u), where $l \in L$ is the current location, and $u \in R^C_{\geq 0}$ is the clock valuation in location l. The initial state is denoted by (l_0, u_0), where $\forall x \in C,\ u_0(x) = 0$. Let $u \vDash g$ denote the clock value u that satisfies guard g. We use $u + d$ to denote the time elapse where all the clock values have increased by d, for $d \in \mathbb{R}_{\geq 0}$. There are two kinds of transitions:

(i) *Delay transitions*: $< l, u > \xrightarrow{d} < l, u + d >$ if $u \vDash I(l)$ and $(u + d') \vDash I(l)$, for $0 \leq d' \leq d$, and

(ii) *Action transitions:* $< l, u > \xrightarrow{a} < l', u' >$ if $l \xrightarrow{g,a,r} l', a \in \Sigma, u \vDash g$, clock valuation u' in the target state (l', u') is derived from u by resetting all clocks in the reset set r of the edge, such that $u' \vDash I(l')$.

A real-time system can be modeled as a *network of TA* (NTA) composed via the parallel composition operator ("||"), which allows an individual automaton to carry out internal actions, while pairs of automata can perform handshake synchronization. The locations of all automata, together with the clock valuations, define the state of an NTA. The properties to be verified by model checking on the resulting NTA are specified in a decidable subset of (Timed) Computation Tree Logic ((T)CTL), and checked by the UPPAAL model checker.

UPPAAL is also capable of handling *statistical model checking* (SMC) [9], where simulations are used to extract information of the system. In UPPAAL SMC, automata have a stochastic interpretation based on: (i) the probabilistic choices between multiple enabled transitions (uniform distribution by default, marked with weighted probabilities otherwise[1]), and (ii) the non-deterministic time delays that can be refined based on probability distributions, either uniform distributions for time-bounded delays or user-defined exponential distributions for unbounded delays. SMC has the downside of not providing full guarantees of results, but allows handling models of much larger sizes, as well as including probabilistic aspects of the system.

3 Formalization of ROS systems

In this section, we provide a formalization of constrained ROS systems. We assume each node can be *subscribed* to zero or more nodes, *read* from zero or more variables, *publish* to zero or one topic, and *write* to zero or one variables. We assume three kinds of nodes: timer nodes, subscription nodes and data-generator nodes. A *timer node* is triggered with periodic intervals and then schedules a task to publish or write its result. A *subscription node* has a special *triggering subscription*, and schedules a task to publish or write its result whenever data is published onto the triggering topic. Finally, a *data-generator* node is a timer

[1] We annotate edge guards with ?*prob* to denote that the edge weight is *prob*

node which has no subscriptions nor reads from any variable. For simplicity, we assume that there is only one publishing node per topic and that each variable is written to by at most one node.

3.1 Nodes

We define a set of nodes $\mathcal{N}$, a set of topics $\mathcal{T}$ and a set of global variable $\mathcal{V}$. For convenience, we introduce a unique topic and variable for each task τ denoted by $\mathcal{T}(\tau)$ and $\mathcal{V}(\tau)$ respectively. We define each kind of node separately:

Definition 1. *A* timer node $tn = (p, wcet, S, St, D, t, v)$*, where:*

- $p \in \mathbb{N}^+$ *is the period,*
- $wcet$ *is the WCET of the main task,*
- $S = \{s_1, \ldots, s_n\}, s_i \in \mathcal{T}$*, are the subscribed topics,*
- $St = \{st_1, \ldots, st_n\}, st_i \in \mathbb{N}$*, are the WCET of subscription tasks,*
- $D \subseteq \mathcal{V}$*, are the read-variables,*
- $t \in \mathcal{T}$ *is the result-topic,*
- $v \in \mathcal{V}$ *is the write-variable.*

Intuitively, a timer node is activated each p period, creating a job of the main task. The node subscribes to the topics S and uses the data from the variables D. Whenever a message is retrieved on topic s_i, a job is created to processes the retrieved value, with WCET st_i.

Definition 2. *A* subscriber node $sn = (s, wcet, S, St, D, t, v)$*, where:*

- $s \in \mathcal{T}, s \notin S$*, is the triggering topic,*
- $wcet$ *is the WCET of the main task,*
- $S = \{s_1, \ldots, s_n\}, s_i \in \mathcal{T}$*, are the subscribed topics,*
- $St = \{st_1, \ldots, st_n\}, st_i \in \mathbb{N}$*, are the WCET of subscriptions tasks,*
- $D \subseteq \mathcal{V}$*, are the read-variables,*
- $t \in \mathcal{T}$ *is the result-topic,*
- $v \in \mathcal{V}$ *is the write-variable.*

A subscriber node works like a timer node, except it is instead triggered whenever a message is received from the triggering node s.

Definition 3. *A* data-generator node $dn = (p, wcet, t, v)$*, where:*

- $p \in \mathbb{N}$*, is the period,*
- $wcet$ *is the WCET of the main task,*
- $t \in \mathcal{T}$ *is the result-topic,*
- $v \in \mathcal{V}$ *is the write-variable.*

A data-generator works as a timer but has no subscriptions nor read-variables.

Example 1. Consider the small system in Fig. 2. It has three nodes, a sensor node which publishes data to the filter node, which in turn publishes data to the actuator node (which is triggered by a timer). We will be interested in measuring the reaction time from the sensor to the actuator. Each component can be formalized (for particular values for WCET, etc) as:

– the sensor, a data-generator node $ns = (40, 10, \mathcal{T}(ns), \emptyset)$,
– the filter, a subscriber node $nf = (\mathcal{T}(ns), 10, \emptyset, \emptyset, \emptyset, \mathcal{T}(nf))$,
– the actuator, is a timer node $na = (100, 20, \{\mathcal{T}(nf)\}, \{10\}, \emptyset, \emptyset, \mathcal{T}(na))$,

Fig. 2. Small ROS network.

3.2 Tasks

Each node contain one or more task. All nodes have a *main task* which is responsible for combining all data, compute and publish/write the output. Additionally, every node has for each (non-triggering) a task is responsible for retrieving the published data, process it, and store it in a local variable. All tasks have an assigned WCET. Let τ_n denote the main task of node n and τ_n^i to refer to the subscription task for subscribed topic s_i of node n.

Example 2. The system presented in the previous example contains three nodes. The sensor and filter node will only contain two main tasks: τ_{ns} triggered by a timer and τ_{nf} triggered by a subscription). The actuator node will have one main task τ_{na} (triggered by a timer) and one subscription task τ_{na}^1 responsible for reading the value from the filter node.

3.3 Job chains

In this report we are looking for maximum reaction times. To define the notion of reaction time, we first present the notion of task and (forward) job chains:

Definition 4. *A* task chain $\mathfrak{T} = \{\tau_1, \ldots, \tau_n\}$ *is a sequence of tasks s.t.:*

– *the task* τ_1 *is the main task of a data-generator, and*
– $\forall \tau_i \in [\tau_1, \ldots, \tau_{n-1}]$ *either:*
  • τ_i *stores its result in* $\mathcal{V}(\tau_i)$ *and* τ_{i+1} *reads from* $\mathcal{V}(\tau_i)$*, or*
  • τ_i *publishes ts result to* $\mathcal{T}(\tau_i)$ *and* τ_{i+1} *subscribes to* $\mathcal{T}(\tau_i)$*.*

Definition 5. *A* (forward) job chain $J = \{j_1, \ldots j_n\}$ *is a sequence of jobs s.t. for a task chain* $\mathfrak{T}$*:*

– $\forall j_i \in J, j_i$ *is an instance of* τ_i*.*
– $\forall i \in [j_1, \ldots, j_{n-1}]$ *either:*
  • j_{i+1} *is the unique earliest job of* τ_{i+1} *which reads the result from* j_i*, or*
  • j_{i+1} *is the unique earliest job of* τ_{i+1} *which receives the result from* j_i *(through subscription).*

Let $start(\tau)$ and $end(\tau)$ represent the start-time and end-time of the task τ respectively, where $start(\tau), end(\tau) \in \mathcal{N}$.

Definition 6. *For a job chain* $J = \{j_1, \ldots, j_n\}$*, the* reaction time $rt(J) = end(j_n) - start(j_1)$.

Example 3. Consider again the small example network in Figure 2, now executed with scheduling as presented in Section 2.1. The resulting trace is shown in Figure 3. The job chain $J = \{\tau_{ns_1}, \tau_{nf_1}, \tau^1_{na_1}, \tau_{na_1}\}$ has a reaction time of 90. Thus if an (external) value is modified at time step zero (right after the sensor task is fired), it would be read at time step 50, which yields a reaction time for the external event of $50 + 90 = 140$.

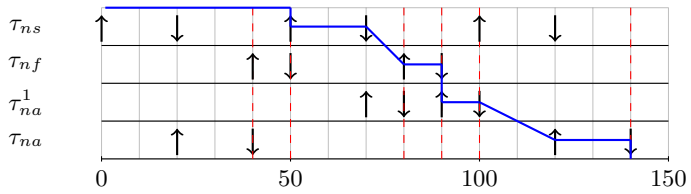


Fig. 3. Schedule for small ROS network example. Processing windows are separated by red dashed lines. An upwards arrow indicates the start of a job and downwards indicates the end. The blue line traces the reaction time.

In this report, we focus on the case where there is only one executor (i.e., processing unit), on which all functions are executed, and we further assume that there are no cycles in the dependencies between nodes. Then we can define the maximum reaction time for a given ROS system:

Definition 7. *We define the* maximum reaction time *from data-generator* $dn = (p, wcet, t, v)$ *to node* n *for a ROS system as:* $max_{J \in \mathcal{J}}(p + rt(J))$*, where* $\mathcal{J}$ *is the set of all job-chains beginning with* dn*.*

Note that we add the period p of the data-generator to account for the worst-case delay of a data-generator observing an external event. We acknowledge that for an over-utilized system, no finite maximum reaction time may exist, but for a non-over-utilized system, the maximum reaction time will be finite.

4 TA Semantics for ROS

In this section, we present a formalization of ROS semantics in UPPAAL timed automata. As we focus on analyzing the *end-to-end maximum reaction time* – we do not model how data is processed, but rather the age of data and check the resulting age at the destination. In particular, we instantiate with a particular task chain $\mathfrak{T}$ in mind, such that the resulting system considers the reaction time of $\mathfrak{T}$.

We introduce TA templates which are instantiated depending on the nodes of the ROS system, and the task chain which should be monitored. Note that in this report we assume that all nodes are scheduled on the same host (i.e., share the same CPU). We begin by introducing the global variables and functions used, followed by each TA template. Finally we describe how a particular system is instantiated. For simplicity, we assume that each task publishes to its dedicated topic and/or writes to its dedicated variable.

4.1 Constants, Variables, Functions and Channels

Table 1 show a set of constants, variable, functions and channels. Some are used for easier reading, while some have a semantic impact. In particular, `BUF_SIZE` and `MONITORS` must be set sufficiently large. In the remained of this report we assume that this is the case. Note that `PRIO[C]`, `WCET[C]`, `PUBLISHER[C]` and `WRITER[C]` must also be initialized with proper values. As mentioned we only allow nodes to either publish or write the results to a topic or variable. Therefore, the *queue_job* is overloaded to accept either a topic or a global variable.

4.2 TA Templates

The semantics are given as template-based instantiation, i.e,. for each task in the ROS system, we introduce an UPPAAL TA, based on the templates given in this section. Furthermore, teach TA is given a unique ID. We are interested only in the reaction time of data processed in the task chain $\mathfrak{T}$ under analysis. Thus we can ignore all other data values (however me must remember to trigger subscription tasks). Thus, every task only needs to consider the received and read values from the previous task of the task chain.

Data-generator The template for a data-generator $dn = (p, wcet, t, v)$, is instantiated given its *task_id* id and its period p, the remaining values are used in the global variables. The data-generator is responsible for generating a value each *period* p, then queuing a job of type τ_{dn}. There is also an extra Boolean parameter: if m is true, it will also activate the monitor whenever a value is read (it is set to true for the first task of $\mathfrak{T}$). The template is shown in Figure 4.[2] Also, location l_{fire} is marked committed.

Monitored Data-generator : A monitored data-generator is identical to a data-generator with the difference that the edge from l_{fire} to l_{wait} is replaced by:

$$l_{fire} \xrightarrow{start\_monitor^{!},\emptyset,x,queue\_job(id,PAYLOAD)} l_{wait}$$

Intuitively, this means that when a monitored data-generator publishes data, a monitored is requested to be started and a payload value is queued.

[2] Graphical representations from UPPAAL of all TA are found in the appendix.

Constant	Description
`EMPTY`	Value for representing empty data.
`MONITORS`	Number of parallel monitors.
`FIRST_PAYLOAD`	Represents the value of the first monitored package.
`MONITOR_FREE`/`MONITOR_SENT`	Status value of a free/busy monitor.
`BUF_SIZE`	Buffer size of queues.
`PRIO[C]`	Priority of each task.
`WCET[C]`	WCET of each task.
Variables	**Description**
payload	value of next payload value to be sent.
last_payload	value of last sent payload.
next_monitor (*nm*)	Index of next free monitor.
last_monitor (*lm*)	Index of oldest busy monitor.
published_data (*pd*)	Value of last published data.
monitor_status[MONITORS]	Status of monitors.
monitor_payload[MONITORS]	Payload monitored.
QUEUES[C][BUF_SIZE]	Job queues (one for each task).
QUEUES_COUNT[C]	Number of jobs in each queue.
HOST_JOBS[C][2]	ID and data of scheduled jobs.
HOST_JOBS_COUNT	Number of scheduled jobs.
DATA[C]	Unique variable for each task.
Function	**Description**
waiting_jobs	Returns the number of jobs waiting for the host.
queue_job	Add a job to the host queue.
dequeue	Dequeue the first job from the host queue.
schedule	Sort all jobs in the host waiting list by priority.
take_jobs	Take (up to) one job of each task.
next_job_idx	Get the index (i.e., task id) of next job.
get_data	Get data for a specific task.
assign_monitor	Assign next free monitor to current package.
free_monitor	Free all finished monitors.
Channel	**Description**
`new_job`	Announce the (possible) scheduling of new job.
`start_monitor`	Signal to start monitoring next data.
`publish[C]`	Unique topic for each task.

Table 1. Constants, variables, functions and channels of the model.

$$
\begin{aligned}
&L = \{l_{wait}, l_{fire}\}, \quad \ell_0 = l_{wait}, \quad C = \{x\}, A = \{new\_job^{!}, new\_job^{?}\},\\
&V = \emptyset, \quad I = \{l_{wait} \mapsto x \leq p\},\\
&E = \{l_{wait} \xrightarrow{new\_job^{!}, x=p, \emptyset, \emptyset} l_{fire},\\
&\quad l_{wait} \xrightarrow{new\_job^{?}, x=p, \emptyset, \emptyset} l_{fire},\\
&\quad l_{fire} \xrightarrow{\tau, \emptyset, x, queue\_job(id, EMPTY)} l_{wait}\}
\end{aligned}
$$

Fig. 4. Template of a Data-Generator.

Subscriber A subscriber $sn = (s, wcet, S, St, D, t, v)$, is instantiated given three parameters: s, *task_id* and a *data_source*. The subscriber waits for a message to be published to s and then queues a job to publish the result, based on data from *data_source* to *task_id* topic. The template is shown in Figure 5.

$$L = \{l\}, \quad \ell_0 = l, \quad C = \emptyset, A = \{s^?\}, V = \{v_{published\_data}\}, I = \emptyset,$$
$$E = \{l \xrightarrow{s^?, \emptyset, \emptyset, queue\_job(id, v_{data\_source})} l\}$$

Fig. 5. Template of a Subscriber.

Timer A timer node $tn = (p, wcet, S, St, D, t, v)$ is instantiated using three parameters: p, *task_id* and *data_source*. The timer is activated each *period* p. The parameter *data_source* is set to data source of $d \in D \cup \{v_{st_i}\}$. If two or more timers are activated at the same time instant, one of them will use the active synchronization $new\_job^!$ and all others will follow using $new\_job^?$, ensuring that all jobs are added at the same time point. The template is shown in Figure 6.

$$L = \{l\}, \quad \ell_0 = l, \quad C = \{x\}, A = \{new\_job^!, new\_job^?\},$$
$$V = \emptyset, I = \{l \mapsto x \leq p\},$$
$$E = \{l_{wait} \xrightarrow{new\_job^!, x=p, x, queue\_job(id, data\_source)} l_{wait},$$
$$l_{wait} \xrightarrow{new\_job^?, x=p, x, queue\_job(id, data\_source)} l_{wait}\}$$

Fig. 6. Template of a Timer.

Host A host is the most complicated template, but has no parameters. The host waits for waiting jobs, and when present picks (up to) one from each task and schedules them according to priority. In the bottom half of the automaton it then simulates the execution of each job with reading, storing and publishing data accordingly. The host sends a message on the corresponding node-specific topic when finishing executing a job with the resulting data written to the associated global variable. The template is shown in Figure 7. Also, locations l_{check}, l_{next} and l_{done} are marked urgent, and l_{loop} is marked committed.

Monitor A monitor is instantiated with two parameters: *actuator*, the last task of the task chain, and the period p of the first task if the chain. The monitor waits for a data-generator to start monitoring and assigns a free monitor (setting the clock to p to allow for worst-case analysis). The location measure is reached whenever the monitored actuator publishes data, so queries can be over this

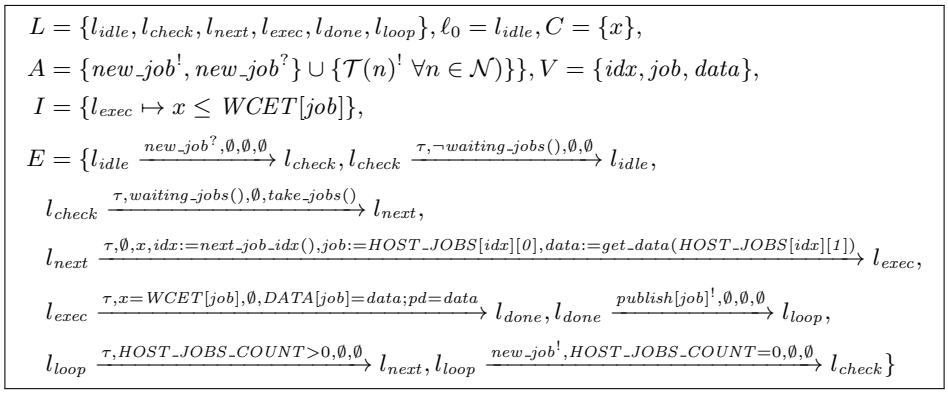


Fig. 7. Template of a Host.

location to check worst-case reaction times. The template is shown in Figure 8. Also, location $l_{mesuare}$ is marked committed.

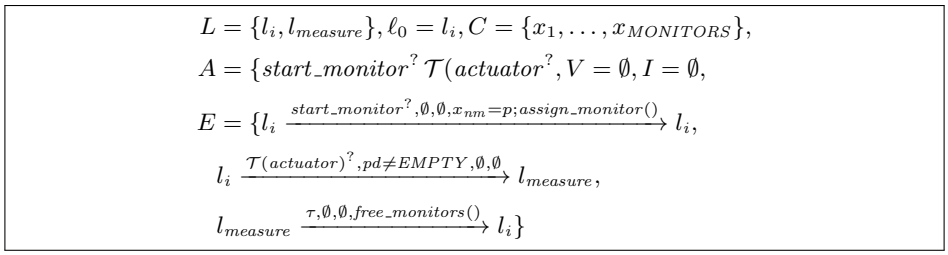


Fig. 8. Template of a Monitor.

4.3 Instantiation

For a given ROS network, each component is instantiated with the corresponding template using suitable values for the parameters. Each component also is given a global integer id (from zero and up), and the global constants `PRIO` and `WCET` are set accordingly. The first data-generator node of the task chain $\mathfrak{T}$ is instantiated as a monitored data-generator, while remaining data-generator nodes are instantiated as regular data-generators. The resulting network will be deterministic insofar that the value of the clocks in the monitor automaton when measured in the state *measure* will always be the same.

Example 4. We instantiate the ROS network from Ex. 3 according to the above formula and obtain five TAs: a Monitored Data-Generator($task\_id = 3, p = 40$), a Subscriber ($task\_id = 2, s = \mathcal{T}(3), data\_source = pd$), a Subscriber ($task\_id = 1, s = \mathcal{T}(1), data\_source = pd$), a Timer ($task\_id = 0, data\_source = \mathcal{V}(1)$), a Monitor ($actuator = 0, p = 40$), and a Host. Note that when $data\_source = pd$,

this means that the data should be read from the subscribed topics most recently published data.

4.4 Queries

Given an instantiated ROS system we can establish an upper bound on the maximum reaction time for $\mathfrak{T}$ using the following TCTL query:

$$\Box monitor.measure \rightarrow monitor.x[last\_monitor] \leq t$$

Intuitively, this query checks that reaction times are lesser than t, i.e., that t is an upper bound. This bound is not guaranteed to be tight. We can find a tight bound by using the following query to check if reaction time can exceed than t:

$$\diamond monitor.measure \ \wedge monitor.x[last\_monitor] \geq t$$

To use this to find an upper bound, the above query is used using $t = 0$. When UPPAAL finds a greater bound t', the query is checked once again with the new bound t'. This process is repeated until UPPAAL states that no greater value exists, establishing the final value of t to be the upper bound. Furthermore, UPPAAL allows the extraction a trace for a given bound.

Example 5. If we use UPPAAL to query for an upper bound for Example 4, we can find that the upper bound is 190, and extract the trace shown in Figure 9. Note that the time starts at 50. Thus the first sensor reading has a shorter reaction time, but every subsequent value change has the same maximum reaction time of 190.

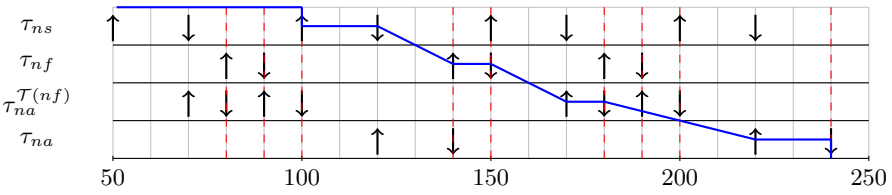


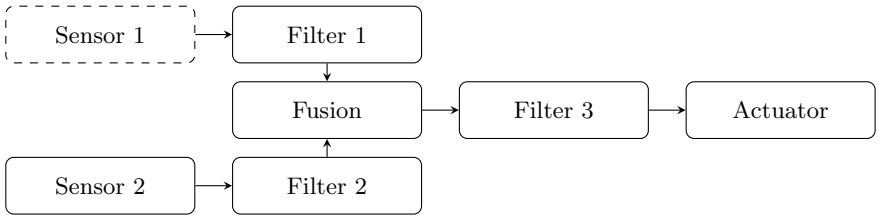
Fig. 9. Example trace for maximum reaction time for small example.

5 Validation

To validate our semantics, we have implemented the case-study ROS network from [14], where Teper et. al. provides a simulation-based measurement of the maximum end-to-end reaction time. It is a small network, connecting sensors with filters, representing processing of data, a fusion node which merges data and an actuator and the end of the processing chain. The network is depicted in

Component	Formal object
`Sensor 1`	$s_1 = (420, 10, \mathcal{T}(s_1), \mathbf{0})$
`Sensor 2`	$s_2 = (420, 20, \mathcal{T}(s_2), \mathbf{0})$
`Filter 1`	$f_1 = (\mathcal{T}(s_1), 10, \emptyset, \emptyset, \emptyset, \mathcal{T}(f_1), \mathbf{0})$
`Filter 2`	$f_2 = (\mathcal{T}(s_2), 20, \emptyset, \emptyset, \emptyset, \mathcal{T}(f_2), \mathbf{0})$
`Fusion`	$fs = (\mathcal{T}(f_1), 30, \{\mathcal{T}(\{_\in)\}, \{30\}, \emptyset, \mathcal{T}(fs), \mathbf{0})$
`Filter 3`	$f_3 = (\mathcal{T}(fs), 30, \emptyset, \emptyset, \emptyset, \mathcal{T}(f_3), \mathbf{0})$
`Actuator`	$a = (840, 30, \{\mathcal{T}(f_3)\}, \{30\}, \emptyset, \mathbf{0}, \mathbf{0})$

Table 2. Formalized validation case



Component	Parameter	Value	Component	Parameter	Value
`Sensor 1`	*WCET*	10	`Sensor 1`	*period*	420
`Sensor 2`	*WCET*	20	`Sensor 2`	*period*	420
`Filter 1`	*WCET*	10	`Filter 2`	*WCET*	20
`Fusion Sub1`	*WCET*	30	`Fusion Sub2`	*WCET*	30
`Filter 3`	*WCET*	30			
`Actuator`	*WCET*	30	`Actuator`	*period*	840
`Actuator Sub1`	*WCET*	30			

Fig. 10. Case-study network and parameter values for subscriber/timer-scenario from [14].

Figure 10. The case-study varies the type of the fusion node and the actuator node. Both of the nodes can either be subscription-based or timer-based resulting in four combinations. Each node is formalized as shown in Table 2.

In Table 3 the computed maximum reaction times are shown for both the simulation-based measurement of Teper et. al. and the model checking-based computation from UPPAAL. Note that we only consider the under-utilized case of [14]. For the over-utilized system we have different results, as the solution from Teper et. al. throws away messages when buffers are overflown.

6 Non-Deterministic Modeling

In this section we extend our semantics where we allow for non-determinism: we allow for tasks to execute in faster than their worst-case execution time, and data-generators to only generate data with a certain probability.

Case	Fusion	Actuator	Simulation	Model-Checking
1	Subscriber	Subscriber	540	540
2	Subscriber	Timer	1320	1320
3	Timer	Subscriber	1470	1470
4	Timer	Timer	2490	2490

Table 3. Comparison of simulation and model-checking approaches.

6.1 Non-deterministic execution times

In the base semantics, the host would always execute jobs for the duration of their WCET. However, in general a jobs execution time t is constraint by $BCET \leq t \leq WCET$, where BCET is the best-case execution time. We modify the host accordingly, by changing the guard accordingly: $x = WCET[job]$ is replaced by $BCET[job] \leq x \wedge x \leq WCET$. Note that this require the introduction of a BCET-constant for each task.[3]

6.2 Probabilistic Data-Generator

The sensor as presented in Section. 4 would always read a new value each *period*. However, we now introduce a *probabilistic data-generator* which only generates a value each *period* with a probability p, i.e., each p with chance *prob* a new value is generated and the job scheduled. If $prob = 100$, the behaviour is identical to a regular data-generator. The probabilistic data-generator is shown in Figure. 11. Also, location l_{choose} and l_{fire} are marked committed.

Monitored Probabilistic Data-generator : A monitored probabilistic data-generator is identical to a probabilistic data-generator with the difference that the edge from l_{fire} to l_{wait} is replaced by:

$$l_{fire} \xrightarrow{start\_monitor^!,\emptyset,\emptyset,queue\_job(id,PAYLOAD)} l_{wait}\}$$

Intuitively, this means that when a probabilistic monitored data-generator publishes data, a monitored is requested to be started and a payload value is queued.

6.3 Statistical Model Checking

When a probabilistic model is chosen, it might no longer be interesting to establish maximum upper bounds, as in many cases these will be the same as for the non-probabilistic case (i.e., the worst case). However, the worst case could be very rare, and thus acceptable. Using statistical model checking (SMC) it is possible to find an upper bound which is only violated with a certain (low)

[3] We assume for convenience that in this report $BCET[job] = WCET[job]/2$.

$$L = \{l_{wait}, l_{choose}, l_{fire}\}, \quad \ell_0 = l_{wait}, \quad C = \{x\}, A = \{new\_job^!, new\_job^?\},$$
$$V = \emptyset\}, I = \{l_{wait} \mapsto x \leq p\},$$
$$E = \{l_{wait} \xrightarrow{new\_job^!, x=p, x, \emptyset} l_{choose}, l_{wait} \xrightarrow{new\_job^?, x=p, x, \emptyset} l_{choose},$$
$$l_{choose} \xrightarrow{\tau, ?prob, x, \emptyset} l_{fire}, l_{choose} \xrightarrow{\tau, ?100-prob, x, \emptyset} l_{wait},$$
$$l_{fire} \xrightarrow{\tau, \emptyset, \emptyset, queue\_job(id, EMPTY)} l_{wait}\}$$

Fig. 11. Template of a Probabilistic Data-Generator.

chance. For example, the following UPPAAL SMC query yields the probability that the reaction time is observed to be more than t within u time-steps:

$$\texttt{Pr}[\leq u] \quad (\diamond((monitor.measure \wedge monitor.x[last\_monitor] \geq t)))$$

The answer to such a query can be $p \leq 0.0499$ with 95 % CI. This means that the probability that the bound of t is violated within u is less than 5 %, and that if we would redo the test, the probability that we would yield the same answer is 95 %. Depending on the application these probabilities could be sufficiently tight to be acceptable.

‘

7 Related Work

Closest to our work is the TA-based approach, proposed by Halder et. al. [7], which models and verifies safety and liveness properties of ROS applications, focusing on the communication between nodes, and considering queue sizes and internal timeouts. While the work is carried out at a lower level of abstraction than ours, the authors consider only a publish-subscribe scenario and do not propose methods to enable both end-to-end reaction time verification in deterministic settings, as well as stochastic analysis of reaction time under probabilistic loads. Dust et. al. [5] propose a pattern-based modeling and UPPAAL-based verification of latencies and buffer overflow in distributed robotic systems, including all versions of the single-threaded executor in ROS 2, yet the authors do not consider processing chains in their verification, focusing on the node behavior only. Lin et. al. [10] propose formal models for the real-time publish-subscribe protocol using UPPAAL and analyze the protocol's behavior by simulation in Simulink/Stateflow, however the authors do not validate the formal models against the simulation results, as we show in this report.

The Coq-based verification of ROS implementations has been the focus of several works, out of which that of Cowley and Taylor verifies robotic behaviour using linear logic embedding in Coq [4], and that of Anand and Knepper proposes ROSCoq, a framework for developing certified Coq programs for robots, where subsystems communicate using messages [2]. Neither of these works focuses on verifying end-to-end reaction time of job chains, as they analyze implementation levels instead.

8 Conclusions and Future Work

In this technical report we present a formalization of ROS semantics in UPPAAL timed automata. We began with a deterministic template-based scheme validated against previous work. Afterwards we extended it with non-determinism by allowing variable run-times as well as probabilistic data generation. We demonstrate how the UPPAAL tool can be used with regular and statistical model checking to establish maximum upper bounds of the formalized ROS systems.

Acknowledgements. We acknowledge the support of the Swedish Knowledge Foundation via the synergy ACICS – Assured Cloud Platforms for Industrial Cyber-Physical Systems, grant nr. 20190038, and via the profile DPAC - Dependable Platform for Autonomous Systems and Control, grant nr. 20150022.

References

1. Rajeev Alur. Timed automata. In Nicolas Halbwachs and Doron Peled, editors, *Computer Aided Verification*, pages 8–22, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.
2. Abhishek Anand and Ross Knepper. Roscoq: Robots powered by constructive reals. In *Interactive Theorem Proving*, pages 34–50, Cham, 2015. Springer International Publishing.
3. Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.
4. Anthony Cowley and Camillo J. Taylor. Towards language-based verification of robot behaviors. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4776–4782, 2011.
5. Lukas Dust, Rong Gu, Cristina Seceleanu, Mikael Ekström, and Saad Mubeen. Pattern-based verification of ros 2 nodes using uppaal. In *Formal Methods for Industrial Critical Systems (FMICS)*, pages 57–75. Springer Cham, 2023.
6. Endre Erős, Martin Dahl, Kristofer Bengtsson, Atieh Hanna, and Petter Falkman. A ros2 based communication architecture for control in collaborative and intelligent automation systems. *Procedia Manufacturing*, 38:349–357, 2019. 29th International Conference on Flexible Automation and Intelligent Manufacturing (FAIM 2019), June 24-28, 2019, Limerick, Ireland.
7. Raju Halder, José Proença, Nuno Macedo, and André Santos. Formal verification of ros-based robotic applications using timed-automata. In *2017 IEEE/ACM 5th International FME Workshop on Formal Methods in Software Engineering (FormaliSE)*, pages 44–50, 2017.
8. Kim G Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International journal on software tools for technology transfer*, 1(1-2):134–152, 1997.
9. Axel Legay, Anna Lukina, Louis Marie Traonouez, Junxing Yang, Scott A. Smolka, and Radu Grosu. Statistical model checking. In Bernhard Steffen and Gerhard Woeginger, editors, *Computing and Software Science: State of the Art and Perspectives*, pages 478–504. Springer International Publishing, Cham, 2019.
10. Qian-Qian Lin, Shu-Ling Wang, Bo-Hua Zhan, and Bin Gu. Modelling and Verification of Real-Time Publish and Subscribe Protocol Using Uppaal and Simulink/Stateflow. *Journal of Computer Science and Technology*, 35:1324–1342, 2020.

11. Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66), 2022.
12. Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, et al. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.
13. Michael Reke, Daniel Peter, Joschua Schulte-Tigges, Stefan Schiffer, Alexander Ferrein, Thomas Walter, and Dominik Matheis. A self-driving car architecture in ros2. In *2020 Int. SAUPEC/RobMech/PRASA Conference*, pages 1–6, 2020.
14. Harun Teper, Mario Günzel, Niklas Ueter, Georg von der Brüggen, and Jian-Jia Chen. End-to-end timing analysis in ros2. In *2022 IEEE Real-Time Systems Symposium (RTSS)*, pages 53–65, 2022.

A UPPAAL Functions

```
bool waiting_jobs() {
    int i;
    for (i=0; i<C; i++) {
        if (QUEUES_COUNT[i] > 0) return true;
    }
    return false;
}

void queue_job(int task, int data) {
    QUEUES[task][QUEUES_COUNT[task]] = data;
    QUEUES_COUNT[task] += 1;
}


// Right-most job first, i.e., greatest priority first
void schedule() {
    int i, j, tmp_id, tmp_data;

    // Sort first by id
    for (i=0; i<HOST_JOBS_COUNT; i++) {
        for (j=0; j<HOST_JOBS_COUNT-1; j++) {
            if (PRIO[HOST_JOBS[j][0]] > PRIO[HOST_JOBS[j
    +1][0]]) {
                tmp_id = HOST_JOBS[j][0];
                tmp_data = HOST_JOBS[j][1];
                HOST_JOBS[j][0] = HOST_JOBS[j+1][0];
                HOST_JOBS[j][1] = HOST_JOBS[j+1][1];
                HOST_JOBS[j+1][0] = tmp_id;
                HOST_JOBS[j+1][1] = tmp_data;
            }
        }
    }
}

int dequeue(int task) {
    int i, tmp;
    assert(QUEUES_COUNT[task] > 0);
    tmp = QUEUES[task][0];
    for (i=0; i<BUF_SIZE-1; i++)
        QUEUES[task][i] = QUEUES[task][i+1];
    QUEUES[task][BUF_SIZE-1] = 0;
    QUEUES_COUNT[task] -= 1;
    return tmp;
}

void take_jobs() {
    int i, j;
```

```
47
48      assert(HOST_JOBS_COUNT == 0);    // Host jobs should be
        zero here
49      for (i=0; i<C; i++) {
50          if (QUEUES_COUNT[i] > 0) {
51              j = dequeue(i);
52              HOST_JOBS[HOST_JOBS_COUNT][0] = i;
53              HOST_JOBS[HOST_JOBS_COUNT][1] = j;
54              HOST_JOBS_COUNT += 1;
55          }
56      }
57      schedule();
58  }
59
60  int next_job_idx() {
61      HOST_JOBS_COUNT--;
62      return HOST_JOBS_COUNT;
63  }
64
65  // If value is negative, we don't pick from queue, but from
        node
66  int get_data(int value) {
67      if (value < 0) {
68          return value;
69      } else {
70          return DATA[value]; // Do we need to remove read
        values?
71      }
72  }
73
74  void assign_monitor() {
75      if (lm == -1) {
76          lm = nm;
77      }
78      monitor_status[nm] = MONITOR_SENT;
79      monitor_payload[nm] = PAYLOAD;
80      PAYLOAD = PAYLOAD - 1;
81      if (PAYLOAD < MIN_PAYLOADS) {
82          PAYLOAD = FIRST_PAYLOAD;
83      }
84      nm = (nm + 1) % MONITORS;
85  }
86
87
88
89  // When freeing up, we free up all monitors incl. those whose
        data got thrown away.
90  void free_monitors() {
91      int i;
```

```
92      // We could get an old value, in that case it is not in
        the monitored payloads
93      bool old_value = true;
94      for (i = 0; i < MONITORS; i++) {
95          if (monitor_payload[i] == pd)
96              old_value = false;
97      }
98
99      // If it is an old value, we can just ignore it as it has
        already been seen once
100     if (old_value)
101       return;
102
103     // Free previously used monitors
104     while (monitor_payload[lm] != pd) {
105         monitor_status[lm] = MONITOR_FREE;
106         monitor_payload[lm] = EMPTY;
107         lm = (lm + 1) % MONITORS;
108     }
109
110     // Also free the one just handled.
111     monitor_status[lm] = MONITOR_FREE;
112     LAST_PAYLOAD = monitor_payload[lm];
113     monitor_payload[lm] = EMPTY;
114     lm = (lm + 1) % MONITORS;
115 }
```

B Graphical Representation of UPPAAL TA

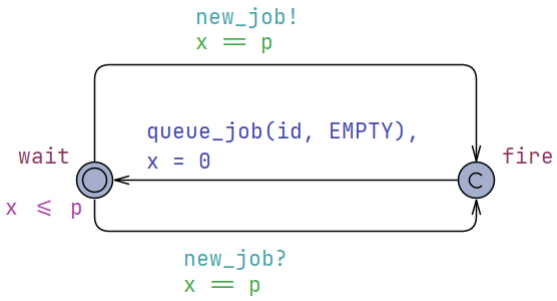


Fig. 12. UPPAAL figure of data-generator.

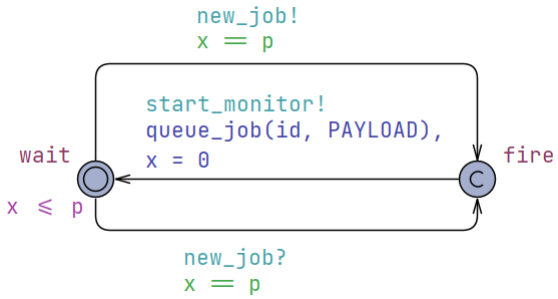


Fig. 13. UPPAAL figure of monitored data-generator.

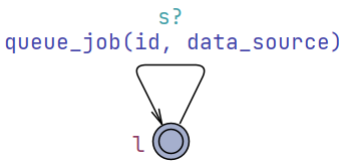


Fig. 14. UPPAAL graphical representation of subscriber.

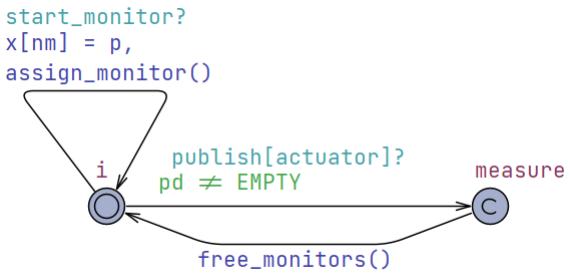


Fig. 15. UPPAAL graphical representation of Monitor.

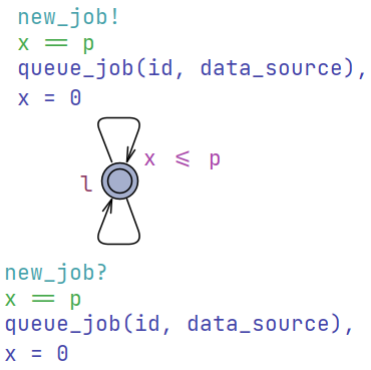


Fig. 16. Template for timer.

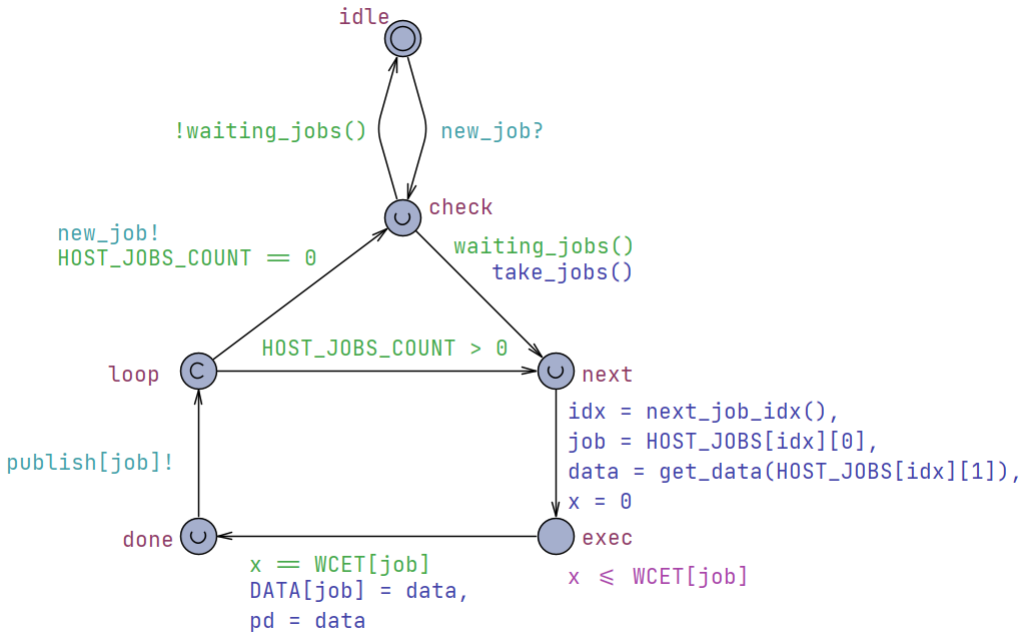


Fig. 17. Template for host.

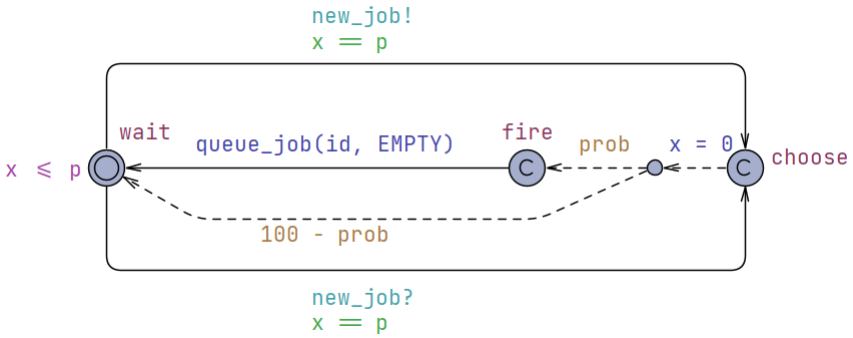


Fig. 18. Template for probabilistic data-generator.

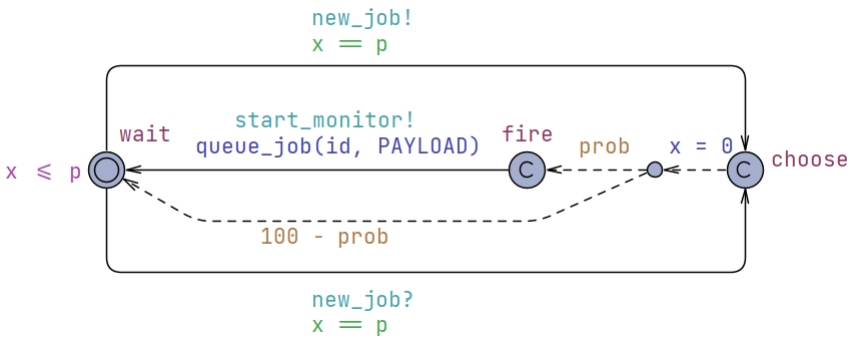


Fig. 19. Template for monitored probabilistic data-generator.