Verifying ROS-based Applications Using Timed and Stochastic Timed Automata

Peter Backeman and Cristina Seceleanu

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 ROS 2 offering real-time capabilities. However, determining the maximum reaction time within a ROS-based application is intricate due to complex variable processing and scheduling, especially with periodic and event-triggered tasks. In this paper, we propose a model of ROS-based designs with timed automata semantics, facilitating exhaustive real-time model checking of system behavior. We extend this model to stochastic timed automata, thus incorporating non-deterministic execution time and probabilistic loads, employing statistical model checking for scalability and accuracy. We compare against previous work to confirm the validity of our approach, and show its applicability on a real-world robotic system example.

1 Introduction

This paper is dedicated to Professor Marjan Sirjani, to honor her impactful contributions to the field of formal methods, which have advanced the reliability and correctness of complex systems profoundly, including robotic systems. Marjan's work embodies a vision where robotic systems not only meet functional requirements, but also achieve high standards of dependability, addressing the nuanced interplay of concurrency, real-time constraints, and emergent behaviors [19].

According to this vision, we also consider robotic systems to be subject to real-time constraints that should be obeyed, for example, a system might be required to react to a certain sensor input within a time bound (e.g., door opening when the light sensor is activated). Ensuring such requirements already at early design stages is beneficial and can be done through formal methods. Model checking is an approach that can help establish both liveness and safety properties subject to timing constraints, by employing timed automata as the modeling formalism [4]. When designing a robotic system there are many aspects to take into consideration, e.g., computation and communication issues must be decided upon and addressed. To address this, one can use a pre-existing solution that already solves many challenges; the Robot Operating System (ROS) provides

such a framework for setting up nodes and their communication. However, establishing properties of a ROS-based application, e.g., finding an upper bound for the maximum reaction time, is complex as it is affected by the run-time of the involved tasks, respectively, and how the latter are scheduled.

Previous work has helped to establish bounds on reaction times in ROS, through simulation [18]. However, such an approach is hard to extend with non-deterministic behavior, as it is based on simulating one scenario at a time. For example, while task run-time is usually limited by a worst-case execution time (WCET), in practice a task can finish early (and tasks finishing earlier than their WCET can counter-intuitively result in a longer reaction time). Using a model-checking approach, it is possible to investigate multiple traces simultaneously (i.e., corresponding to different task execution times), thus enabling efficient checking of properties in non-deterministic systems. With this in mind, we address the following research question: *How can model checking help analyze end-to-end reaction times in ROS-based applications with non-deterministic properties?*

In this paper, we present a model of ROS-based designs and assign semantics to allow model checking with respect to real-time behavior to help establish such properties, in particular on end-to-end reaction time bounds. We assign semantics in the form of *timed automata* (TA) [2] templates, yielding a precise definition of the underlying behavior of ROS scheduling. We begin with basic semantics and validate it against previous work [18], after which we extend the semantics with non-deterministic execution times and probabilistic loads. For scalability and richness of modeling purposes, we employ the *statistical model checking* (SMC) [13] technique, where properties are guaranteed to a specific degree of confidence. We model and check systems in this paper via the UPPAAL SMC [12] tool. In sum, we present the following contributions:

- Introduction of a pattern-based TA semantics of ROS designs, covering both deterministic and probabilistic running times and loads.
- Validation of our base semantics against previous simulation-based work that uses analytic real-time scheduling theory.
- Application of UPPAAL/UPPAAL SMC to find maximum reaction times in an industrial example, demonstrating the approach.[1]

The paper is structured as follows: after the preliminaries in Section 2, we present our formalization of ROS-based application behavior in Section 3, followed by our TA semantics in Section 4 and its validation in Section 5. Afterwards, we extend the semantics in several directions in Section 6, and demonstrate with an industrial example in Section 7. Finally, related work and comparison to our approach are presented in Section 8, and our conclusions in Section 9.

[1] Source code is available at: `https://github.com/ptrbman/ros2-modeling/`

2 Preliminaries

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

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 [16]. The operating system's version ROS 1 underwent a major revision and became ROS 2 [15], bringing many improvements, most notably the Data Distribution Service (DDS) support. DDS acts as middleware 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 ROS 1.

The ROS 2 platform has already been used in designing the communication architecture of collaborative and intelligent automation systems [10], or of self-driving cars that require safe and reliable real-time behavior [17]. 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 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. In this paper, we consider a ROS architecture to consist of *nodes* and *topics*. Communication is done through two paradigms, publish/subscribe and local variables. Whenever a node wishes to communicate data, it can *publish* it onto a topic, or *write* it to a local variable. A node can read data either through *receiving* it from a *subscribed* topic, or *reading* it from a local variable.

Scheduling A node is a component that has one or more *tasks* (i.e., callbacks) that can be scheduled for execution. When a task is scheduled, a *job* is instantiated and added to 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 as follows: at each *polling point*, i.e., beginning of a *processing window*, it picks one job from each task (which has queued job) and schedules them according to priority (with timers always having higher priority than subscribers). Next, each of the tasks are executed in order, 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. In this paper we consider a ROS system with only one host, that is, all nodes are processed on the same CPU.

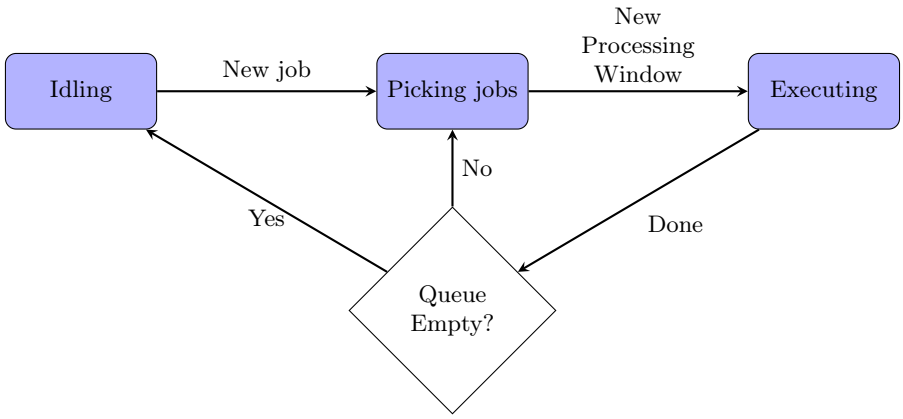


Fig. 1: Scheduler in ROS.

Reaction Time In this paper, we are interested in the reaction time of a ROS application. By this, we mean the time that passes between a job (e.g., corresponding to a sensor read) being released (i.e., when the job is instantiated) and the reaction of the system's actuator (i.e., when the actuator job is executed). We wish to establish an upper bound, that is, to find the maximum reaction time from a particular sensor being able to observe some data, until the actuator finishes executing, according to the scheduling policy described above.

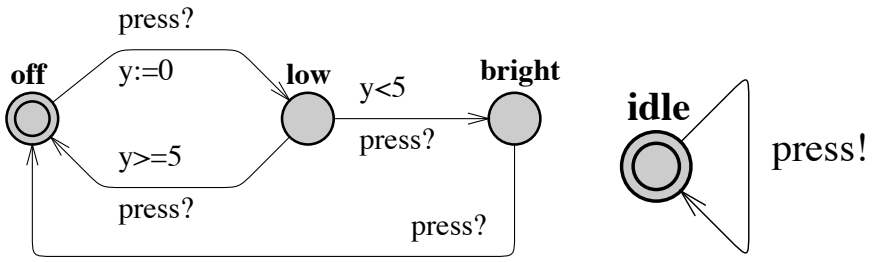
2.2 Timed Automata and UPPAAL

Formal Syntax. A Timed Automaton (TA), as used in the model-checker UPPAAL [12], 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 a data variable update action. Initially, all clocks are set to 0, and the update action assigns 0 to 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.

Example. For illustration purposes, in Figure 2, we show a simple TA model of a lamp 2(a), and its user 2(b). The lamp has three locations: `off`, `low`, and `bright`. If the user presses a button, modeled by the synchronization variable `press!`, then the lamp is turned on, that is, the lamp TA synchronizes with the user TA via `press?`, and changes location from `off` to `low`. If the user presses the button again, the lamp is turned off, and the lamp TA moves back to location `off`. However, if the user is fast and presses the button twice, rapidly, the lamp is turned on and becomes bright, modeled by the lamp TA moving to location `bright`. The user can press the button arbitrarily at any time or not press the button at all. The clock y of the lamp is used to detect if the user is fast, via the guard $(y < 5)$ over the edge between locations `low` and `bright`, or slow, via the guard $(y >= 5)$ over the edge between locations `low` and `off`.



(a) Lamp (b) User

Fig. 2: A Simple TA model of a lamp and its user

Operational Semantics. The semantics of TA [5] is defined as 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 is the clock valuation in location l. The initial state is denoted by (l_0, u_0), where all clocks are initialized to 0: $\forall x \in C,\ u_0(x) = 0$. 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 requirements (queries in UPPAAL) 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. In this paper, we verify (T)CTL queries of the following kinds (p is a state property):

- Reachability: $E \Diamond p$ - The requirement evaluates to true if there exists a path where p eventually holds.
- Invariance: $A \Box p$ - The requirement evaluates to true if (and only if) every reachable state satisfies p, in other words, for all paths p always holds.

UPPAAL is also capable of handling *statistical model checking* (SMC) [13], 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[2]), 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. UPPAAL SMC uses a probabilistic extension of *weighted metric temporal logic* (WMTL) [6] to provide qualitative analysis, that is, *hypothesis testing*, and *probability comparison*, as well as quantitative analysis, by *probability evaluation*: calculate the probability $Pr[<= bound](\star_{x \leq C} \phi)$ to reach a state ϕ within time cost $x \leq C$ for some network of stochastic timed automata, where $\star$ stands for either *future* ($\Diamond$) or *globally* ($\Box$) temporal operator, and $[<= bound]$ denotes the time bound of the executions. In this paper, we focus on probability evaluation only.

3 Formalization of ROS-based Systems

In this section, we provide a formalization of constrained ROS-based systems. In this work, we model communication via publish/subscribe for inter-node communication and read/write for intra/node communication and the scheduling of tasks on nodes. We assume that each node can be *subscribed* to zero or more nodes, *read* from zero or one variable, *publish* to zero or one topic, and *write* to zero or one variable. We define three kinds of nodes: timer nodes, subscription nodes and data-generator nodes. A *timer node* is triggered at 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 node that has no subscriptions or reads from any variable. For simplicity, we assume that there is only one publishing node per topic.

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

3.1 Nodes

We define a set of nodes $\mathcal{N}$, a set of topics $\mathcal{T}$ and a set of global variables $\mathcal{V}$. The set of global variables includes a special variable, pd, which is the most recently published data. We define each kind of node separately, as follows.

Definition 1. *A* timer node *is defined as:* $tn = TMR(p, d, wcet, S, St, t, rv, wv)$*, where:*

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

Intuitively, a timer node is activated each p period, with the first activation after a delay of d, (i.e., the second activation is at $p + d$, third at $2p + d$, etc.), creating a job of the main task. The node subscribes to the topics S and uses the data from the variable rv. Whenever a message is received on topic s_i, a job is created to process the retrieved value, with WCET st_i, writing its result to variable $\mathcal{V}(tn^i)$.

Definition 2. *A* subscriber node *is defined as:* $sn = SUB(s, wcet, S, St, t, rv, wv)$*, where:*

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

A subscriber node works like a timer node, except that it is triggered whenever a message is published on the triggering topic s, instead.

Definition 3. *A* data-generator node *is defined as:* $dn = DGEN(p, d, wcet, t, wv)$*, where:*

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

A data-generator works as a timer except that it has no subscriptions or read-variable.

Example 1. Consider the small system in Fig. 3 with four nodes. One sensor node publishes data to a filter node, which in turn publishes data to the actuator node. There is also an additional sensor node that publishes directly to the actuator. We are interested in measuring the reaction time from the second sensor node to the actuator, that is, to monitor data published by the second sensor and measure the time until it is processed by the actuator. Each component can be formalized (for particular values of WCET, etc.) as:

- first sensor, a data-generator node:
  $s_1 = DGEN(200, 0, 50, \mathcal{T}(s_1), \mathcal{V}(s_1))$,
- second sensor, a data-generator node:
  $s_2 = DGEN(200, 50, 30, \mathcal{T}(s_2), \mathcal{V}(s_2))$,
- filter, a subscriber node:
  $f = SUB(\mathcal{T}(s_1), 30, \emptyset, \emptyset, \mathcal{T}(f), pd, \mathcal{V}(f))$,
- actuator, a subscriber node:
  $a = SUB(\mathcal{T}(f), 10, \{\mathcal{T}(s_2)\}, \{10\}, \mathcal{T}(a), \mathcal{V}(a^1), \mathcal{V}(a))$.

Note that the read-variable for the actuator is $\mathcal{V}(a^1)$, indicating that the value to be used in published data is equal to the value retrieved from the subscription to the second sensor.

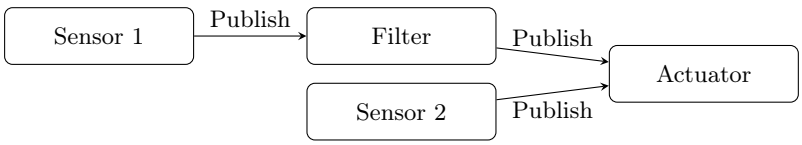


Fig. 3: Small ROS network

3.2 Tasks

Each node contains one or more tasks. All nodes have a *main task* that is responsible for combining all data, compute and publish/write the output. Additionally, every node has for each non-triggering subscription a task 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. For convenience, we introduce a unique topic and variable for each task τ denoted by $\mathcal{T}(\tau)$ and $\mathcal{V}(\tau)$, respectively.

Example 2. The system presented in the previous example contains four nodes. The sensors and filter nodes contain only one main task each: τ_{s_1}, τ_{s_2} triggered by timers and τ_f triggered by a subscription. The actuator node has one main task τ_a triggered by the filter node, and one subscription task τ_a^1 triggered by the second sensor node.

3.3 Job chains

To define the notion of reaction time precisely, we first present the notion of task chain, and (forward) job chains, respectively.

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

- *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 its 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 such that 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} *that reads the result from* j_i*, or*
  - j_{i+1} *is the unique earliest job of* τ_{i+1} *that receives the result from* j_i *(through subscription).*

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

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

Example 3. Consider again the small example network in Figure 3, now executed (according to semantics in next section) with scheduling as presented in Section 2.1. The resulting trace is shown in Figure 4. The job chain $J = \{\tau_{s_2}, \tau_a^1, , \tau_a\}$ has a reaction time of 80.

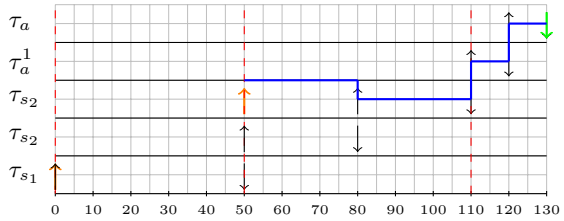


Fig. 4: Schedule for small ROS network example. Processing windows are separated by red dashed lines. An orange upwards arrow indicates the release of a job, a black upwards arrow the start of a job, and downwards indicates the end of a job. The blue line traces the reaction time.

In this paper, we focus on the case where there is only one host (i.e., executor), 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 *for a task-chain* $\mathfrak{T}$ *as* $max_{j \in J}(rt(j))$*, where* $\mathcal{J}$ *is the set of all job-chain instances of* $\mathfrak{T}$*.*

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. 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 that are instantiated depending on the nodes of the ROS application, and the task chain that should be monitored. 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 always publishes to its dedicated topic, and writes to its dedicated variable.

4.1 Constants, Variables, Functions and Channels

Table 1 shows a set of constants, variables, functions and channels. Some constants are used for easier reading, while some have a semantic impact. In particular, `BUF_SIZE` and `MONITORS` must be set sufficiently large. In the remainder of this paper we assume that this is the case. Note that `PRIO[C]`, `WCET[C]` must also be initialized with proper values.

4.2 TA Templates

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

Subscriber. A subscriber, $sn = SUB(s, wcet, S, St, t, rv, wv)$, is represented by an instantiated Subscriber TA template using three parameters: $task\_id$, s and $data\_source$. A suitable $task\_id$ is chosen, and while the triggering subscription s is initialized to the counter-part in sn, the $data\_source$ is set to rv. 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 the $task\_id$ topic (as we assume each task publishes to its own topic). The template is shown in Figure 5.

Moreover, for each subscribed topic $s_i \in S$, the Subscriber TA template is additionally instantiated using a suitable $task\_id$, $s = s_i$, and $data\_source$ equal to pd.

Constant	Description
`EMPTY`	Value for representing empty data.
`MONITORS`	Number of parallel monitors.
`FIRST_PAYLOAD`	Represents the value of the first monitored package.
`MIN_PAYLOAD`	Minimum value of payloads (to reduce state space).
`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.
`deterministic_host`	If true, jobs always takes *wcet* to execute.
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.
JOBS[C][2]	ID and data of scheduled jobs.
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.
relevant_payload	True if just published data is monitored.
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

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

Fig. 5: Template of a Subscriber

Timer. A timer node $tn = TMR(p, d, wcet, S, St, t, rv, wv)$ is represented by an instantiated Timer TA template using four parameters: $task\_id$, p, d, and $data\_source$. A suitable $task\_id$ is chosen, and while the period p and delay d are initialized to the counter-part in tn, $data\_source$ is set to rv. The clock is initialized as $x = p - d$, to ensure that the first activation happens after d. The timer is activated each *period* p. 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.

In the same vein as for the subscriber node, for each subscribed topic an additional Subscriber TA template is instantiated.

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

Fig. 6: Template of a Timer

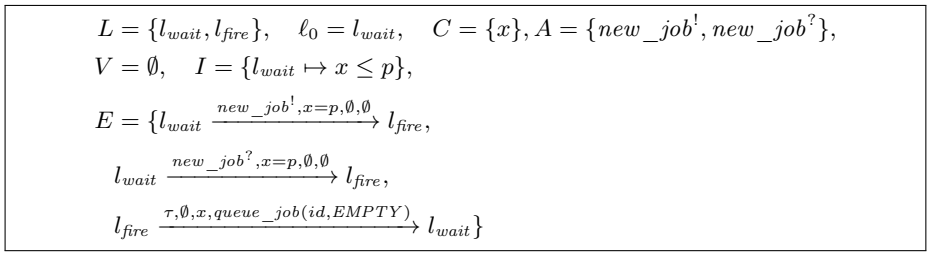
Data-Generator. The template for a data-generator, $dn = DGEN(p, d, wcet, t, wv)$, is represented by an instantiation of the Data-Generator TA template using three parameters: $task\_id$, p, and d. A suitable $task\_id$ is chosen, and the period p and delay d are initialized to their counter-part in tn. The clock is initialized as $x = p - d$, to ensure that the first activation occurs after d. The data-generator is responsible for generating a value each *period* p, then queuing a job of type τ_{dn}. The template is shown in Figure 7. In addition, 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 monitor is requested to be started and a payload value is queued. The first task of the task chain $\mathfrak{T}$ should be instantiated as this.

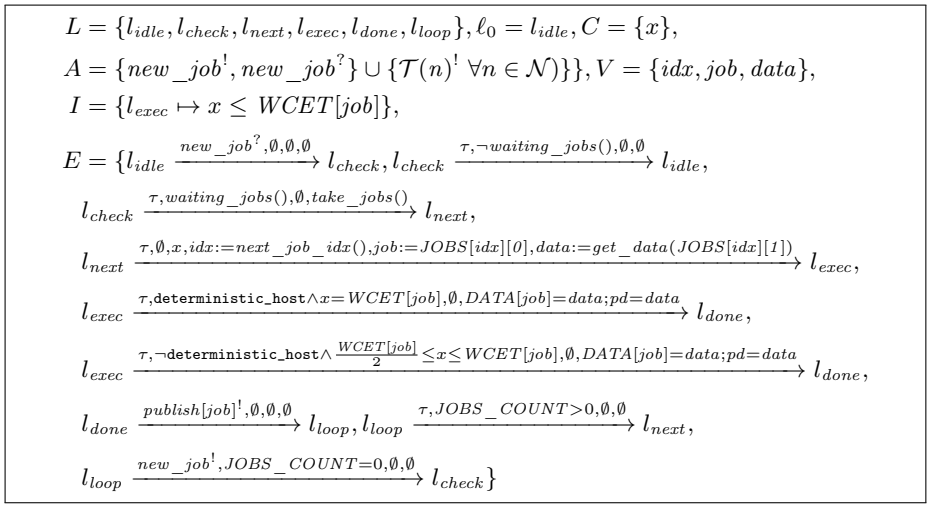
Host. A host is the most complicated template, but has no parameters. The host awaits waiting jobs, and when the latter are present, it picks (up to) one of each task, and schedules them according to their respective priority (through the *schedule* function, called by the $take_j\, obs$ function). It 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 it finishes



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

Fig. 7: Template of a Data-Generator

executing a job with the resulting data written to the associated variable. The template is shown in Figure 8. In addition, locations l_{check}, l_{next} and l_{done} are marked urgent, and l_{loop} is marked committed.



$$L = \{l_{idle}, l_{check}, l_{next}, l_{exec}, l_{done}, l_{loop}\}, \ell_0 = l_{idle}, C = \{x\},$$
$$A = \{new\_job^!, new\_job^?\} \cup \{\mathcal{T}(n)^! \ \forall n \in \mathcal{N})\}\}, V = \{idx, job, data\},$$
$$I = \{l_{exec} \mapsto x \leq WCET[job]\},$$
$$E = \{l_{idle} \xrightarrow{new\_job^?, \emptyset, \emptyset, \emptyset} l_{check}, l_{check} \xrightarrow{\tau, \neg waiting\_jobs(), \emptyset, \emptyset} l_{idle},$$
$$l_{check} \xrightarrow{\tau, waiting\_jobs(), \emptyset, take\_jobs()} l_{next},$$
$$l_{next} \xrightarrow{\tau, \emptyset, x, idx:=next\_job\_idx(), job:=JOBS[idx][0], data:=get\_data(JOBS[idx][1])} l_{exec},$$
$$l_{exec} \xrightarrow{\tau, \texttt{deterministic\_host} \wedge x = WCET[job], \emptyset, DATA[job]=data; pd=data} l_{done},$$
$$l_{exec} \xrightarrow{\tau, \neg \texttt{deterministic\_host} \wedge \frac{WCET[job]}{2} \leq x \leq WCET[job], \emptyset, DATA[job]=data; pd=data} l_{done},$$
$$l_{done} \xrightarrow{publish[job]^!, \emptyset, \emptyset, \emptyset} l_{loop}, l_{loop} \xrightarrow{\tau, JOBS\_COUNT>0, \emptyset, \emptyset} l_{next},$$
$$l_{loop} \xrightarrow{new\_job^!, JOBS\_COUNT=0, \emptyset, \emptyset} l_{check}\}$$

Fig. 8: Template of a Host

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 $l_{measure}$ is reached whenever the monitored actuator publishes data, so queries can be over this location to check worst-case reaction times. For compatibility with the validation case, we allow for a second parameter p, which can be set to the period of the data-generator of the first task in the chain. This allows the measured reaction-time to include the worst-case delay from an external event occurring (instead of its measuring). The template is shown in Figure 9. In this model, location $l_{measure}$ is marked committed.

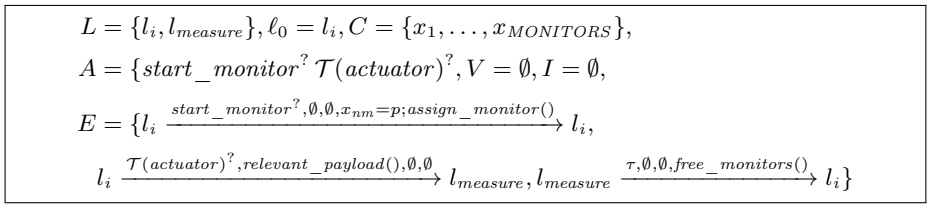


Fig. 9: Template of a Monitor

4.3 Instantiation

For a given ROS-based design, each component is instantiated with the corresponding template using suitable values for the parameters. In addition, each component is also 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 TA network will be deterministic in the sense that the clock values in the monitor automaton will always be the same whenever they are measured at the *measure* location.

Example 4. We instantiate the ROS-based application of Ex. 3 according to the above formula, and obtain five TA: a Data-Generator for Sensor 1 ($task\_id = 0, p = 150, d = 0$), a Subscriber for Filter ($task\_id = 1, s = \mathcal{T}(0), data\_source = pd$), a Monitored Data-Generator for Sensor 2 ($task\_id = 2, p = 150, d = 50$), two Subscribers for the Actuator (($task\_id = 2, s = \mathcal{T}(1), data\_source = pd$) and ($task\_id = 3, s = \mathcal{T}(1), data\_source = pd$)), a Monitor ($actuator = 0, p = 0$), and a Host.

4.4 Queries

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

$$A\,\square\, monitor.measure \rightarrow monitor.x[last\_monitor] \leq t$$

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 t:

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

In order to find an upper bound, the above query is used with $t = 0$. 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 of a trace for a given bound.

Example 5. If we use UPPAAL to query for an upper bound for Example 1, we can the extract trace in Figure 4. This shows that the figure indeed demonstrates (an instance of) the worst case.

5 Validation

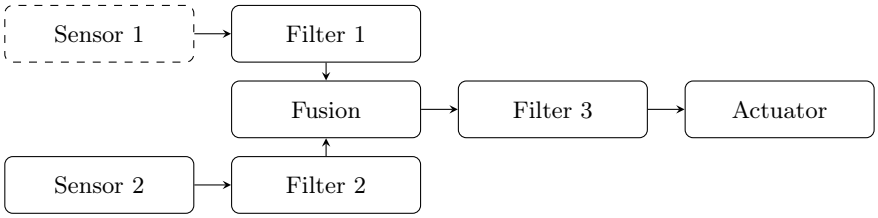
To validate our semantics, we have implemented the case-study ROS system from literature [18], where Teper et. al. provide a simulation-based measurement of the maximum end-to-end reaction time. It is a system that consists of: (i) sensors with filters, representing processing of data, (ii) a fusion node, which merges data, and (iii) an actuator at the end of the processing chain. The network is depicted in Figure 10. The case-study varies the type of the fusion node and the actuator node. Both nodes can either be subscription-based or timer-based, resulting in four combinations, respectively.

For a specific case (Fusion subscription/Actuator timer), each node is formalized as shown in Table 2 (we do not show the other three cases). Note that in our formalization, we must pick a specific task-chain to observe, in this case from Sensor 1 to the Actuator (it can be changed by changing the read-variable of the Fusion node).

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 presented in related work [18]. For the over-utilized system we have different results, as the solution from Teper et. al. throws away messages when buffers are overflown. The results show that our semantics seems to agree with the one presented by Teper et. al. (at least on the four cases tested).

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

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 system and parameter values for subscriber/timer-scenario from literature [18]. Fusion Sub1 is the triggering subscription task of the fusion node, whereas Fusion Sub2 is the first non-triggering subscription task of the Fusion node.

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

Table 3: Comparison of the simulation approach [18] and (our) model-checking approach. Results indicate the computed maximum end-to-end reaction time in milliseconds.

6 Modeling Non-Determinism

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

6.1 Non-deterministic Execution Time

In the base semantics, the host would always execute jobs for the duration of their respective WCET. However, in general, a job's execution time t is constrained by $BCET \leq t \leq WCET$, where BCET is the best-case execution time. In this extended semantics, we modify the host accordingly, by changing the guard as

follows: $x = WCET[job]$ is replaced by $BCET[job] \leq x \wedge x \leq WCET[job]$. Note that this requires the introduction of a BCET-constant for each task.[3]

Example 6. We revisit the small system introduced in Ex. 1. If we allow for non-deterministic running times (that is, a job's running time is in the interval $[\frac{wcet}{2}, wcet]$), the maximum reaction time is actually increased, demonstrated by the trace in Fig. 11. The reaction time depicted in the graph is $280 - 50 = 230$, greater than 80, the original bound when only worst-case execution times are considered. The reason for the increase in reaction time is due to the filter task being triggered before the second sensor is scheduled (which cannot happen if the first sensor task executes according to its WCET).

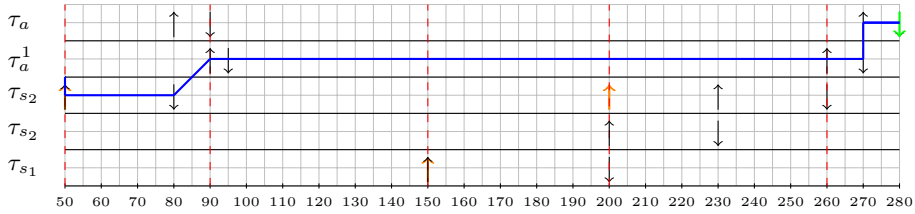


Fig. 11: Schedule for small ROS-based system example with non-deterministic reaction times. Processing windows are separated by red dashed lines. An orange upward arrow indicates the release of a job, a black upward arrow the start of a job, and a black downward one indicates the end of a job. The blue line traces the reaction time.

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. In other words, for each period p, there is a *prob* chance that a new value is generated and thus a job scheduled, in the other cases nothing happens. If $prob = 100$, the behavior is identical to a regular data-generator. The probabilistic data-generator is shown in Figure. 12. Also, location l_{choose} and l_{fire} are marked committed.

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

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

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

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

Fig. 12: Template of a Probabilistic Data-Generator

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 with maximum load). 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) 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:

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

The answer to such a query can be $p \leq 0.05$ 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 Industrial Example: Camera-Guided Robots

In this section, we present an example inspired from an industrial use case. To assist in production, a factory uses autonomous transport robots to transfer tools and parts as necessary. The robots are guided by a central control system that observes the environment through cameras. The cameras are equipped with a local processor that ensures that pictures are only sent if they changed since last time, that is, if the image is static no data is sent. During production humans can walk around in the same areas as the autonomous robots, and it is important that a collision is avoided by stopping the robot if someone would walk in front of it. Hence, it is crucial to ensure a low maximum reaction time from observing an obstacle through a camera, to sending a stop signal to the robot.

We model each camera as a separate data-generator, each being forwarded to a separate object detection, modeled as a subscriber. All object detection is fused in the fusion timer node. Finally, the managing and actuation is a subscriber node. The resulting ROS design is shown in Figure 13.

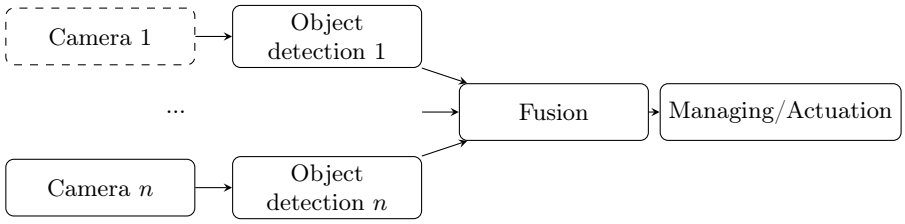


Fig. 13: Industrial example modeled as a ROS Design

We consider configurations with a various amount of cameras and instantiate the suitable templates in accordance with Section 4 with parameters and constant values shown in Table 4. Since cameras might not have new information to send, we model them using probabilistic data-generators. For simplicity, we let all cameras have the same probability of being activated at each instant, and we consider different combinations of number of cameras and the probability that a camera will be activated (load). All experiments are run on a 11th Gen i5 @ 2.40 GHz with 16.0 GB of RAM. Run-times are presented for each query and can be seen to mostly be around one second or less.

Component	Parameter	Value	Component	Parameter	Value
`Camera`	*WCET*	20	`Camera`	*period*	1000
`Object Detection`	*WCET*	50	`Fusion sub`	*WCET*	10
`Fusion Timer`	*WCET*	90	`Fusion Timer`	*period*	500
`Managing/Actuation`	*WCET*	50			

Table 4: Parameter values for the case-study system

We use the queries from Section 6.3, to establish if within 10,000 time steps there is less than a five percent chance of the reaction time exceeding 850 with a 95 % confidence interval. The results for the different scenarios are shown in Table 5. From the table, we can see that having more than five cameras already risks overloading the system in such a way as to violate a deadline of 850 time units. Note that since we do not distinguish the cameras from each other, the probabilities that we estimate are for the deadline violation of one camera. Since in most cases, for more than six cameras the deadline requirement is violated, in the following, we only consider cases from one to six cameras. The results in Table 5 are from the perspective on the data from the first camera, and when monitoring other cameras the results remain the same.

Next, we consider what happens if we extend the time period tenfold. Thus we consider a period of 100,000 time steps and with same reaction time (850), chance (5%) and confidence interval (95%). The results can be seen in Table 6. As expected, the acceptable loads are reduced, since running the system longer increases the chances of bad scenarios to occur: now only two cameras are safe

#Cams	Load	≤ 850	Time	#Cams	Load	≤ 850	Time	#Cams	Load	≤ 850	Time
1	25%	Yes	0.18	2	25%	Yes	0.19	3	25%	Yes	0.20
1	50%	Yes	0.16	2	50%	Yes	0.19	3	50%	Yes	0.21
1	75%	Yes	0.18	2	75%	Yes	0.20	3	75%	Yes	0.23
1	100%	Yes	0.17	2	100%	Yes	0.23	3	100%	Yes	0.25
4	25%	Yes	0.22	5	25%	Yes	0.22	6	25%	Yes	0.23
4	50%	Yes	0.23	5	50%	Yes	0.24	6	50%	No	0.18
4	75%	Yes	0.23	5	75%	Yes	0.27	6	75%	No	0.13
4	100%	Yes	0.29	5	100%	Yes	0.31	6	100%	No	0.13
#Cams	Load	≤ 850	Time	#Cams	Load	≤ 850	Time	#Cams	Load	≤ 850	Time
7	25%	Yes	0.29	8	25%	Yes	0.34	9	25%	No	0.30
7	50%	No	0.14	8	50%	No	0.13	9	50%	No	0.14
7	75%	No	0.12	8	75%	No	0.14	9	75%	No	0.12
7	100%	No	0.13	8	100%	No	0.12	9	100%	No	0.14
10	25%	No	0.22	11	25%	No	0.19	12	25%	No	0.15
10	50%	No	0.13	11	50%	No	0.14	12	50%	No	0.13
10	75%	No	0.14	11	75%	No	0.14	12	75%	No	0.14
10	100%	No	0.12	11	100%	No	0.13	12	100%	No	0.14

Table 5: Results from industrial example. A Yes in column ≤ 850 means that there is less than 5 % chance that the system violates the deadline (within the first 10000 time steps) under that configuration.

under a 75 % load, and three for a 50 % load. Moreover, the fact that the analysis time is roughly five times longer is also noteworthy.

#Cams	Load	≤ 850	Time	#Cams	Load	≤ 850	Time	#Cams	Load	≤ 850	Time
1	25%	Yes	0.68	2	25%	Yes	0.82	3	25%	Yes	0.93
1	50%	Yes	0.77	2	50%	Yes	0.95	3	50%	Yes	1.15
1	75%	Yes	0.81	2	75%	Yes	1.09	3	75%	No	1.38
1	100%	Yes	0.91	2	100%	No	1.20	3	100%	No	1.55
4	25%	Yes	1.05	5	25%	No	1.22	6	25%	No	3.04
4	50%	No	1.31	5	50%	No	1.58	6	50%	No	0.19
4	75%	No	1.59	5	75%	No	1.85	6	75%	No	0.14
4	100%	No	1.83	5	100%	No	2.19	6	100%	No	0.13

Table 6: Use case monitoring 1, with timer-based fusion (period of 500) analyzing 100000 time steps.

We can also use modeling to explore alternatives. As an example, we change the fusion node from a timer-based node to a subscription-based one, triggering whenever the first camera produces data (remember, a subscription node is triggered by only one topic), with a WCET of 90. Now there is a substantial difference between the end-to-end reaction times from different cameras. Since the first camera triggers the fusion, its reaction time will be very short. However,

if we look at camera three, the performance is very poor, as shown in Table 7, indicating that not even a scenario with two cameras is safe. If the load is 100 %, the scenario *is* safe, since that boils down to a timer-based fusion with a period equal to the period of the first camera.

#Cams	Load	≤ 850	Time	#Cams	Load	≤ 850	Time	#Cams	Load	≤ 850	Time
1	25%	n/a	0.00	2	25%	No	0.11	3	25%	No	0.13
1	50%	n/a	0.00	2	50%	No	0.12	3	50%	No	0.11
1	75%	n/a	0.00	2	75%	No	0.11	3	75%	No	0.11
1	100%	n/a	0.00	2	100%	Yes	0.18	3	100%	Yes	0.21
4	25%	No	0.11	5	25%	No	0.11	6	25%	No	0.12
4	50%	No	0.12	5	50%	No	0.13	6	50%	No	0.12
4	75%	No	0.14	5	75%	No	0.14	6	75%	No	0.12
4	100%	Yes	0.23	5	100%	Yes	0.28	6	100%	Yes	0.30

Table 7: Use case monitoring 2, with subscription-based fusion analyzing 10000 time steps.

Another point that can be investigated regards answering the question: Can increasing/decreasing the period of the fusion node improve the performance of the system? In Table 8, we observe the results (when monitoring the first camera) with fusion periods 250 and 750. It can be seen that both alternatives, in comparison to a fusion time of 500, are performing worse, with only four and cameras being safe for the shorter period and none for the longer period.

Further investigations can be made by additional queries with different confidence intervals, limits on guarantees, etc., to help in gathering insights of the system model, and allow for experimentation with different parameters, aiming to help eliminate poor design choices.

8 Related Work

The closest to our work is the TA-based approach, proposed by Halder et. al. [11], 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. [9] 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. [14] propose formal models for the real-time publish-subscribe protocol using UPPAAL and analyze the protocol's behavior by sim-

#Cams	Load	≤ 850	Time	#Cams	Load	≤ 850	Time	#Cams	Load	≤ 850	Time
1	25%	Yes	0.22	2	25%	Yes	0.24	3	25%	Yes	0.33
1	50%	Yes	0.22	2	50%	Yes	0.25	3	50%	Yes	0.29
1	75%	Yes	0.26	2	75%	Yes	0.28	3	75%	Yes	0.31
1	100%	Yes	0.30	2	100%	Yes	0.28	3	100%	Yes	0.32
4	25%	Yes	0.26	5	25%	Yes	0.32	6	25%	No	1.32
4	50%	Yes	0.29	5	50%	No	0.16	6	50%	No	0.12
4	75%	Yes	0.32	5	75%	No	0.13	6	75%	No	0.11
4	100%	Yes	0.35	5	100%	No	0.11	6	100%	No	0.12

#Cams	Load	≤ 850	Time	#Cams	Load	≤ 850	Time	#Cams	Load	≤ 850	Time
1	25%	No	0.11	2	25%	No	0.12	3	25%	No	0.12
1	50%	No	0.12	2	50%	No	0.11	3	50%	No	0.12
1	75%	No	0.10	2	75%	No	0.11	3	75%	No	0.11
1	100%	No	0.10	2	100%	No	0.11	3	100%	No	0.12
4	25%	No	0.11	5	25%	No	0.12	6	25%	No	0.12
4	50%	No	0.11	5	50%	No	0.11	6	50%	No	0.13
4	75%	No	0.11	5	75%	No	0.12	6	75%	No	0.10
4	100%	No	0.11	5	100%	No	0.11	6	100%	No	0.12

Table 8: Use case monitoring 1, with timer-based fusion period of 250 (above) and 750 (below) analysing 10000 time steps.

ulation in Simulink/Stateflow, however the authors do not validate the formal models against the simulation results, as we show in this paper.

Carvalho et al. [7] introduce a model-checking technique for verifying system-wide safety properties in message-passing systems, employing an Alloy extension called Electrum, and its Analyzer. Their approach emphasizes high-level architectural verification of message passing, while our work focuses on timing properties verification. Webster et al. [20] propose a formal verification method for industrial robotic programs using the SPIN model checker, with an emphasis on behavioral refinement and the verification of selected robot requirements.

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 [8], and that of Anand and Knepper proposes ROSCoq, a framework for developing certified Coq programs for robots, where subsystems communicate using messages [3]. Neither of these works focuses on verifying end-to-end reaction time of job chains, the authors analyzing implementation levels instead.

The work of Hong et al. [19] can be seen as complementary to our work, as it bridges the Timed Rebeca [1] model of a realistic multiple autonomous mobile robots system and its generated ROS 2 demo code, showing the match between the model and the program. Our work does not focus on ROS 2 code generation, hence on such semantic bridge, but on modeling and analyzing formally the main communication concepts of ROS 2, both in a deterministic and a probabilistic context.

9 Conclusions and Future Work

In this paper, we have presented a formalization of ROS semantics in UPPAAL timed automata. To establish its validity, we ensure equal behavior with previous work available in the literature, with respect to the application's timeliness. Afterwards, we have extended it with non-determinism, by allowing variable runtime of tasks, as well as probabilistic data generation, by employing stochastic timed automata. To demonstrate the usability of the approach, we demonstrate how the UPPAAL tool can be used with regular and statistical model checking, respectively, to establish upper bounds of end-to-end timing properties of ROS 2 applications.

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. Luca Aceto, Matteo Cimini, Anna Ingólfsdóttir, Arni Hermann Reynisson, Steinar Hugi Sigurdarson, and Marjan Sirjani. Modelling and Simulation of Asynchronous Real-Time Systems using Timed Rebeca. In *Proceedings 10th International Workshop on the Foundations of Coordination Languages and Software Architectures, FOCLASA 2011*, volume 58 of *EPTCS*, pages 1–19, 2011.
2. Rajeev Alur. Timed automata. In Nicolas Halbwachs and Doron Peled, editors, *Computer Aided Verification*, pages 8–22, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.
3. Abhishek Anand and Ross Knepper. ROSCoq: Robots Powered by Constructive Reals. In *Interactive Theorem Proving*, pages 34–50, Cham, 2015. Springer International Publishing.
4. Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.
5. Johan Bengtsson and Wang Yi. *Timed Automata: Semantics, Algorithms and Tools*, pages 87–124. Springer Berlin Heidelberg, 2004.
6. P. Bulychev, A. David, K.G. Larsen, A. Legay, G. Li, and D.B. Poulsen. Rewrite-based Statistical Model Checking of WMTL. In *Runtime Verification Conference (RV 2012)*, pages 260–275. Springer, 2012.
7. Renato Carvalho, Alcino Cunha, Nuno Macedo, and André Santos. Verification of system-wide safety properties of ros applications. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2020.
8. 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.
9. 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.
10. 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.
11. 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.
12. 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.
13. 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.
14. 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.
15. 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.
16. 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.
17. 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.
18. 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.
19. Hiep Hong Trinh, Marjan Sirjani, Fereidoun Moradi, Antonio Cicchetti, and Federico Ciccozzi. Combining model-based development and formal verification of a complex ROS2 multi-robots system using Timed Rebeca. In *International Workshop on Reliability Engineering Methods for Autonomous Robots – REMARO 2024*, June 2024.
20. Matt Webster, Clare Dixon, Michael Fisher, Maha Salem, Joe Saunders, Kheng Lee Koay, Kerstin Dautenhahn, and Joan Saez-Pons. Toward Reliable Autonomous Robotic Assistants Through Formal Verification: A Case Study. *IEEE Trans. on Human-Machine Systems*, 46(2):186–196, 2016.

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<JOBS_COUNT; i++) {
        for (j=0; j<JOBS_COUNT-1; j++) {
            if (PRIO[JOBS[j][0]] > PRIO[JOBS[j+1][0]]) {
                tmp_id = JOBS[j][0];
                tmp_data = JOBS[j][1];
                JOBS[j][0] = JOBS[j+1][0];
                JOBS[j][1] = JOBS[j+1][1];
                JOBS[j+1][0] = tmp_id;
                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;

```

```
48      assert(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              JOBS[JOBS_COUNT][0] = i;
53              JOBS[JOBS_COUNT][1] = j;
54              JOBS_COUNT += 1;
55          }
56      }
57      schedule();
58  }
59
60  int next_job_idx() {
61      JOBS_COUNT--;
62      return 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_PAYLOAD) {
82          PAYLOAD = FIRST_PAYLOAD;
83      }
84      nm = (nm + 1) % MONITORS;
85  }
86
87
88  // True if payload is monitored (i.e., not empty or already
        seen)
89  bool relevant_payload(int payload) {
90      int i = 0;
91      if (payload == EMPTY)
92          return false;
93      for (i = 0; i < MONITORS; i++) {
```

```
94          if (monitor_payload[i] == payload)
95              return true;
96      }
97
98      return false;
99  }
100
101
102
103
104 // When freeing up, we free up all monitors incl. those whose
        data got thrown away.
105 void free_monitors() {
106     int i;
107     // We could get an old value, in that case it is not in
        the monitored payloads
108     bool old_value = true;
109     for (i = 0; i < MONITORS; i++) {
110         if (monitor_payload[i] == pd)
111             old_value = false;
112     }
113
114     // If it is an old value, we can just ignore it as it has
        already been seen once
115     if (old_value)
116       return;
117
118     // Free previously used monitors
119     while (monitor_payload[lm] != pd) {
120         monitor_status[lm] = MONITOR_FREE;
121         monitor_payload[lm] = EMPTY;
122         lm = (lm + 1) % MONITORS;
123     }
124
125     // Also free the one just handled.
126     monitor_status[lm] = MONITOR_FREE;
127     LAST_PAYLOAD = monitor_payload[lm];
128     monitor_payload[lm] = EMPTY;
129     lm = (lm + 1) % MONITORS;
130 }
```

B Graphical Representation of UPPAAL TA

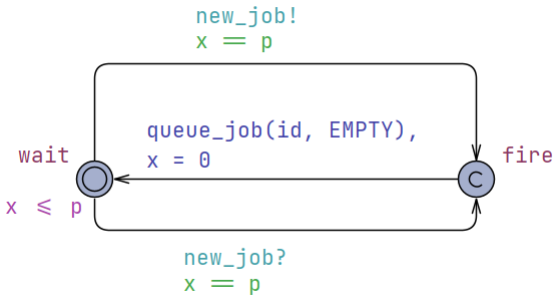


Fig. 14: UPPAAL figure of data-generator.

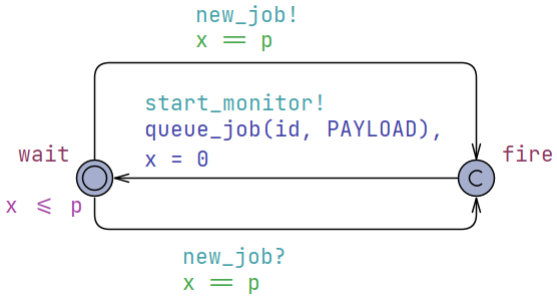


Fig. 15: UPPAAL figure of monitored data-generator.

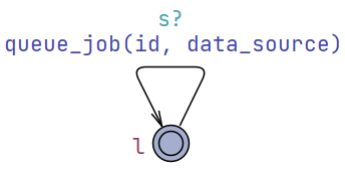


Fig. 16: UPPAAL graphical representation of subscriber.

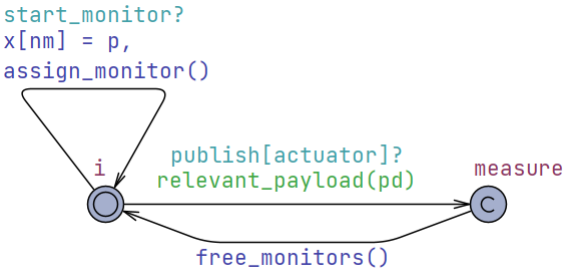


Fig. 17: UPPAAL graphical representation of Monitor.

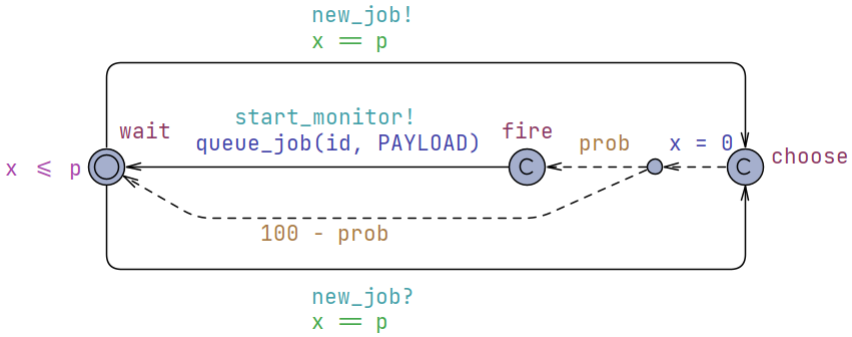


Fig. 18: Template for monitored probabilistic data-generator.

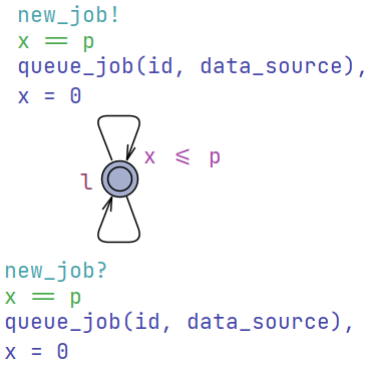


Fig. 19: Template for timer.

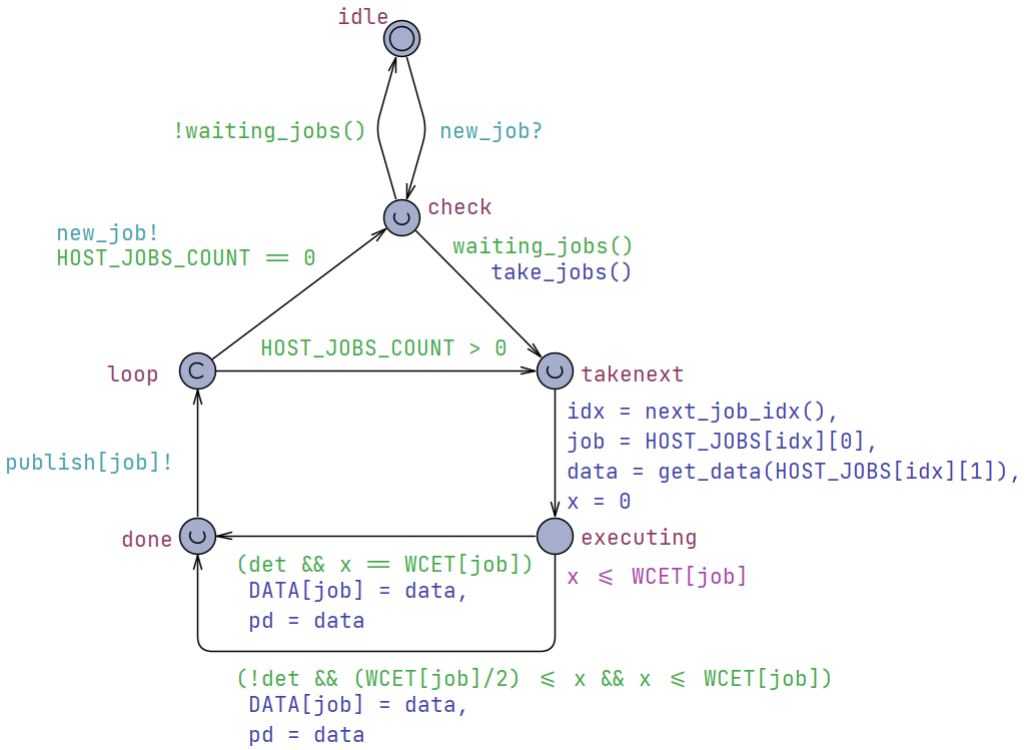


Fig. 20: Template for host.

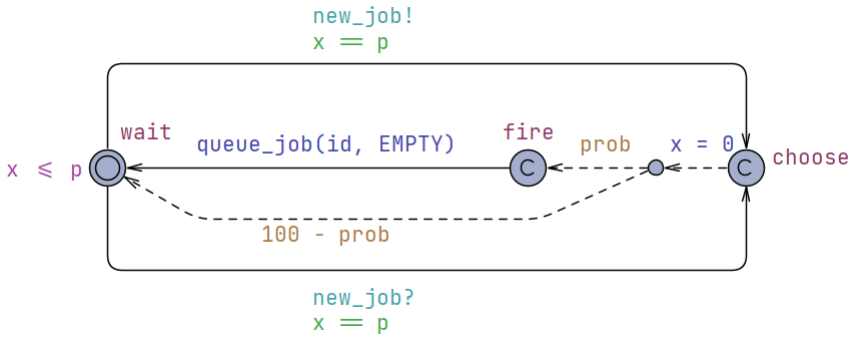


Fig. 21: Template for probabilistic data-generator.