Formal Verification of ROS-based Robotic Applications using Timed-Automata

Raju Halder
HASLab, INESC TEC, Portugal &
Indian Institute of Technology Patna, India
halder@iitp.ac.in

José Proença Nuno Macedo André Santos
HASLab, INESC TEC &
Universidade do Minho, Braga, Portugal
{jose.proenca,nfmmacedo}@di.uminho.pt
contact.andre.santos@gmail.com

ABSTRACT

Robotic technologies are continuously transforming the domestic and the industrial environments. Recently the Robotic Operating System (ROS), an open source middleware framework, has been widely adopted both by industry and academia becoming the de facto standard for developing robot applications. Guaranteeing the correct behaviour of robotic systems is, however, challenging due the large amount of different parameters and heterogeneity of these systems. Different approaches exist focusing on concrete domain spaces for specific scenarios, but no general approach exists. This paper proposes an approach to model and verify ROS-based systems using real time properties, focusing on the communication between ROS nodes. It takes low-level parameters into account, such as queue sizes and timeouts, and uses timed automata as the modelling language. We use a physical robot Kobuki as a complex case study, and use the UPPAAL model checker to automatically verify properties, identifying problematic parameter combinations.

1. INTRODUCTION

Robotic technologies have dramatically transformed our world by bringing countless benefits to many sectors, including the domestic environment, industrial production sectors, health-care and military activities, leading to an increasingly closer human interaction, where failures can have catastrophic consequences. In this context, verifying the correction of a robot's software controllers is an additional burden imposed on the developers, which are often oblivious of software engineering best practices.

In recent years the *Robot Operating System* (ROS) [9] has gained attention both in industry and academia, and has become the de facto standard for the development of robotic applications. ROS is an open source middleware framework that provides common robot-specific services and libraries, such as component communication, hardware abstraction, and low-level device control. The fundamental components in ROS-based applications are nodes (or processes) that communicate through a publisher-subscriber paradigm, where messages are organized into named topics. A node (e.g, a sensor) shares information by publishing messages on the appropriate topic, while nodes that want to receive information (e.g., an actuator) subscribe to the relevant topics. A special node, dubbed ROS master, ensures that publishers and subscribers find each other in order to establish peer-to-peer communications.

ROS gives a lot of flexibility to developers. They can choose from a set of popular programming languages, and customise core libraries to modify architectural parameters such as incoming and outgoing queue sizes, maximum time to wait for incoming messages, and rate of publishing. Consequently there is no complete solution to formally analyse and verify ROS programs, and different approaches have been proposed. These include the use of model checkers, static analysis techniques, proof assistants, and runtime monitors [11, 12, 1, 2, 8, 10]. André et al. [10] provide an attempt to give a global view over code quality of ROS applications by combining existing tools that apply code quality metrics and test against code standards.

This work presents a generic approach to verify real-time properties of ROS-based applications, with special focus on node communication. ROS applications are modelled using timed automata [6], and properties are verified using model checking – more specifically the UPPAAL model checker [5]. Our approach is illustrated via a small example, where two publishers communicate with a subscriber, and a more complex example that models the controllers of the physical robot *Kobuki*[1]. Using our approach we are able to identify problematic combinations of configuration parameters, such as queue sizes and timeouts, and possible problems with the existing code used by Kobuki.

To summarize, our main contributions in this paper are:

- Formalisation of concrete ROS-based applications, varying the values of architectural parameters;
- Use of the UPPAAL model checker to implement and verify ROS-based applications;
- Application of our proposed approach to the ROS-based Kobuki robot as a case study, describing the construction of a timed model from the sourcecode and the verification of safety and liveness properties.

This paper is structured as follows. Section 2 describes related approaches. Section 3 recalls timed automata and a

[1] `http://kobuki.yujinrobot.com/`

logic to specify timed properties. Section 4 describes how to model and verify ROS applications using timed automata. Section 5 illustrates our approach using the more complex Kobuki case study. Finally, Section 6 concludes this paper.

2. RELATED WORK

Je Huang et al. [8] proposed ROSRV, a runtime verification framework for safety and security properties of ROS-based applications. ROSRV provides a specification language to express safety properties, and automatically generates ROS nodes that monitor said properties during execution. Recently, Adam et al. [1] introduced Declarative Robot Safety (DeRoS), a Domain-Specific Language, to express functional safety properties for mobile robots. The idea is similar to ROSRV, in that it also produces nodes that monitor these properties during runtime. However, both approaches are limited in terms of expressiveness, and both incur some overhead due to monitor activity.

Cowley and Taylor [7] proposed a static verification of robot behaviour using dependent type theory and linear logic embedding in Coq. More recently, Anand and Knepper [2] presented ROSCoq, a Coq framework for developing certified systems in ROS by extending the LoE framework, to enable holistic reasoning about the cyber-physical behaviour of robotic systems. The use of CoRN's theory of constructive real analysis enables the framework to accurately reason about computations with real numbers. Nonetheless, even correct-by-construction code produced by Coq is prone to flaws in ROS-specific architectural constraints.

Webster et al. [12] proposed a formal verification approach of industrial robotic programs using the SPIN model checker, addressing only three properties – deadlocks, collisions, and kill-switch violations. The proposal does not involve any kind of ROS-specific architectural properties. The authors use SPIN to formally verify the ROS-based autonomous robotic assistant "Care-O-bot" [11]. The proposal is very specific to that particular robot, and is aligned only towards the verification of high-level decision making rules.

3. PRELIMINARIES: TIMED AUTOMATA

Timed automata [6, 3, 4] is one of the most widely used formal models to specify and verify real-time systems. A timed automaton consists basically of a finite automaton extended with a set of real-valued variables modelling clocks. Transitions are labelled by constraints defined on clock variables (called clock-constraints) to restrict the behaviour of an automaton. All clocks of an automaton are initialized to zero when the system is started, and increase synchronously whenever time evolves. Individual clocks may be reset to zero when certain transitions are taken.

This section starts by formalising timed automata, followed by a brief explanation on how to verify systems modelled with timed automata using temporal logics.

3.1 Specifying Timed Automata

Timed automata are labelled transition systems enriched with constraints over so-called *clocks*. A clock is a special variable capturing the time passed since it was last reset.

Definition 1 (Clock Constraint). *A clock constraint g over a set of clocks C given by the grammar $g ::= true \mid x \odot n \mid x - y \odot n \mid g \wedge g$, where $n \in \mathbb{N}$, $x, y \in C$ and $\odot \in \{>, \geqslant, =, <, \leqslant\}$.*

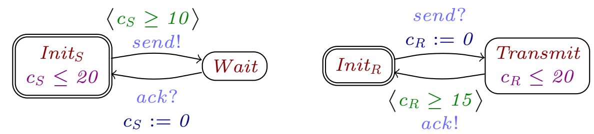


Figure 1: Network of two timed automata, modeling a sender (left) and a receiver (right).

Definition 2 (Timed Automata). *A timed automaton is a tuple $\langle L, l_0, \Sigma, C, T, Inv\rangle$ where L is a finite set of locations, $\ell_0 \in L$ is the initial location, Σ is a finite alphabet of actions, C is a set of clocks, $T \subseteq L \times CC(C) \times \Sigma \times 2^C \times L$ is the set of transitions, $CC(C)$ denotes the set of all clock constraints over C, and $Inv : L \to CC(C)$ assigns invariants to locations.*

Intuitively, a connector in a location ℓ can evolve either by (1) letting time pass, i.e., by incrementing all its clocks without breaking the invariant $Inv(\ell)$, or by (2) taking a transition (ℓ, g, a, C, ℓ') if the conditions g and $Inv(\ell')$ hold, going to the location ℓ' and setting the clocks in C to zero.

The actions in the alphabet Σ are used to synchronise with other automata. More precisely, two automata with a shared action $a \in \Sigma$ are only allow to take a transition labelled with a when the other automata can also take a transition with a. A set of automata running in parallel, synchronising actions and evolving their clocks simultaneously, is called a *network of timed automata*. We follow the convention that action synchronisation can only occur in pairs, and we use their notation $a!$ and $a?$ to mean that performing $a!$ triggers $a?$ to be performed [5]. Furthermore, we omit clock constraints, actions, and reset sets from the labels whenever they are *true*, irrelevant, and $\emptyset$, respectively. This is illustrated in the example below with a network of two timed automata.

Example 1. *We depict in Figure 1 a network of two simple timed automata. Colours are used to distinguish the different elements: node invariants, guards, actions, and clock resetting. We use angle brackets $\langle\cdot\rangle$ to denote guards on edges, and double-lines to denote initial states.*

Initially both the sender S and the channel C are in their left locations, and their clocks c_S and c_R are set to 0. Then S can wait at most 20 time units until it can fire send! and go to the Wait state, making R to take the send? transition. The clock c_R is reset in the process, and the automata can now wait at most 20 time units until the ack! and ack? actions can be taken. The guards in the labels produce a delay of at least 10 and 15 time units when taking a transition, when in the left and the right locations, respectively.

UPPAAL extensions. To ease the encoding of ROS systems, we rely on some UPPAAL extensions: (1) *committed states*, (2) *internal variables*, and (3) *parametric actions*. Committed states are special states with a time invariant that does not allow time to pass, and with higher priority than any other (non-committed) state. Internal variables are variables assigned to each automata, which are bounded and can be both read in the guards (together with the clock constraints) and updated via an *update statement*, i.e., after the transition is taken, together with the reset of the clocks. Finally, actions can have parameters and variables, i.e., it is possible to write the action $send(42)!$ to send a value 42, and $send(x)?$ to bound a received value to the variable x.

The formal semantics of these extensions is omitted for simplicity, but can be found in the literature (e.g., [6, 4]).

3.2 Verifying Timed Automata with UPPAAL

UPPAAL [5] is a model-checker toolbox based on the theory of timed automata which performs forward analysis with extrapolation. It provides some extra features, such as bounded integer variables and broadcast channels. This section presents a temporal logic named *Timed Computation Tree Logic* (TCTL) [5, 3], used by UPPAAL as a query language to describe desired properties of (networks of) timed automata. This query language consists of path formulas ϕ, which in turn use more dedicated state formulas ψ. State formulas are defined over automata locations and clocks.

Definition 3 (TCTL formulas). *A TCTL formula ϕ is given by the grammar below.*

$$\phi \ ::= \ \exists\Diamond\,\psi \mid \forall\Diamond\,\psi \mid \exists\Box\,\psi \mid \forall\Box\,\psi \mid \psi_1 \rightarrow \psi_2$$
$$\psi \ ::= \ A.\ell \mid g \mid \neg\psi \mid \psi_1 \vee \psi_2 \mid \psi_1 \wedge \psi_2 \mid \psi_1 \Rightarrow \psi_2$$

$A.\ell$ represents the location ℓ in the automaton A, g is a clock constraint, and $\neg$, $\vee$, $\wedge$ and $\Rightarrow$ represent the usual logical negation, disjunction, conjunction, and implication. The temporal operators $\exists$, $\forall$, $\Diamond$, and $\Box$ describe the range of states for which the state formulas ψ must hold, and $\psi_1 \rightarrow \psi_2$ is a shorthand for $\forall\Box\,(\psi_1 \Rightarrow \forall\Diamond\,\psi_2)$ (which cannot be written in our syntax), read ψ_1 leads to *ψ_2.*

We make precise the meaning of the temporal operators using the timed automata in Figure 1 as a running example.

$\exists\Diamond\,\psi$ means that there must *exist* a sequence of transitions such that, at some point, ψ holds. For example, $\exists\Diamond\, S.Inits \Rightarrow (c_S > 19)$ means that the clock c_S can become higher than 19 while in location *Inits* in S.

- $\forall\Diamond\,\psi$ means that *for every* sequences of transitions, at some point ψ can hold. For example, $\forall\Diamond\, c_R > 19$ means that, at any given point of the execution, one can find a future state where c_R is higher than 19.

- $\exists\Box\,\psi$ means that there must *exist* a sequence of transitions such that ψ *always* holds. For example, $\exists\Box\,(c_S = 11) \Rightarrow S.Wait$ means that there exist a sequence of transition where the clock c_S is 11 while the automaton S is in *Wait* location.

- $\forall\Box\,\psi$ means that *for every* sequences of transitions, ψ *must hold* in every intermediate state. For example, $\forall\Box\,(c_S \geq 0 \wedge c_S \leq 40)$ means that the clock c_S will always be within 0 and 40 in the automaton S at any point of executions.

- $\psi_1 \rightarrow \psi_2$ (i.e., $\forall\Box\,(\psi_1 \Rightarrow \forall\Diamond\,\psi_2)$) denotes that whenever ψ_1 holds then ψ_2 must eventually hold. For example, $S.Wait \rightarrow R.Transmit$ means that, once $S.Wait$ is reached, $R.Transmit$ will always be reachable.

4. VERIFYING ROS APPLICATIONS

This section describes how to model and verify ROS-based applications with time constraints, using as a running example a publisher-subscriber implementation. We start by exploring how to *extract* key parameters from the source code of ROS applications (Subection 4.1), which are then used to *formally model* them as a network of timed automata (Subsection 4.2). UPPAAL is then used to reason and to *verify properties* about such applications (Subsection 4.3).

Code Snippet 1: A Subscriber Node

```
void chatterCallback(const
    std_msgs::String::ConstPtr msg) {
  //... do some work ...
}

int main(int argc, char **argv) {
  ros::init(argc, argv, "listener");
  ros::NodeHandle n;
  ros::Subscriber sub =
      n.subscribe<std_msgs::String>("chatter",
      1000, chatterCallback);
  ros::Rate loop_rate(10);
  while (ros::ok()) {
     //... do some work ...
     ros::spinOnce();
     loop_rate.sleep();
  }
  return 0;
}
```

4.1 Code Analysis of a ROS Application

The fundamental components in ROS-based applications include *nodes* (or processes), transmission *channels* (or *topics*), and *messages*. Nodes communicate via a publisher-subscriber message passing mechanism: a publisher can send a message to a given *channel*, and every subscriber of that channel will receive the message. Publisher nodes send messages to a channel by adding them into the channel's queue, which are subsequently dequeued and added to the subscribers' queue. Observe that the same channel can be used by multiple publishers and subscribers [9].

An example of a subscriber node in ROS is depicted in Code Snippet 1. Observe that the node subscribes the channel `chatter` of the message type `std_msgs::String`, with a queue-size of 1000. By invoking `ros::spinOnce` in a regular interval (the `loop_rate` object is set to 10 in this example), the node processes incoming messages in the queue by executing the callback function `chatterCallback`.

Observe that the rate at which ROS can empty a publishing queue depends on the time taken to actually transmit the messages to subscribers, and is largely out of our control. In contrast, the speed with which ROS empties a subscribing queue depends on how quickly it processes callbacks. Thus, the application developer is responsible for setting reasonable publisher and subscriber queue sizes to avoid overflows. When a queue is full, new upcoming messages will replace the oldest ones. Note that one can reduce the likelihood of a subscriber queue overflowing by (1) ensuring that callbacks, via `ros::spin` or `ros::spinOnce`, are frequent, and (2) reducing the amount of time consumed by each callback.

The static analysis phase, currently not automated, requires the extraction of ROS code parameters that affect the desired properties of ROS-based robotic applications, including the publisher's publishing rate, the subscriber's "spin" rate to process callbacks, the time to transmit messages over channels, and the time to process callbacks.

4.2 ROS Applications as Timed Automata

Consider a ROS application with three processes, publishers P_1 and P_2 and subscriber P_3 for a channel Ch_1 (Figure 2). The notation $Q_{i \rightarrow j}$ denotes a queue associated with

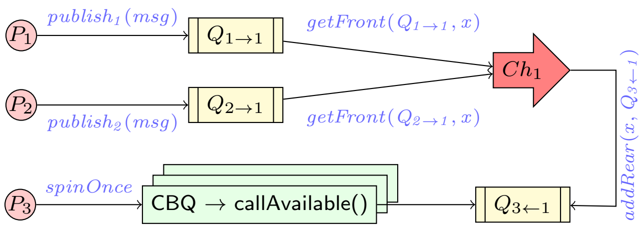


Figure 2: Simple ROS publisher-subscriber scenario.

the *publisher* P_i and the channel Ch_j, and $Q_{i\leftarrow j}$ a queue associated with the *subscriber* P_i and the channel Ch_j. This ROS publisher-subscriber mechanism is modelled in Figure 3 as timed automata. Different values of the parameters PubTime, SubTime, Tmin, Tmax, CBmin, and CBmax yield different variations of the automata.

Publishers P_1 and P_2 are uniquely identified with *id* as a parameter (Figure 3(a)), and send messages every PubTime time-units. The subscriber invokes `ros::spinOnce` to process callbacks every SubTime time-units (Figure 3(b)), and the transmission of messages over the channel takes between Tmin and Tmax time-units (Figure 3(e)). Variable CBavail represents the number of queued callback invocations, and is shared by the automata in Figures 3(d) and 3(f). The *replaceOld*() method of $Q_{3\leftarrow1}$ replaces the oldest message in the queue by the upcoming message when the queue is full.

When `ros::spinOnce` is invoked the method `callAvailable()` is called on the callback queue CBQ, which processes all callbacks currently in the queue. The processing of callbacks takes between CBmin and CBmax time-units. Observe that we model the `callAvailable()` method without a timeout parameter in `ros::CallbackQueue` (Figure 3(f)).

4.3 Verification in UPPAAL

The corresponding implementation of the models in UPPAAL is presented in Appendix A. Varying queue sizes and time constraints, we verify the following properties about the associated queues ("*whether no path leads to an overflow of the queue*") using the UPPAAL model checker:

$$\mathsf{Pr}_1\colon\ \forall\Box\ \neg Q_{1\to1}.\mathit{Overflow}$$
$$\mathsf{Pr}_2\colon\ \forall\Box\ \neg Q_{2\to1}.\mathit{Overflow}$$
$$\mathsf{Pr}_3\colon\ \forall\Box\ \neg Q_{3\leftarrow1}.\mathit{Overflow}$$

Using UPPAAL it is possible to experiment different combinations of values of parameters and investigate which ones validate these desired properties. The parameters define the three queue sizes, the transmission time, the processing callback time, the publishing time-gap, and the spin time-gap.

For example, using the assignment $Q_{1\to1} = Q_{2\to1} = Q_{3\leftarrow1} = 5$, Tmin $= 3$, Tmax $= 4$, P_1.PubTime $= 8$, P_2.PubTime $= 7$, and P_3.SubTime $= 15$, none of the three properties hold. By using instead P_2.PubTime $= 8$ and P_3.SubTime $= 18$ the properties Pr_1 and Pr_2 hold, and all property hold if we lower the value of P_3.SubTime to 17.

5. CASE STUDY: KOBUKI ROBOT

Kobuki is a ROS open source robotic application[2,3] developed by Yujin Robotics (Korean firm) and Willow Garage (from USA) for research and educational purposes.

[2] `http://wiki.ros.org/kobuki`
[3] `https://github.com/yujinrobot/kobuki`

5.1 Kobuki Source Code Analysis

Kobuki is integrated with various sensors, velocity controllers, a command multiplexer, and a high precision motor. The schematic diagram of its ROS-based architecture is depicted in Figure 4. Our analysis focuses on the SafetyController, which identifies obstacles and tries to move the robot to a safer position, and the Multiplexer, which manages movement messages that arrive from different controllers.

The SafetyController-Update node subscribes the *events/wheel_drop*, *events/bumper* and *events/cliff* channels, to receive messages from the wheel-drop, bumper and cliff sensors, respectively. Published messages are enqueued into the corresponding subscriber queues (*QWheel*, *QBumper*, and *QCliff*, respectively). These queues are inspected at a given rate by invoking the callAvailable() method, processing the sensor messages and updating shared boolean state variables capturing, e.g., if the left wheel is dropped. Based on these shared variables, the SafetyController-Publisher node publishes at a fixed rate command-velocity (*CmdVel*) messages to a channel subscribed by the Multiplexer node, such as "stop" when wheel-drop events occurs or "move back" if the bumper is pressed or a cliff detected. In turn, Multiplexer combines these messages with messages from other nodes that control the robot, like a RandomWalker node, giving higher priority to messages from the safety controller.

5.2 Timed Modeling of the Safety Controller

This section formally specifies the `SafetyController-Update` component. `SafetyController-Publisher` can be modelled as a traditional publisher, as depicted in Section 4.2.

The upper half of the architecture from Figure 4 is modelled by the automata in Figure 5. Figure 5(a) models any of the three sensors (*Wheel-Drop*, *Bumper*, or *Cliff*) and their position (*Left*, *Center*, or *Right*)). Its time constraint ensures that sensors wait at least 1 time unit before publishing a new message. Figure 5(b) models any of the subscriber queues assigned to the safety controller. The variable CBavail, shared with the automaton in Figure 5(d), captures the amount of received messages, which will trigger the addition of callbacks to the callback queue.

The SafetyController-Update node (Figure 5(c)) is a subscriber that processes incoming sensor messages by invoking `ros::spinOnce` and updates the state accordingly. This is done by periodically calling callAvailable() (Figure 5(d)), which processes all callbacks in the callback queue. The former periodically calls `ros::spinOnce` based on the spinRate parameter. Observe that callAvailable() is parameterised by a TimeOut parameter that controls how long to wait for a callback to be available before returning. In ROS 0.10 the default timeout is 0.1 seconds, whereas in ROS 0.11 it is 0 seconds. A complete implementation of the SafetyController-Update is included in Appendix B for reviewing purposes.

UPPAAL Verification of SafetyController-Update. Using the timed automata models in Figure 5 it is possible to experiment with different parameters and queue sizes and verify if desired properties are valid. The results of such experiments can be found in Table 1, using the properties Pr_W, Pr_B, and Pr_C.One can conclude, for example, that no sensor will overflow its queue when all queues have size 12, the spin rate of the safety controller and the timeout for the callAvailable are 1, and the callback time is between 1 and 2.

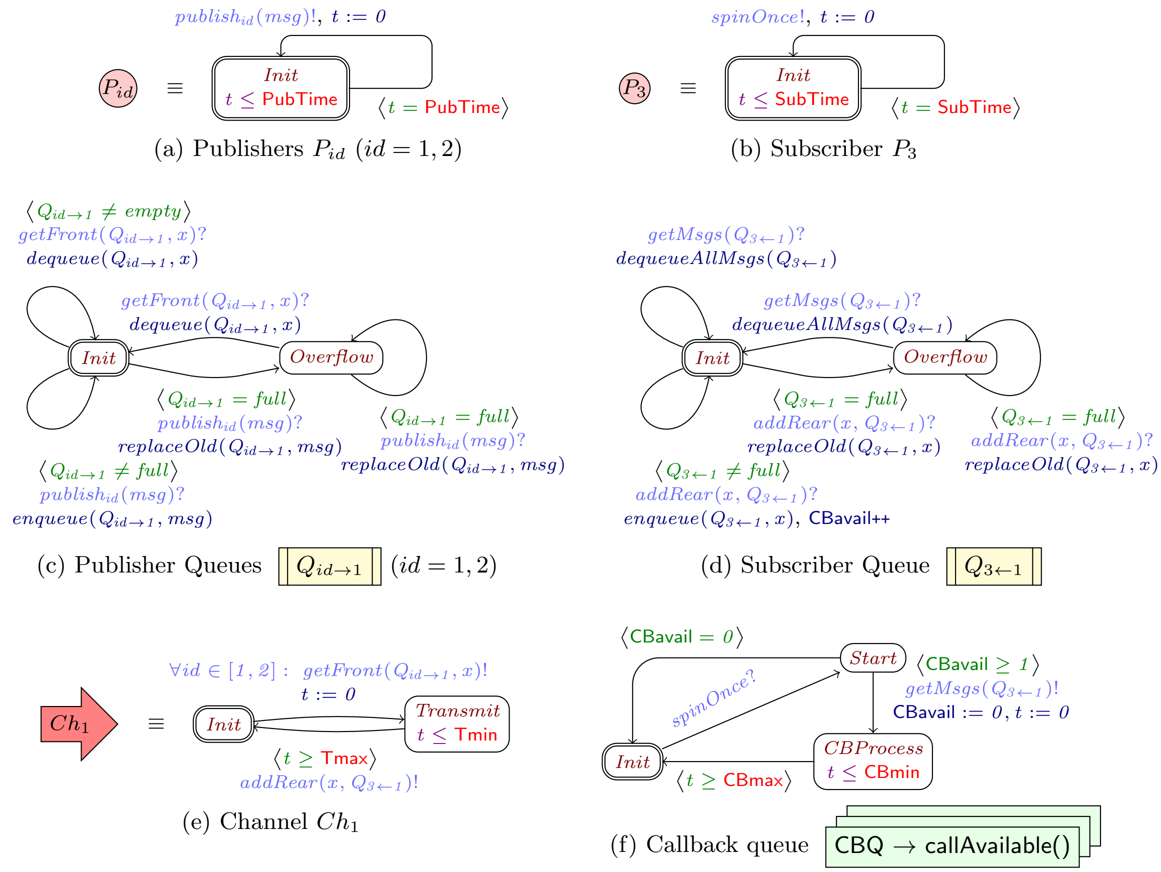


Figure 3: Formal timed modelling of a ROS publisher-subscriber message passing scenario.

Queue sizes of safety controller			Spin time-gap of safety controller	callAvailable() timeout	Callback time		Properties		
QWheel	*QBumper*	*QCliff*	spinRate	TimeOut	CBmin	CBmax	Pr_W	Pr_B	Pr_C
10	10	10	1	2	1	2	✓	✗	✗
				4	1	2	✓	✗	✗
			2	2	1	2	✓	✗	✗
			3	2	1	2	✗	✗	✗
12	12	12	1	1	1	2	✓	✓	✓
					4	5	✗	✗	✗
			3	2	1	2	✓	✗	✗
			6	2	1	2	✗	✗	✗

Table 1: Queue-Overflow w.r.t. various dependable parameters in the module SafetyController-Update.

5.3 Finding problems in Kobuki

In addition to the above safety properties pertaining to queue overflow, this section identifies some desirable, context specific, properties of the Kobuki system. Uusing the UPPAAL model checker, we will show that the safety controller node may lose (important) information from the sensors in the presence of overflows (Subsection 5.3.1), and that in some scenarios, messages from the RandomWalker never reach the Kobuki engine (Subsection 5.3.2).

5.3.1 *Lost Sensor Messages*

The models in Figure 5 feature the timing constraints and queue sizes but do not encode the particular behaviour of the Kobuki nodes, like the message processing or the update of the internal state. This subsection shows that a sensor—we will use the left wheel sensor—may fail to trigger the desired change in the state variables of the safety controller. For this, we enhance the model for the wheel sensor by replacing the one in Figure 5(a) by an equivalent one that alternates between *on* and *off* states. Assuming the safety controller state variable *wheel_left_dropped* represents if the wheel is dropped, the desired property can be formulated as follows.

$$\text{Wheel\_Left}.\mathit{on}\ \&\ \text{SafetyController-Update}.\mathit{spinLoc} \rightarrow \mathit{wheel\_left\_dropped} \quad \textbf{(Sensor-Property)}$$

This formula asserts that, whenever the left wheel is dropped and the safety controller invokes `ros::spinOnce`, the event will eventually be reflected in the corresponding safety controller's state variable *wheel_left_dropped*.

The property validity depends on whether the subscriber queue *QWheel* may or not overflow. If *QWheel* can overflow, the property will not be satisfied, since the Wheel_Left.*on* sensor message may be replaced by other sensor message due to queue-overflow. Otherwise the property holds, which

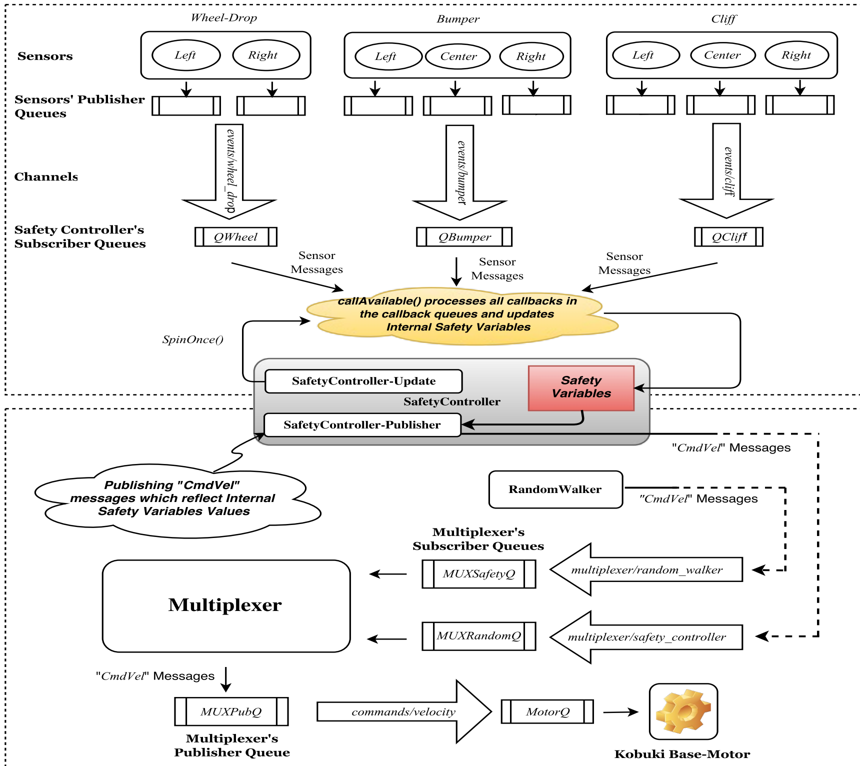
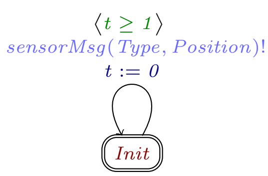
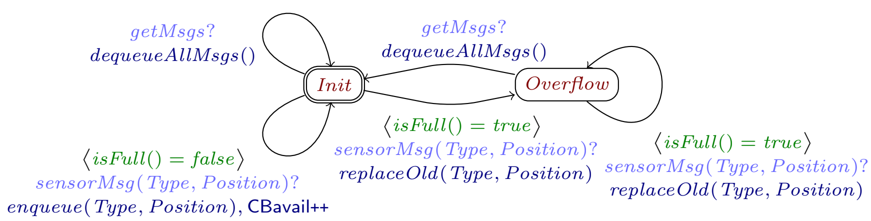


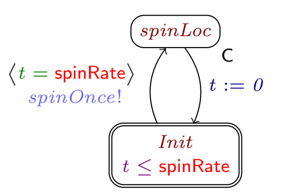
Figure 4: Schematic diagram of ROS-based Kobuki architecture



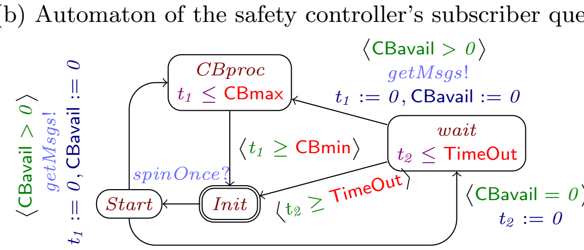
(a) Automaton of a sensor.



(b) Automaton of the safety controller's subscriber queue.



(c) SafetyController-Update (C denotes *committed state*).



(d) callAvailable (ros::WallDuration TimeOut) method to process all callbacks in the callback queue of the safety controller.

Figure 5: Formal Timed Modeling of the module SafetyController-Update

means the safety controller will always react to the messages from the left wheel sensor. This example shows the importance of correctly setting the parameters to ensure queues do not overflow in our Kobuki case study, and the advantages of formally verifying which parameters can be used.

5.3.2 *Lost Messages From RandomWalker*

Kobuki supports multiple nodes to control the movement of the robot by simply sending command velocity messages to the Multiplexer, which is responsible to sort and filter out messages based on their priority. In our example there is one such node, the RandomWalker. Thus, the Multiplexer subscribes two topics, from the safety controller and from the random walker nodes, and sets a timer used by the callbacks cmdVelCallback() and timerCallback().

The cmdVelCallback() and timerCallback() callbacks are

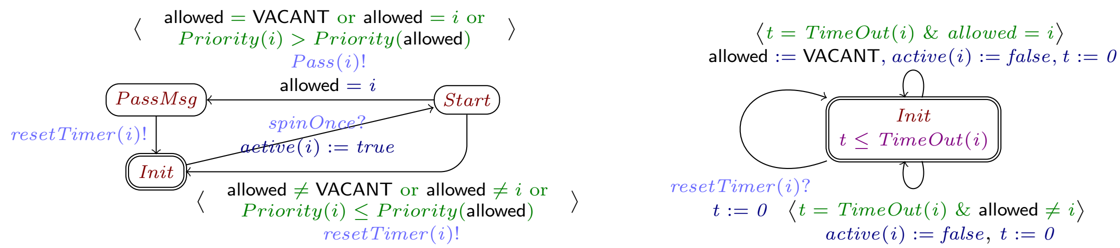


Figure 6: Timed Model of cmdVelCallback() (left) and timerCallback() (right) from the Multiplexer.

formalised as timed automata in Figure 6 – the automata for the remaining subscriber and publisher queues of the Multiplexer can be defined as in previous sections. The cmdVelCallback(), when processing a value from the i^{th} subscribed topic, acts as follows. It resets and starts the timer associated to the i^{th} topic – this timer will trigger timerCallback() at a fixed rate. It assigns $active(i)$ to true, indicating the i^{th} topic is active. It publishes the value if one of 3 conditions are met: if there is no other active topic (i.e. allowed =VACANT), or if the topic is already in an allowed state (i.e. allowed $= i$), or if the topic has higher priority than the currently allowed topic (i.e. $priority(i) > priority(\mathsf{allowed})$). The callback timerCallback() for the i^{th} topic, based on a timeout, sets $active(i)$ to false and sets allowed to VACANT when this is the currently allowed topic.

We now formulate a desired property for the Multiplexer (specified in UPPAAL in Appendix C), stating that the RandomWalker can send messages to the engine.

$$\exists\Diamond\ \mathsf{Random\_cmdVelCallback}.\mathit{PassMsg} \qquad \textbf{(MUX-Property)}$$

Here Random_cmdVelCallback is the cmdVelCallback() automata that analyses messages from the random walker.

By experimenting with different parameters, we observe that the model does not satisfy this property for higher publishing rates, higher priority of SafetyController-Publisher and higher values of *TimeOut(i)*. This means that *CmdVel* messages from the RandomWalker component may never reach the Kobuki base-motor if messages from the (higher priority) safety controller are published frequently enough.

6. CONCLUSIONS

This paper proposes a generic approach to model-check real-time properties of ROS-based applications using timed automata, with special focus on the communication between nodes. This approach allows to verify safety and liveness properties of complex ROS-based robots that could be influenced by various architectural parameters, such as queue sizes and internal timeouts. We use the UPPAAL model checker to model ROS applications and to verify real-time properties, and illustrate our approach by analysing the source code of a popular physical robot Kobuki. This model is then used to guide the search for parameters that can validate some desired properties of Kobuki, such as not losing sensor messages nor ignoring movement instructions.

7. REFERENCES

[1] S. Adam, M. Larsen, K. Jensen, and U. P. Schultz. Towards rule-based dynamic safety monitoring for mobile robots. In *SIMPAR*, volume 8810 of *LNCS*, pages 207–218. Springer, 2014.

[2] A. Anand and R. A. Knepper. ROSCoq: Robots powered by constructive reals. In *ITP*, volume 9236 of *LNCS*, pages 34–50. Springer, 2015.

[3] C. Baier and J. Katoen. *Principles of Model Checking*. MIT Press, 2008.

[4] R. Barbuti and L. Tesei. Timed automata with urgent transitions. *Acta Informatica*, 40(5):317–347, 2004.

[5] G. Behrmann, A. David, and K. G. Larsen. A tutorial on Uppaal. In *SFM*, volume 3185 of *LNCS*, pages 200–236. Springer, 2004.

[6] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *LNCS*, pages 87–124. Springer, 2003.

[7] A. Cowley and C. J. Taylor. Towards language-based verification of robot behaviors. In *IROS*, pages 4776–4782. IEEE, 2011.

[8] J. Huang, C. Erdogan, Y. Zhang, B. M. Moore, Q. Luo, A. Sundaresan, and G. Rosu. ROSRV: runtime verification for robots. In *RV*, volume 8734 of *LNCS*, pages 247–254. Springer, 2014.

[9] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. ROS: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[10] A. Santos, N. Macedo, A. Cunha, and C. Lourenço. A framework for quality assessment of ROS repositories. In *IROS*, pages 4491–4496. IEEE, 2016.

[11] M. Webster, C. Dixon, M. Fisher, M. Salem, J. Saunders, K. L. Koay, K. Dautenhahn, and J. Saez-Pons. Toward reliable autonomous robotic assistants through formal verification: A case study. *IEEE Trans. Human-Machine Systems*, 46(2):186–196, 2016.

[12] M. Weißmann, S. Bedenk, C. Buckl, and A. Knoll. Model checking industrial robot systems. In *SPIN*, volume 6823 of *LNCS*, pages 161–176. Springer, 2011.

APPENDIX

A. THE PUBLISH-SUBSCRIBER IN UPPAAL

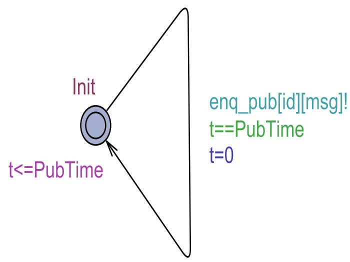
The UPPAAL automata corresponding to the timed automata in Figure 3 are depicted in Figure 7.

B. THE SAFETYCONTROLLER-UPDATE IN UPPAAL

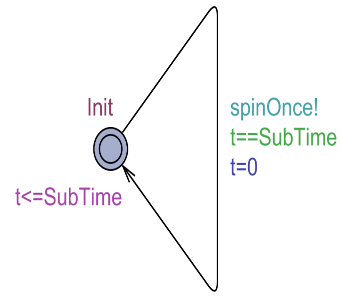
The Uppaal implementation of the timed model of the module SafetyController-Update is shown in Figure 8. Observe that variables SIZE and MsgCount represent the size of the subscriber queue and the number of messages currently present in the queue, to capture the queue's fullness and emptiness conditions.

C. THE MULTIPLEXER MODULE IN UPPAAL

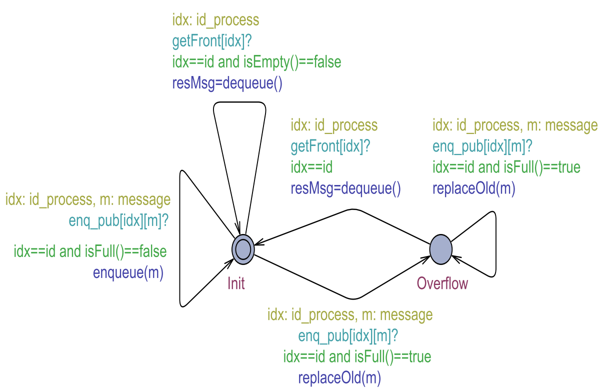
The timed model of the module Multiplexer is implemented in Uppaal and is shown in Figure 9.



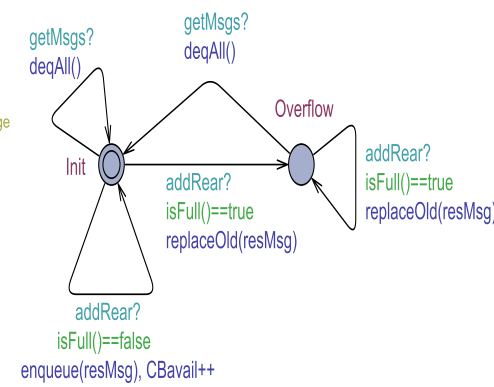
(a) Publishers "P_{id}" ($id = 1, 2$)



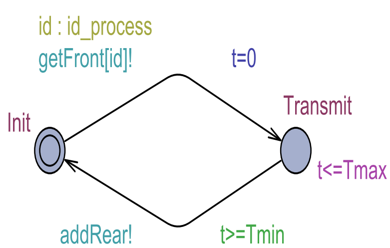
(b) Subscriber "P_3"



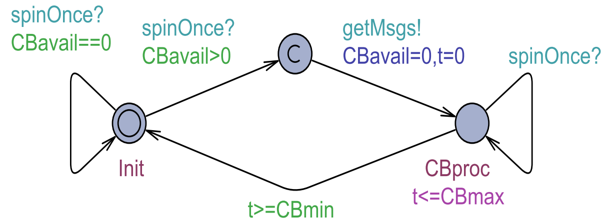
(c) Publisher Queues "$Q_{id \to 1}$"



(d) Subscriber Queue "$Q_{3 \leftarrow 1}$"

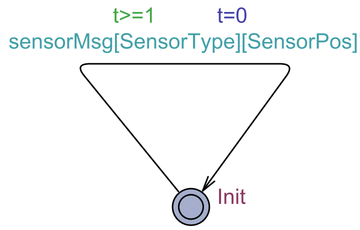


(e) Channel "Ch_1"

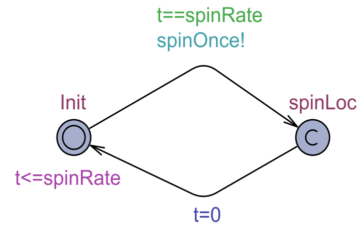


(f) callAvailable() to process callbacks in Callback Queue CBQ

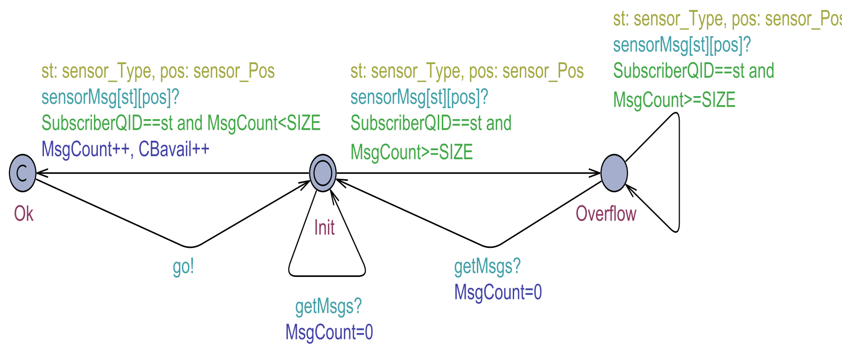
Figure 7: Implementation of the Timed Model of Figure 3 in UPPAAL.



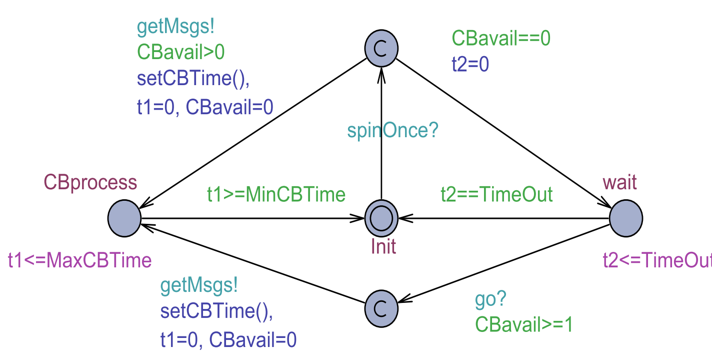
(a) Template for Sensors



(b) SafetyController-Update: safety controller which updates internal state.

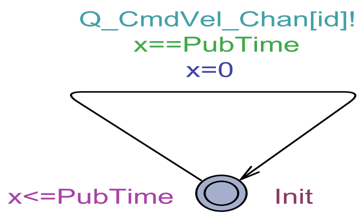


(c) Template for safety controller's Subscriber Queues *QWheel, QBumper,* and *QCliff*.

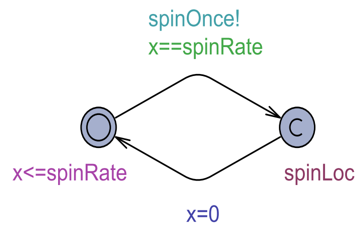


(d) "callAvailable (ros::WallDuration TimeOut)" method to process all callbacks currently in the Callback Queue for Safety-Controller.

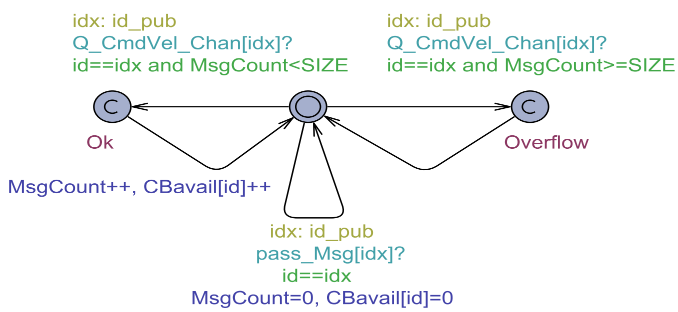
Figure 8: Implementation of the module SafetyController-Update using Uppaal Model Checker



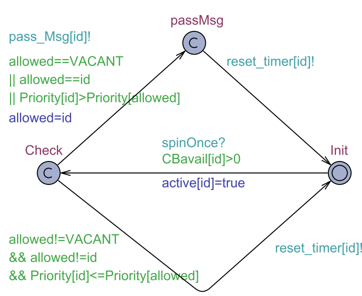
(a) Template for SafetyController-Publisher and RandomWalker who publish Command Velocity Messages



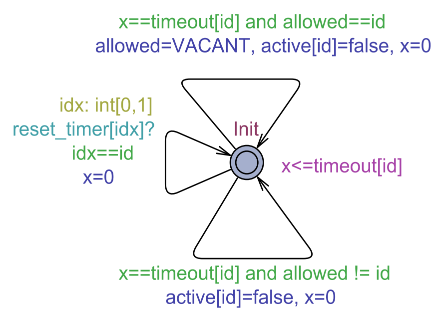
(b) Multiplexer node



(c) Template for Subscriber Queues which are subscribed by Multiplexer



(d) Template for Callback cmdVelCallback() for two subscriptions (denoted Safety_cmdVelCallback and Random_cmdVelCallback resp.)



(e) Template for Callback timerCallback() for two subscriptions (denoted **Safety_timerCallback** and **Random_timerCallback** resp.)

Figure 9: Uppaal Model for the module Multiplexer