

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

Raju Halder & José Proença & Nuno Macedo & André Santos
(Guilhermina Cledou)



FORMALISE 2017



INESCTEC



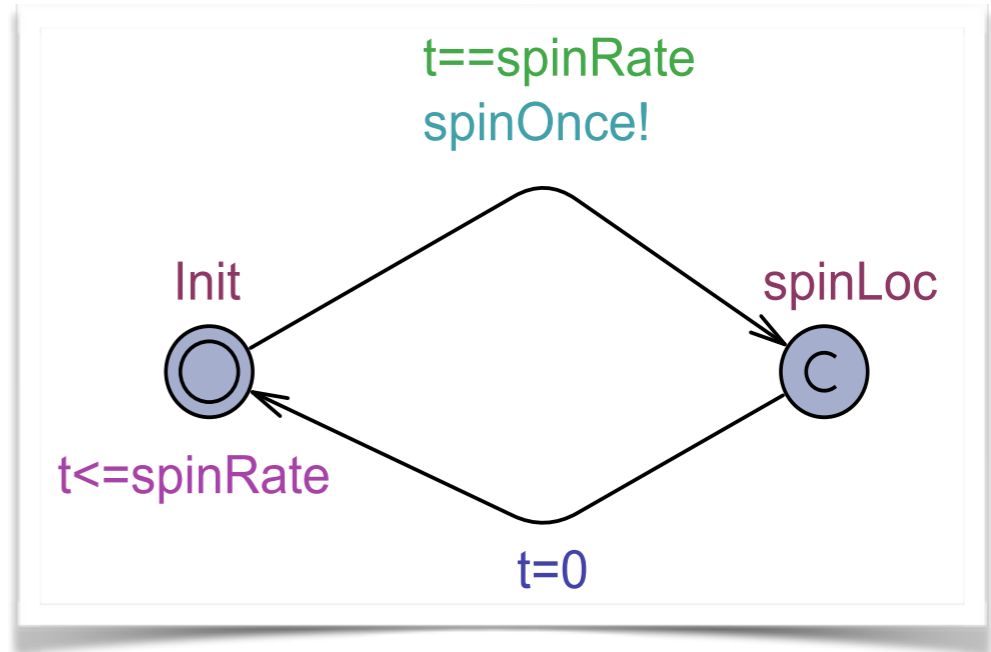
Universidade do Minho

Motivation

Kobuki Robot



ROS inside



Analyse with
Timed-automata

<http://kobuki.yujinrobot.com/>

Outline

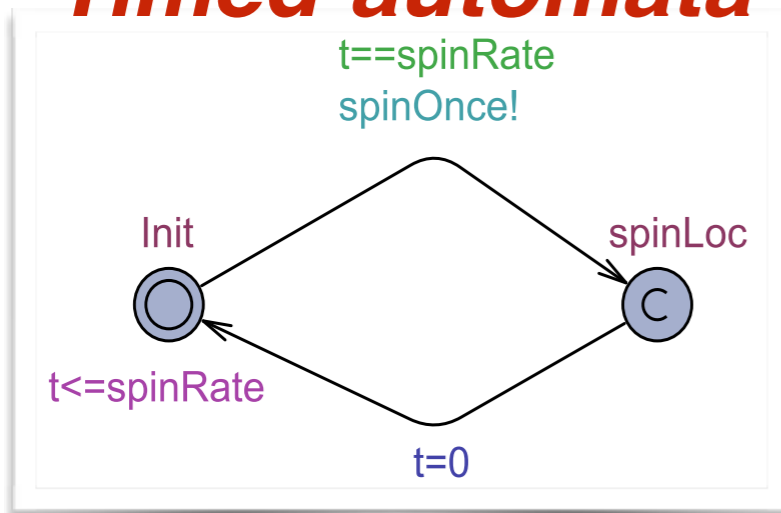


How is it used

Real-time challenges

Starting point of our analysis

Timed-automata



What is it

What we model

What we verify

Why robotics?

- modern robotics are applied in industrial, agricultural, medical and domestic domains
- must be flexible, configurable and adaptive
- ever-closer human-robot interaction

Why robotics?

requires software
controllers

- modern robotics are applied in industrial, agricultural, medical and domestic domains
- must be **flexible, configurable and adaptive**
- intended to have closer **human-robot interaction**

safety verification
is critical

Why ROS?



Why ROS?

- middleware for developing robots
- modular, portable and configurable
- *thousands* of publicly available libraries

ROS Architecture

- Component-based, *nodes* interacting with each other through *topics*
- Synchronous (RPC) and asynchronous (publish-subscribe) communication
- Use of explicit timeouts at application level
- Manually configured message queues and processing rates

ROS Architecture

- Component-based, *nodes* interacting with each other through *topics*
- Synchronous (RPC) and asynchronous (publish-subscribe) communication
- Use of explicit timeouts at application level
- Manually configured message queues and processing rates

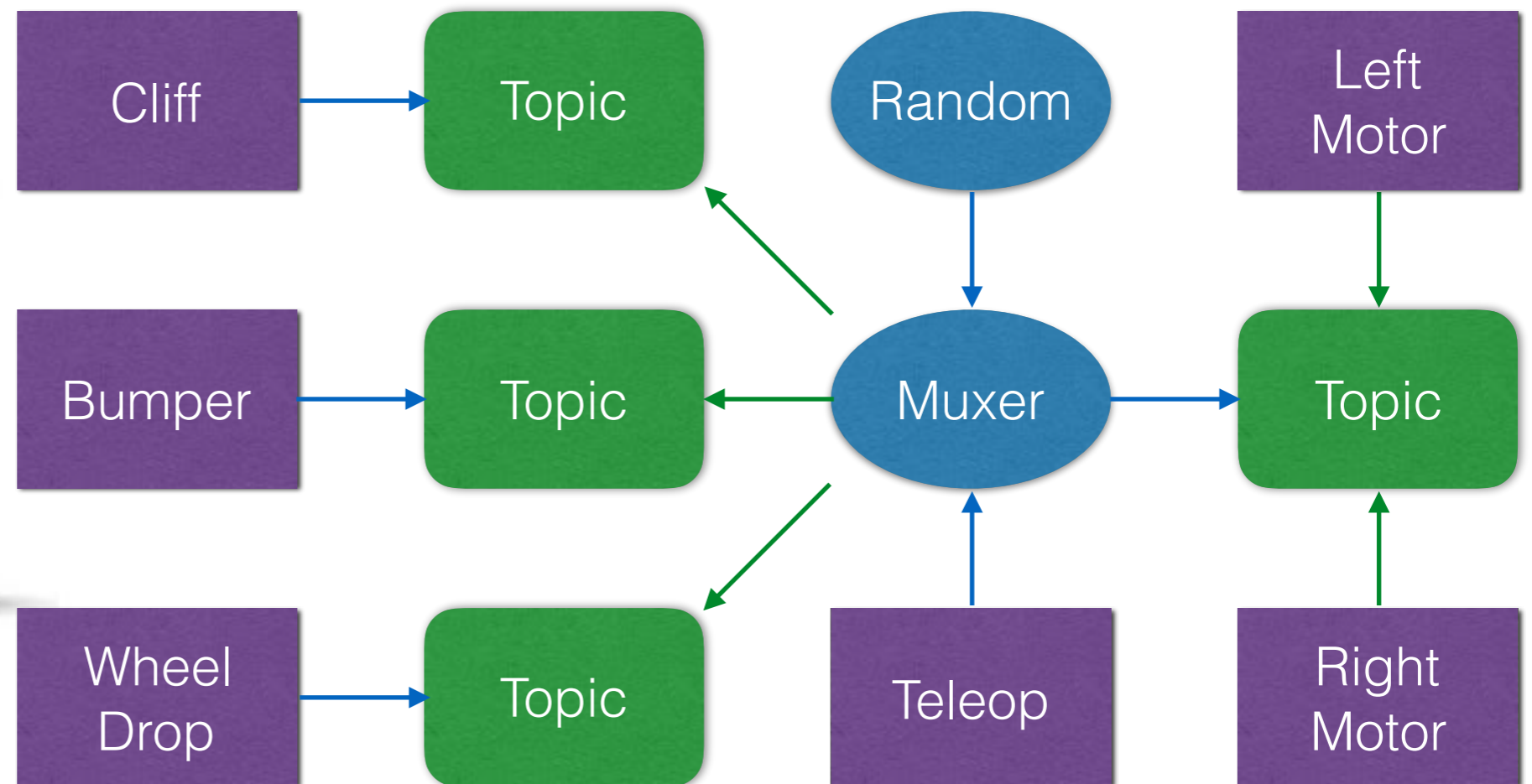


error-prone!

ROS Example



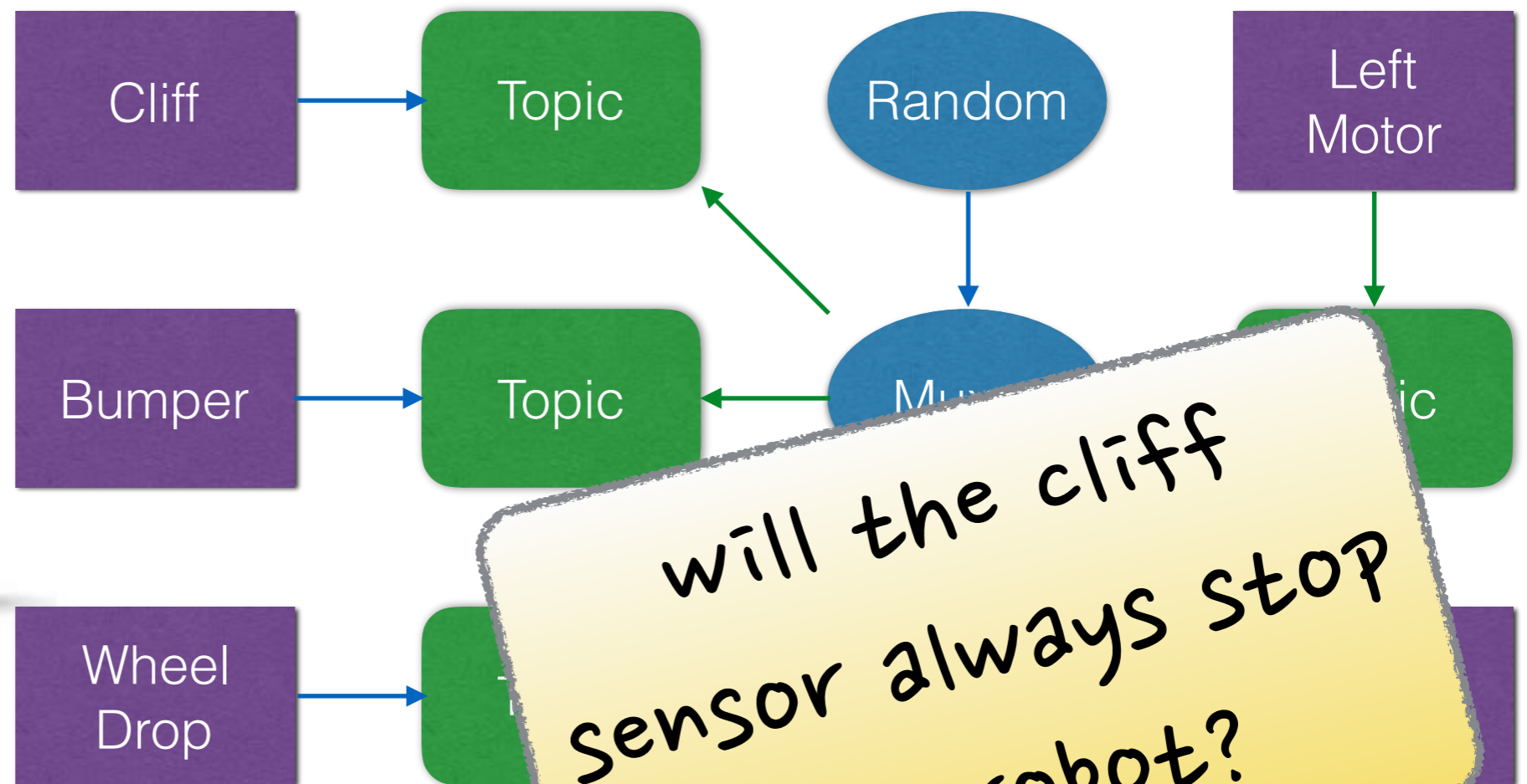
**TurtleBot 2
(Kobuki base)**



ROS Example



**TurtleBot 2
(Kobuki base)**



will the cliff sensor always stop the robot?

Code analysis

Publish/subscribe

```
int main(int argc, char **argv) {
  ros::init(argc, argv, "talker");
  ros::NodeHandle n;
  ros::Publisher chatter_pub =
    n.advertise<std_msgs::String>("chatter",
    1000);
  ros::Rate loop_rate(10);
  while (ros::ok()) {
    std_msgs::String msg;
    //... do some work ...
    chatter_pub.publish(msg);
    loop_rate.sleep();
  }
  return 0;
}
```

```
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;
}
```

Code analysis

Publish/subscribe

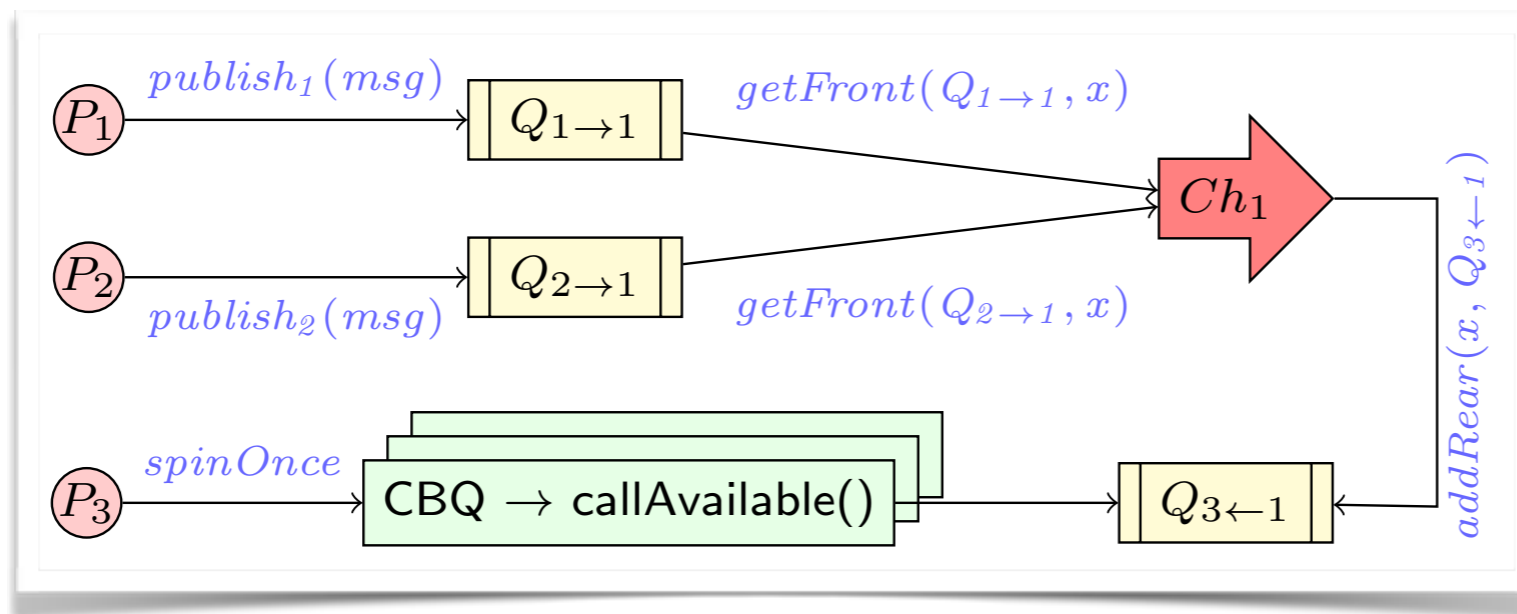
```
int main(int argc, char **argv) {
  ros::init(argc, argv, "talker");
  ros::NodeHandle n;
  ros::Publisher chatter_pub =
    n.advertise<std_msgs::String>("chatter",
    1000);
  ros::Rate loop_rate(10);
  while (ros::ok()) {
    std_msgs::String msg;
    //... do some work ...
    chatter_pub.publish(msg);
    loop_rate.sleep();
  }
  return 0;
}
```

```
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;
}
```

Not done
automatically!

Two examples

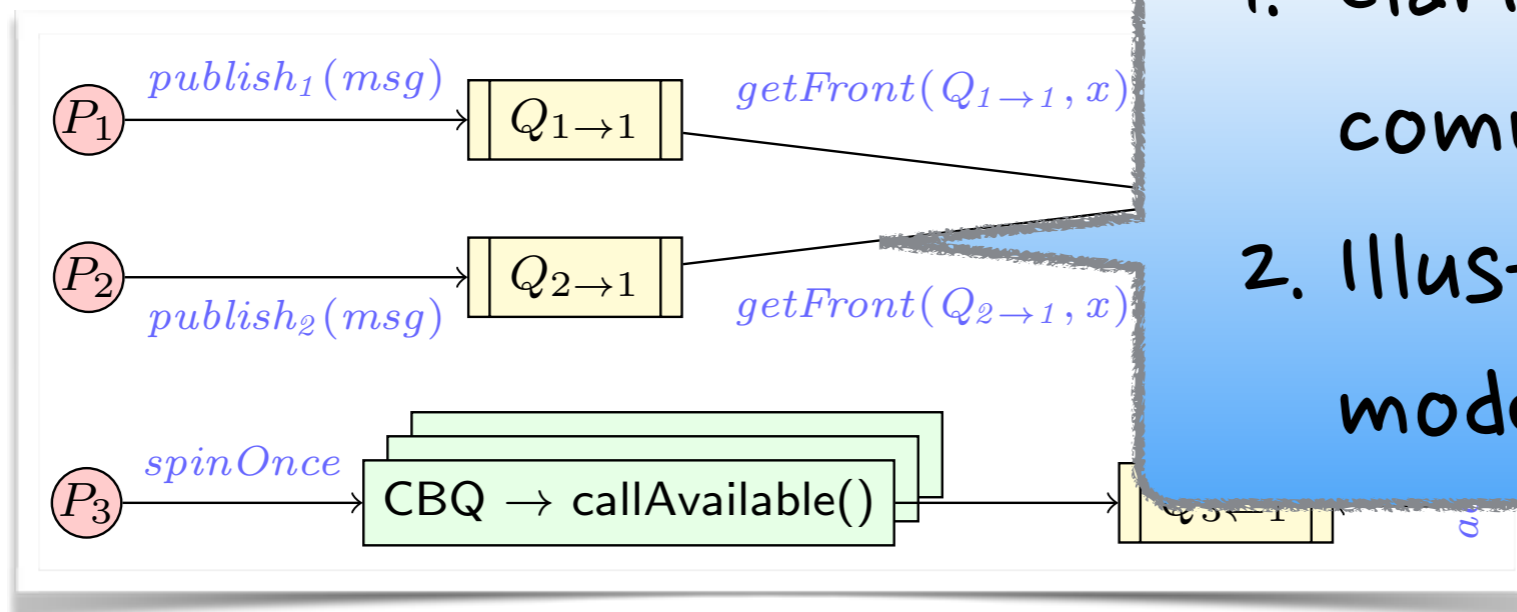


2 publishers
1 subscriber



Safety-controller
combining
safety sensors + random walker

Two examples



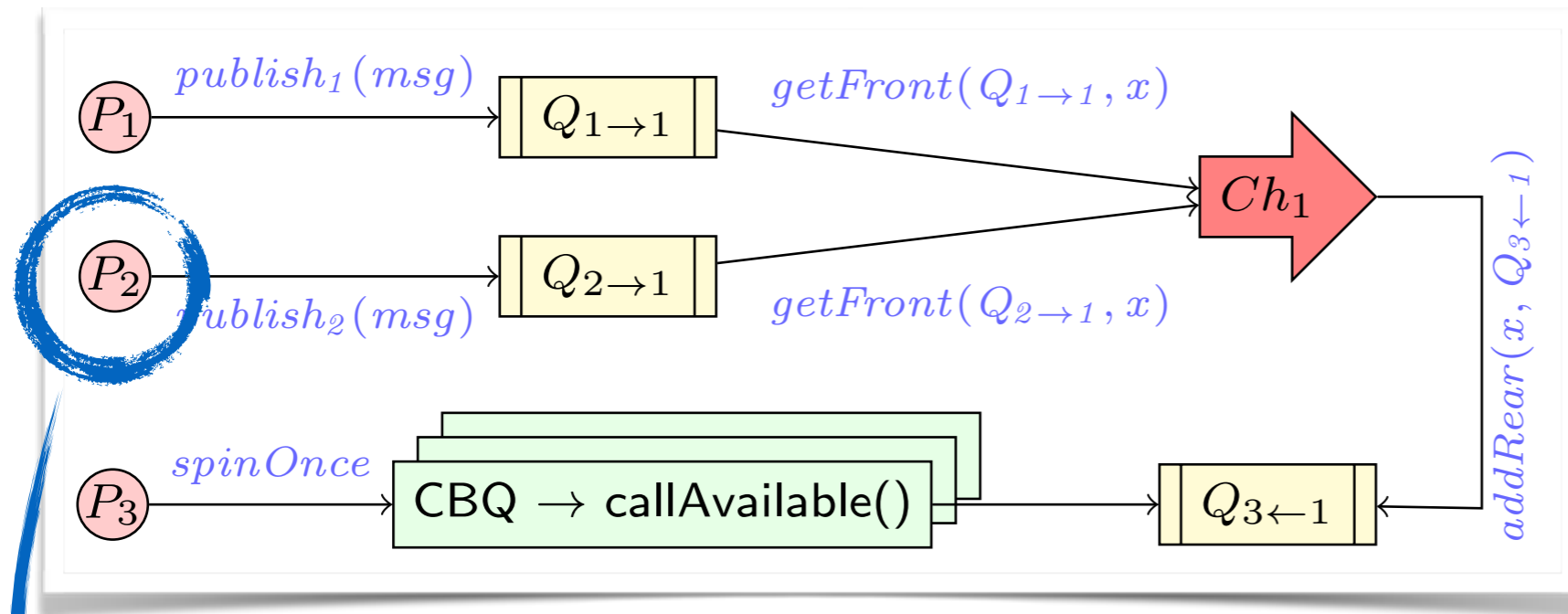
1. clarify how communication works
2. Illustrates how to model with timed-



1. More complex scenario
2. Illustrate what can we learn in a more complex application

safety sensors + random walker

2 Pubs - 1 Subs



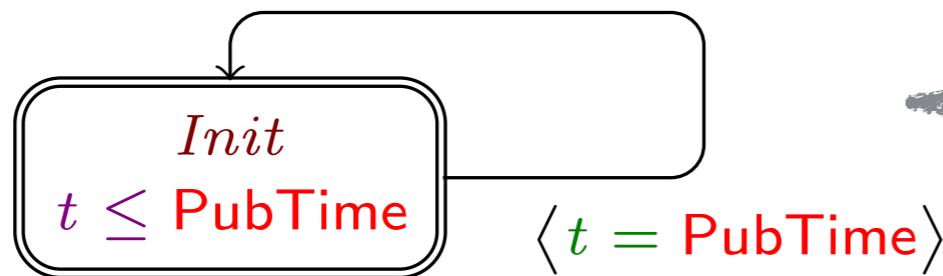
8 automata

communication

=

shared actions
(or shared variables)

$\text{publish}_{id}(msg)!, t := 0$



Timed automaton
- publishes every
"PubTime" seconds

UPPAAL

/Users/jose/Documents/research/uminho/ROS/ROS-formal-se/UPPAAL_Model/Ros_PubSub_Model.xml

Editor Simulator ConcreteSimulator Verifier Yggdrasil

Enabled Transitions

- enq_pub[0][msg]: Pub1 → QueuePub1
- enq_pub[1][msg]: Pub2 → QueuePub1
- addRear: Channel → QueueSub

Next Reset

Simulation Trace

```
getFront[id]: Channel[1] → QueuePub1
(Init, Init, Init, Init, Init, Init, Trans
addRear: Channel → QueueSub
(Init, Init, Init, Init, Init, Init, Init, In
getFront[id]: Channel[0] → QueuePub1
(Init, Init, Init, Init, Init, Init, Trans
```

Trace File:

Prev Next Replay
Open Save Random

Pub1

Init

$t \leq P1rate$

$enq_pub[0][msg]!$
 $t = P1rate$
 $t = 0$

Pub2

Init

$t \leq P2rate$

$enq_pub[1][msg]!$
 $t = P2rate$
 $t = 0$

Sub

Init

$t \leq SubTime$

$spinOnce!$
 $t = SubTime$
 $t = 0$

QueuePub1

$idx: id_process$
 $getFront[idx]?$
 $idx == 0$ and $isEmpty() == false$
 $resMsg = dequeue()$

$idx: id_process, m: message$
 $enq_pub[idx][m]?$
 $idx == 0$ and $isFull() == false$
 $enqueue(m)$

Init

$idx: id_process, m: message$
 $enq_pub[idx][m]?$
 $idx == 0$ and $isFull() == true$
 $replaceOld(m)$

$idx: id_process, m: message$
 $getFront[idx]?$
 $idx == 0$
 $resMsg = dequeue()$

$idx: id_process, m: message$
 $enq_pub[idx][m]?$
 $idx == 0$ and $isFull() == true$
 $replaceOld(m)$

Overflow

QueuePub2

$idx: id_process$

UPPAAL

Simulate
(discover traces)

The screenshot displays the UPPAAL simulator interface. At the top, the file path is `es/jose/Documents/research/uminho/ROS/ROS-formal-se/UPPAAL_Model/Ros_PubSub_Model.xml`. The interface includes a toolbar with navigation icons and a menu bar with options: Editor, Simulator, ConcreteSimulator, Verifier, and Yggdrasil.

On the left side, there are two main panels:

- Enabled Transitions:** A list of transitions with their guard conditions and actions:
 - `enq_pub[0][msg]: Pub1 → QueuePub1`
 - `enq_pub[1][msg]: Pub2 → QueuePub1`
 - `addRear: Channel → QueueSub`Below this list are "Next" and "Reset" buttons.
- Simulation Trace:** A log of simulation events:
 - `getFront[id]: Channel[1] → QueuePub1`
 - `(Init, Init, Init, Init, Init, Init, Trans)`
 - `addRear: Channel → QueueSub`
 - `(Init, Init, Init, Init, Init, Init, Init, Tr)`
 - `getFront[id]: Channel[0] → QueuePub1`
 - `(Init, Init, Init, Init, Init, Init, Trans)`Below the trace are buttons for "Prev", "Next", "Replay", "Open", "Save", and "Random".

The main workspace shows several automata diagrams:

- Pub1:** A diagram with an initial state `Init` and a transition labeled `enq_pub[0][msg]!` with guard `t == P1rate` and action `t = 0`. A timer constraint `t <= P1rate` is shown.
- Pub2:** A diagram with an initial state `Init` and a transition labeled `enq_pub[1][msg]!` with guard `t == P2rate` and action `t = 0`. A timer constraint `t <= P2rate` is shown.
- Sub:** A diagram with an initial state `Init` and a transition labeled `spinOnce!` with guard `t == SubTime` and action `t = 0`. A timer constraint `t <= SubTime` is shown.
- QueuePub1:** A diagram with an initial state `Init` and a transition labeled `enq_pub[idx][m]?` with guard `idx == 0 and isEmpty() == false` and action `resMsg = dequeue()`. It also has a transition labeled `enq_sub[idx][m]?` with guard `idx == 0 and isFull() == false` and action `enqueue(m)`. A transition labeled `replaceOld(m)` has guard `idx == 0 and isFull() == true` and action `replaceOld(m)`. A transition labeled `Overflow` is also present.
- QueuePub2:** A diagram with an initial state `Init` and a transition labeled `enq_sub[idx][m]?` with guard `idx == 0 and isFull() == true` and action `replaceOld(m)`.

A blue speech bubble on the right side of the interface contains the text: "run and view 8 automata in parallel".

UPPAAL

The screenshot shows the UPPAAL Verifier interface. The window title is `/Users/jose/Documents/research/uminho/ROS/ROS-formalise/UPPAAL_Model/Ros`. The interface includes a toolbar with icons for file operations and navigation. Below the toolbar are tabs for `Editor`, `Simulator`, `ConcreteSimulator`, `Verifier` (selected), and `Yggdrasil`.

The `Overview` section displays a list of properties with their verification status indicated by colored circles:

Property	Status
<code>A[] not QueuePub1.Overflow</code>	Green
<code>A[] not QueuePub2.Overflow</code>	Green
<code>A[] not QueueSub.Overflow</code>	Red
<code>A[] not deadlock</code>	Green

Buttons for `Check`, `Insert`, `Remove`, and `Comments` are located to the right of the list.

The `Query` section contains the text `A[] not deadlock`.

The `Comment` section is an empty text box.

The `Status` section at the bottom provides performance metrics:

Verification/kernel/elapsed time used: 0.048s / 0.006s / 0.06s.
Resident/virtual memory usage peaks: 9,620KB / 2,503,008KB.
Property is satisfied.

We can check
temporal
properties

UPPAAL

The screenshot shows the UPPAAL Verifier interface. The 'Verifier' tab is active. In the 'Overview' section, a list of properties is shown with corresponding status indicators (green for satisfied, red for failed). The property 'A[] not QueueSub.Overflow' is highlighted in blue and has a red indicator, indicating it has failed. To the right of the list are buttons for 'Check', 'Insert', 'Remove', and 'Comments'. Below the overview is a 'Query' field containing 'A[] not'. At the bottom, the 'Status' section shows verification details and the message 'Property is satisfied.'.

Property	Status
A[] not QueuePub1.Overflow	Green
A[] not QueuePub2.Overflow	Green
A[] not QueueSub.Overflow	Red
A[] not deadlock	Green

Verification/kernel/elapsed time used: 0.048s / 0.006s / 0.06s.
Resident/virtual memory usage peaks: 9,620KB / 2,503,008KB.
Property is satisfied.

A [] (not QueueSub.Overflow) failed:
the queue of the subscriber can overflow

Experimenting with parameters

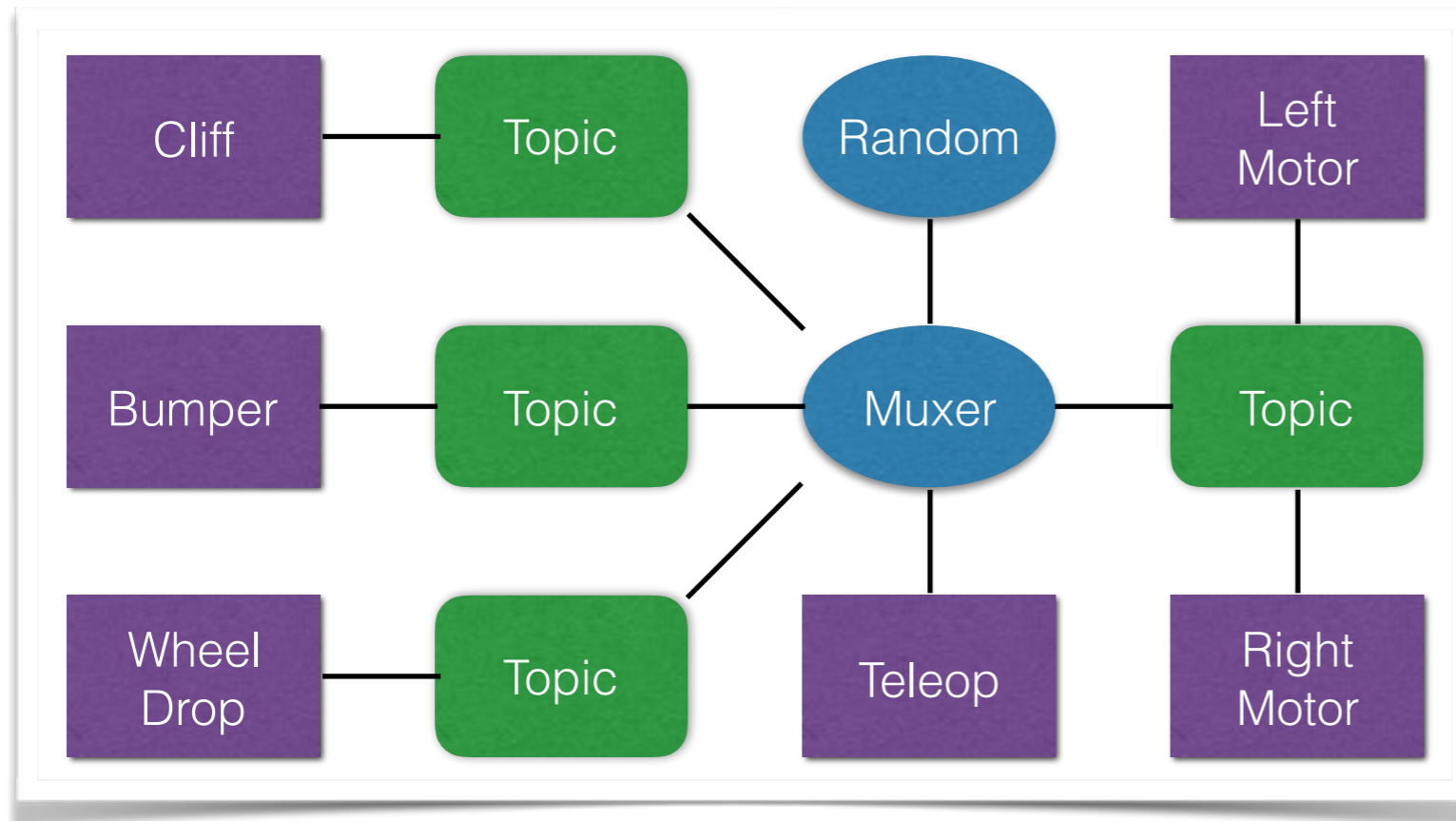
Queue Sizes			Transmission Time		Callback Time		Publishing Time-gap		Spin Time-gap	Properties		
$Q_{1 \rightarrow 1}$	$Q_{2 \rightarrow 1}$	$Q_{3 \leftarrow 1}$	Tmin	Tmax	CBmin	CBmax	PubTime (P_1)	PubTime (P_2)	SubTime (P_3)	Pr ₁	Pr ₂	Pr ₃
5	5	5	1	2	1	2	4	3	4	X	X	✓
					4	5	4	3	5	X	X	X
					4	5	4	4	9	✓	✓	X
			3	4	1	2	8	7	15	X	X	X
					1	2	8	8	17	✓	✓	✓
					1	2	8	8	18	✓	✓	X
10	10	10	1	2	1	2	3	4	9	X	X	✓
					3	4	3	4	10	X	X	X
					4	5	4	4	18	✓	✓	X
			4	5	3	4	3	4	9	X	X	✓
					3	4	3	4	10	X	X	X
					4	4	4	4	18	✓	✓	X

List desired properties

Find the good combinations of parameters

(Manual process)

Repeat approach for large example



Find desired properties

Experiment with parameters

Find out exactly when:

- Sensor messages get lost (too many)
- Remote never manages to go through (sensors have priority)

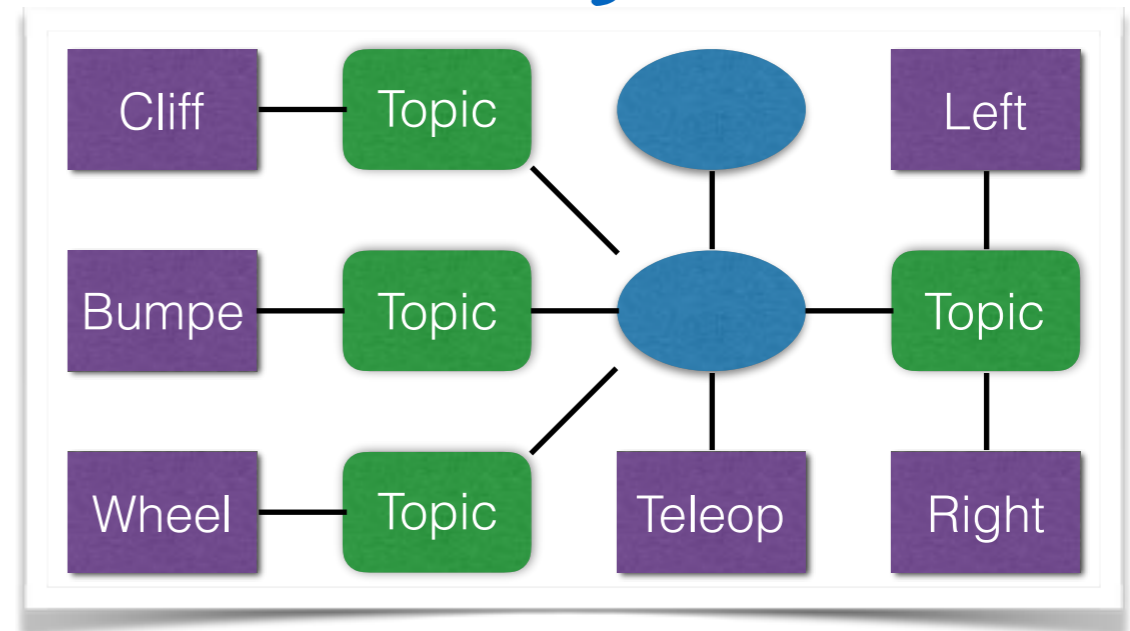
Wrapping up

ROS program



communicating components

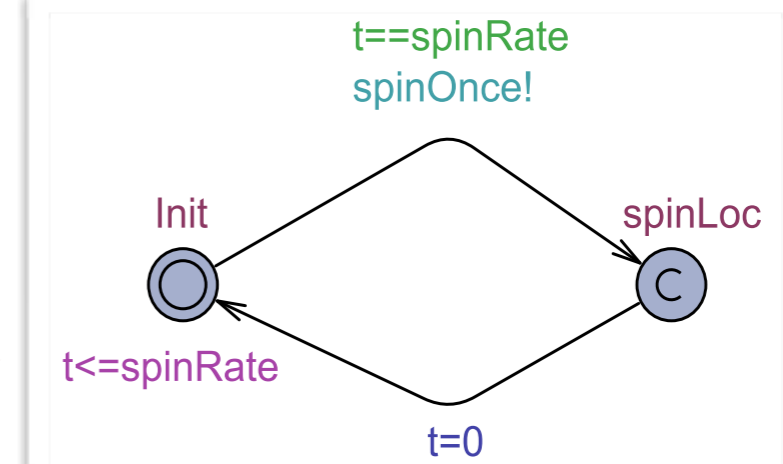
Extract



model as

timed automata

Find parameters that obey timed properties



Wrapping up

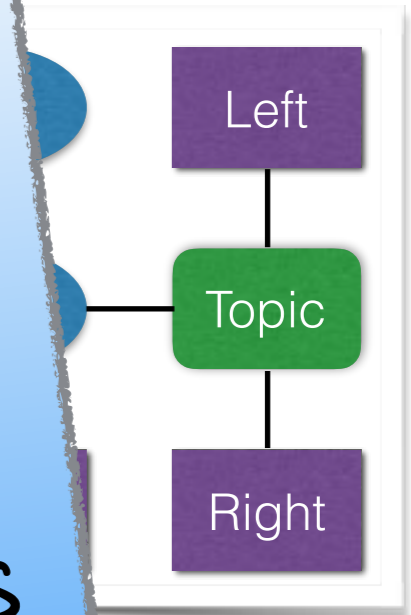
ROS program



Future work:

- Automatic code analysis
- Automatic search for optimal parameters
- ROS as software product lines of components

Components



data

Find parameters that obey timed properties

