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

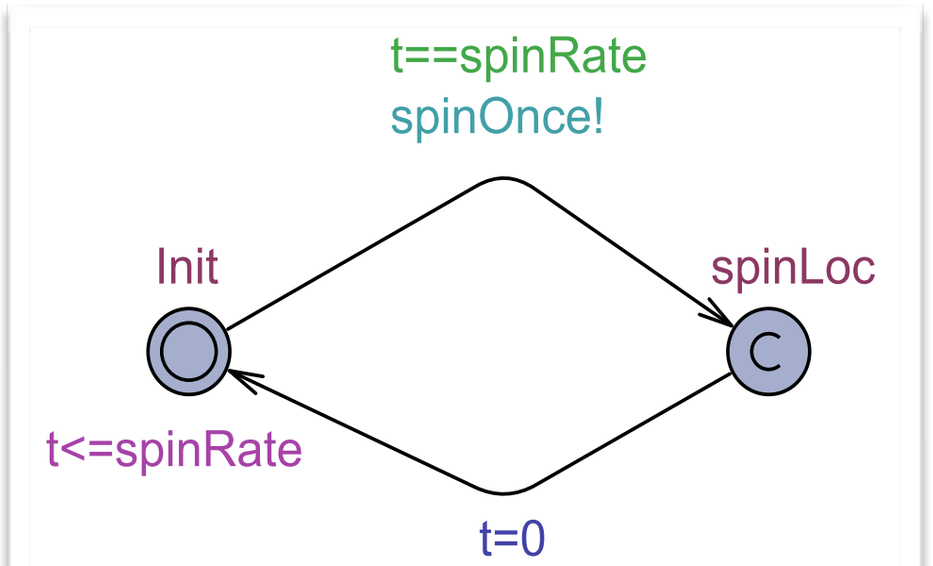
Raju Halder & José Proença & Nuno Macedo & André Santos
(Guillermina Cledou)

FORMALISE 2017

Universidade do Minho

Motivation

Kobuki Robot



ROS inside

Analyse with Timed-automata

http://kobuki.yujinrobot.com/

Outline

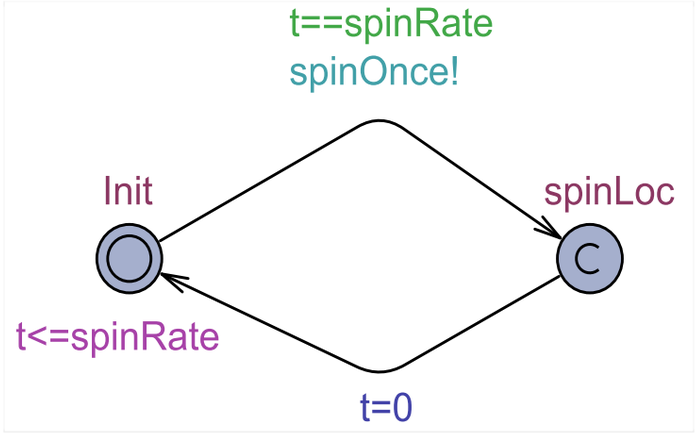
ROS

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

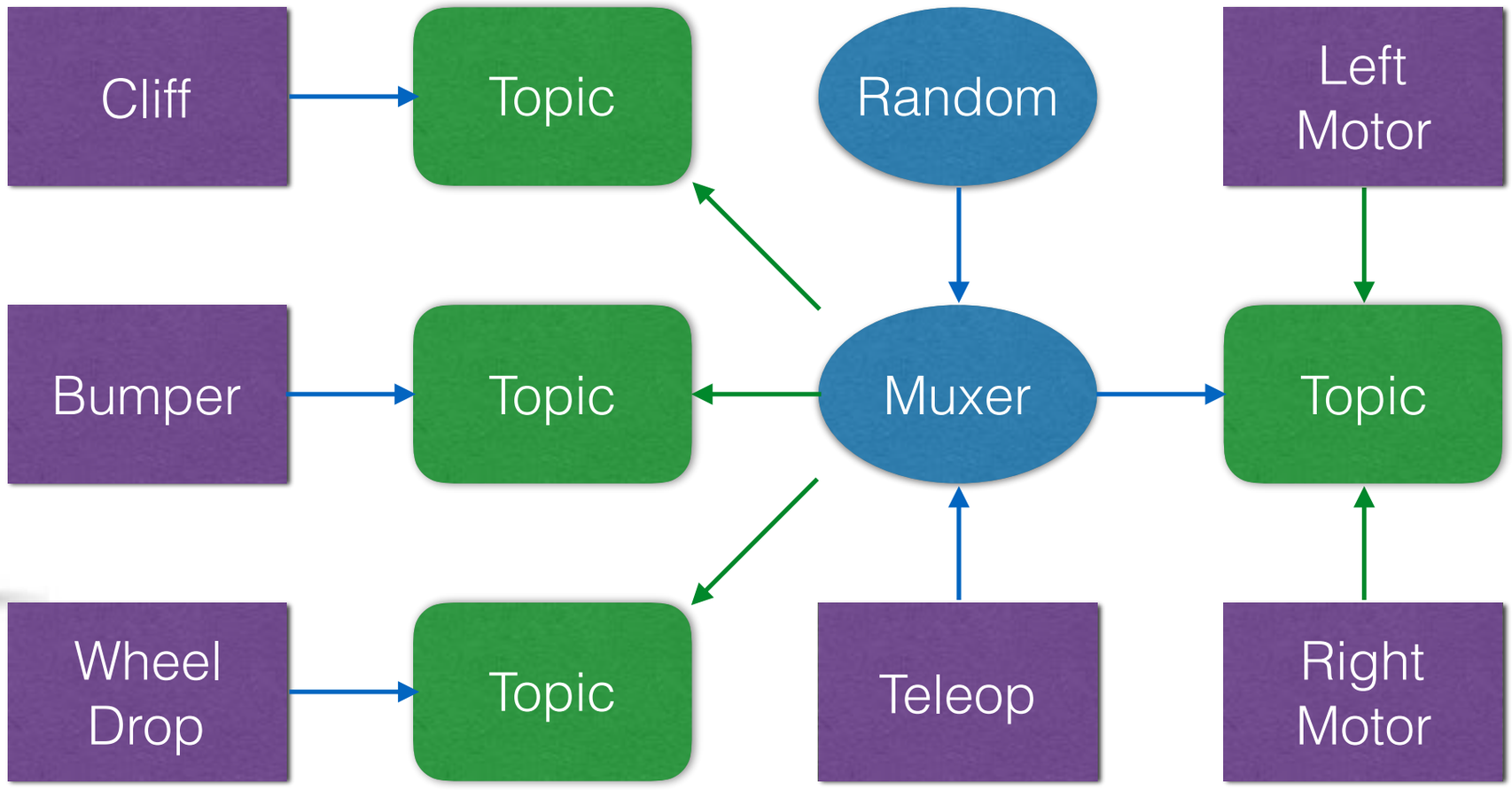
- Component-based, *nodes* interacting with each other through *topics*
- Synchronous (RPC) and asynchronous (publish-subscribe) communication
- Use of explicit timeouts at application level
- Manually configured message queues and processing rates

ROS Architecture

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

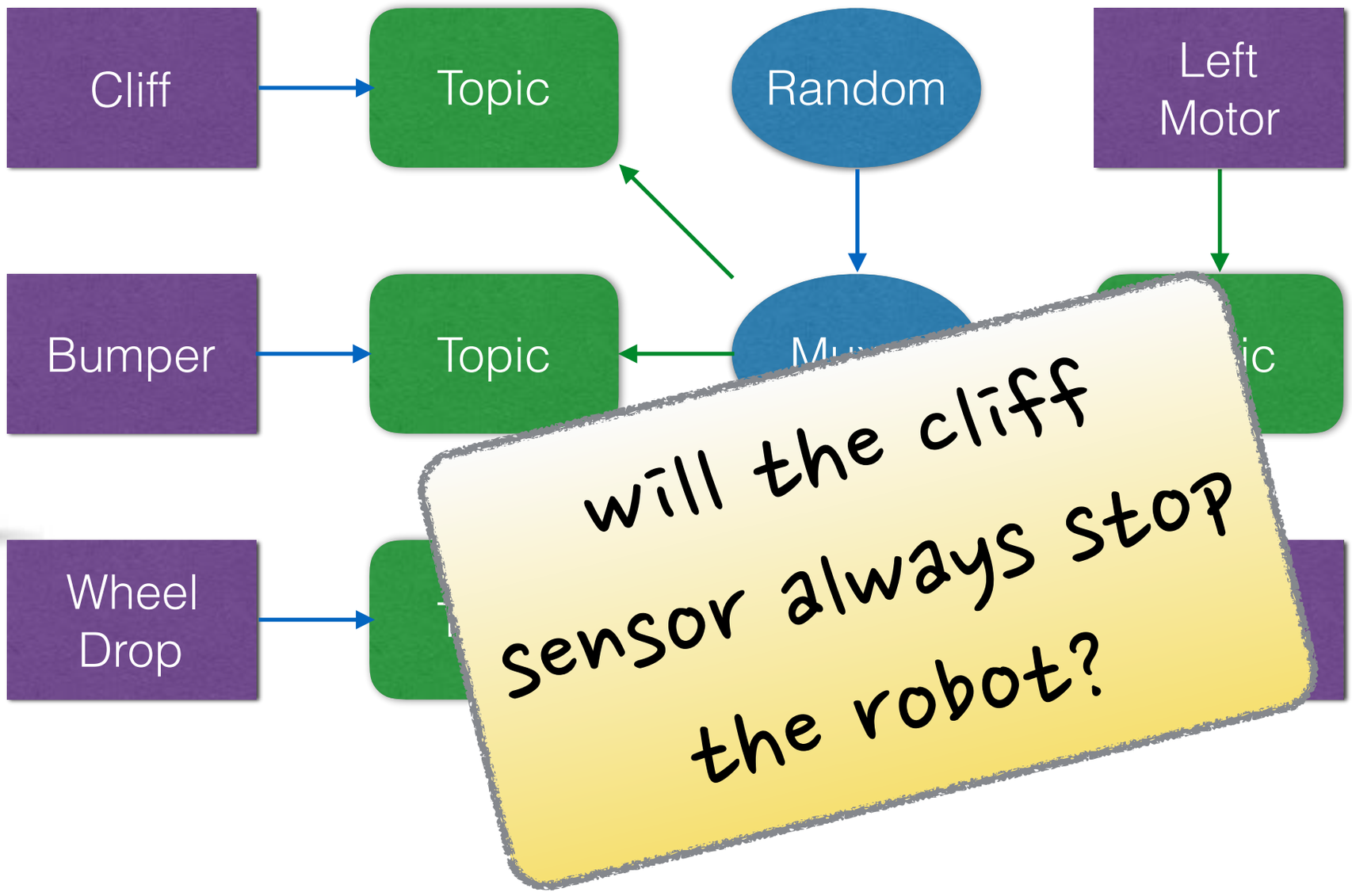
ROS Example

TurtleBot 2
(Kobuki base)



ROS Example

TurtleBot 2
(Kobuki base)



Code analysis

```
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;
}
```

Publish/subscribe

```
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

```cpp
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;
}
```

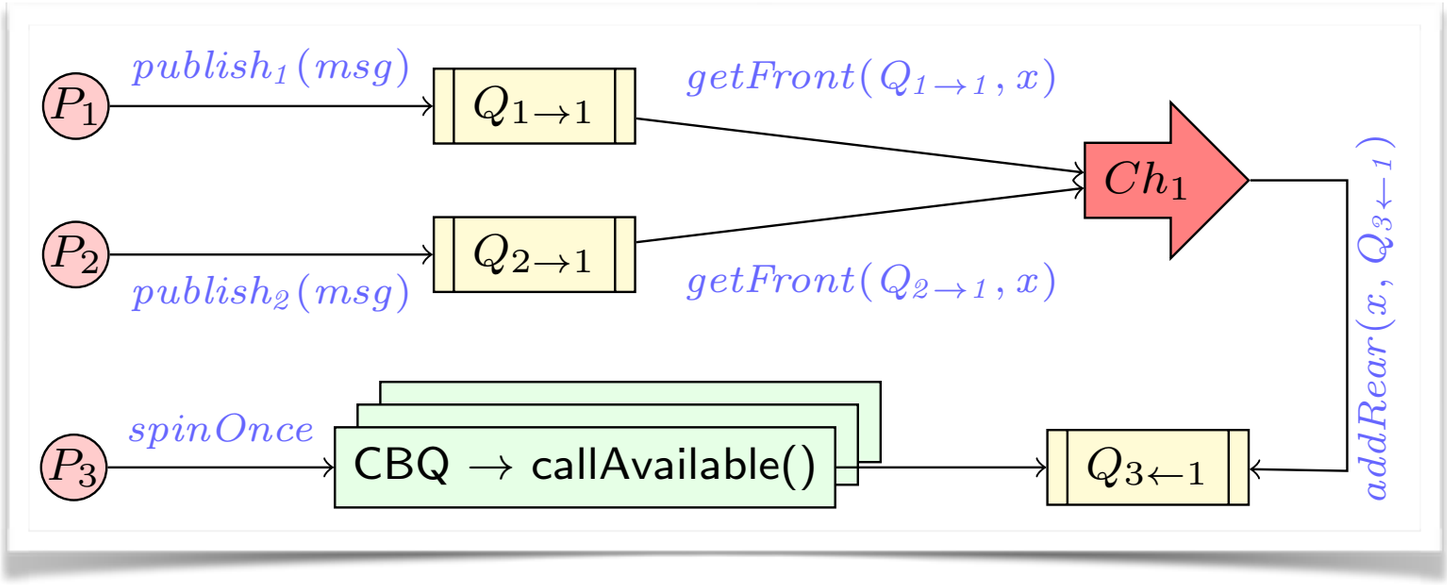
Publish/subscribe

```cpp
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

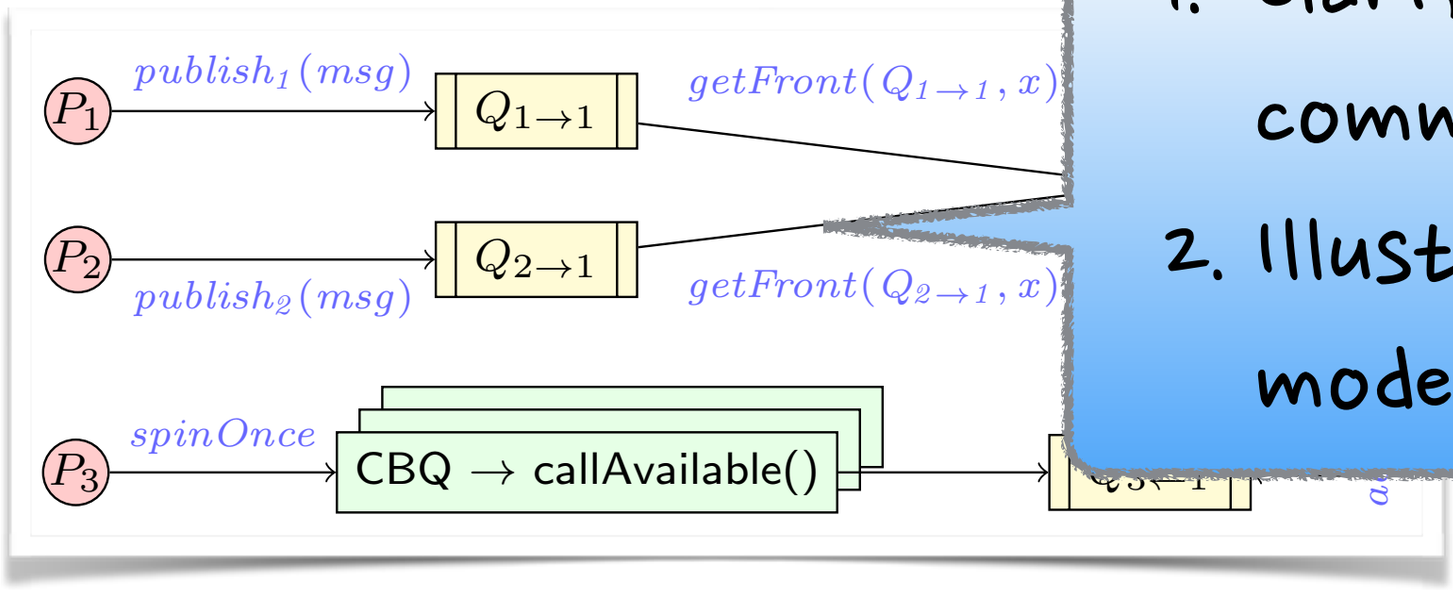
1 subscriber

Safety-controller

combining

safety sensors + random walker

Two examples

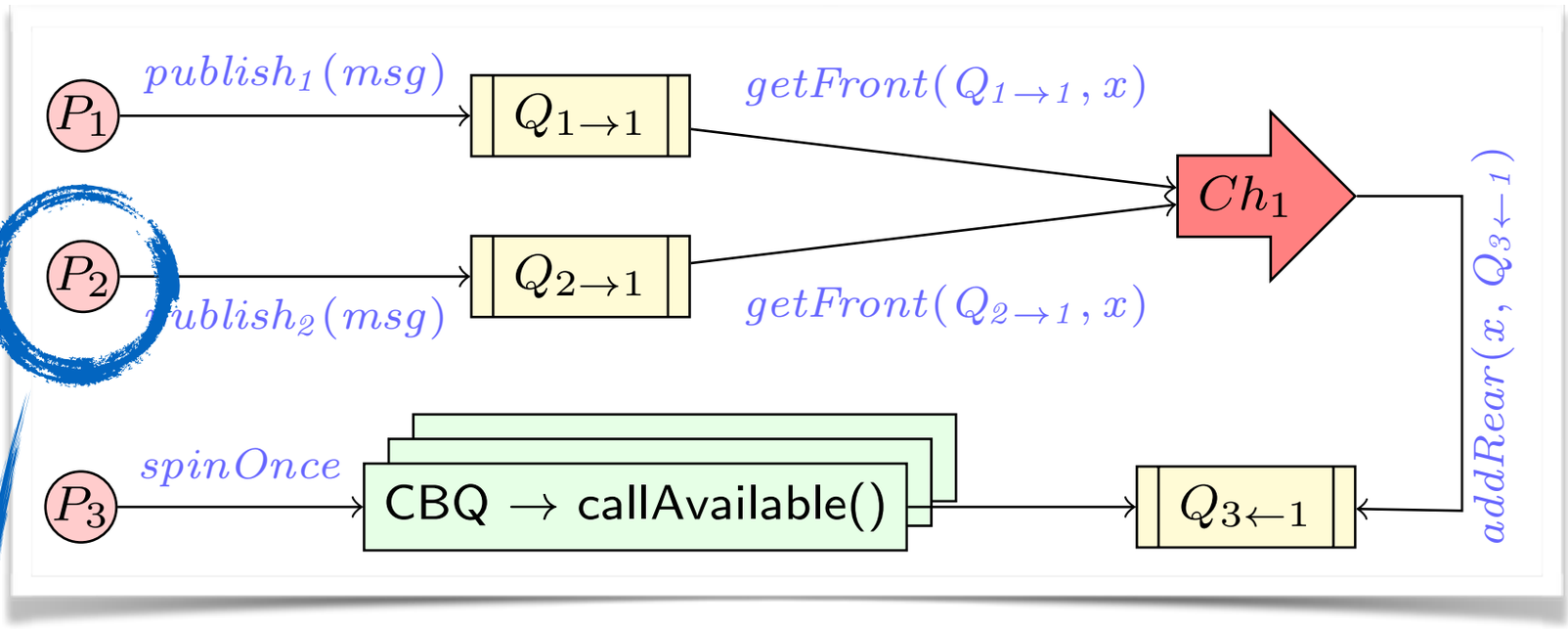


1. clarify how communication works
2. Illustrates how to model with timed-

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

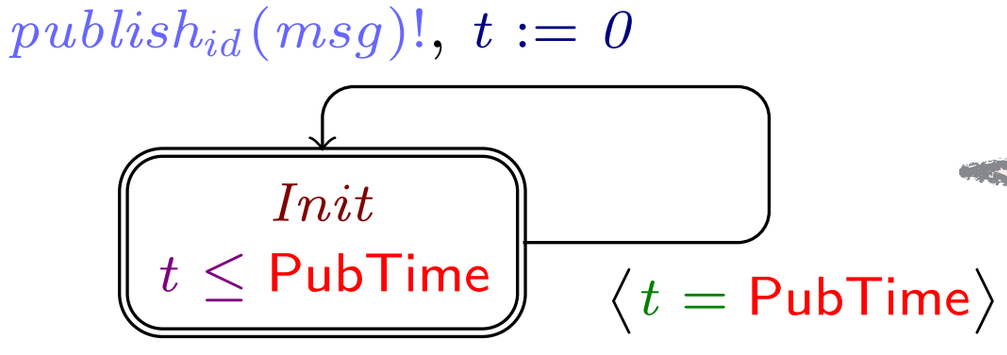
safety sensors + random walker

2 Pubs - 1 Subs



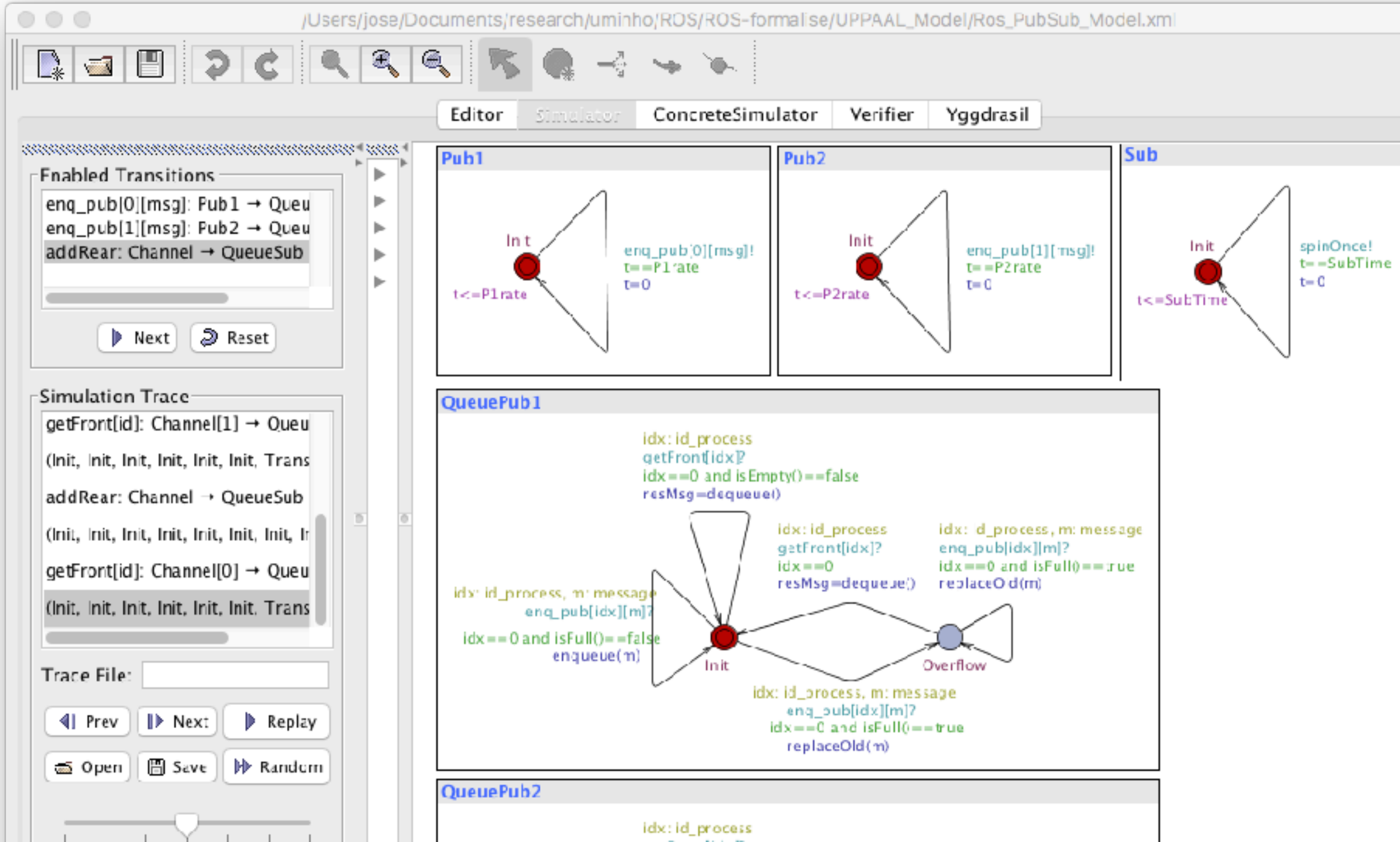
8 automata

communication
=
shared actions
(or shared variables)



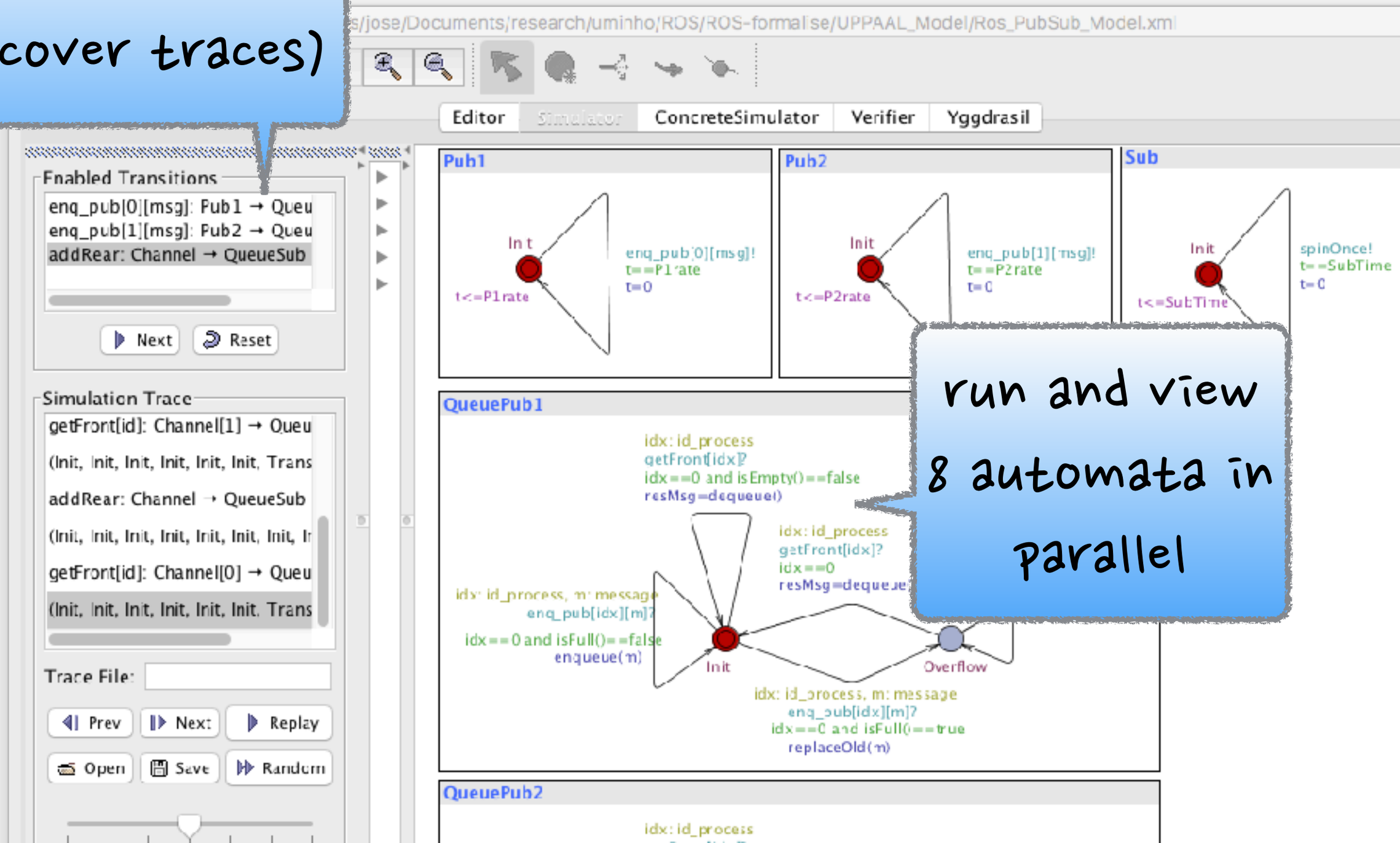
Timed automaton
- publishes every
"PubTime" seconds

UPPAAL


/Users/jose/Documents/research/uminho/ROS/ROS-formalise/UPPAAL_Model/Ros_PubSub_Model.xml
Editor
Simulator
ConcreteSimulator
Verifier
Yggdrasil
Enabled Transitions
enq_pub[0][msg]: Pub1 → Queu
enq_pub[1][msg]: Pub2 → Queu
addRear: Channel → QueueSub
Next
Reset
Simulation Trace
getFront[id]: Channel[1] → Queu
(Init, Init, Init, Init, Init, Init, Trans
addRear: Channel → QueueSub
(Init, Init, Init, Init, Init, Init, Init, I
getFront[id]: Channel[0] → Queu
(Init, Init, Init, Init, Init, Init, Trans
Trace File:
Prev
Next
Replay
Open
Save
Random
Pub1
Init
t<=P1rate
enq_pub[0][msg]!
t==P1rate
t=0
Pub2
Init
t<=P2rate
enq_pub[1][msg]!
t==P2rate
t=0
Sub
Init
t<=SubTime
spinOnce!
t==SubTime
t=0
QueuePub1
idx: id_process
getFront[idx]?
idx==0 and isEmpty()==false
resMsg=dequeue()
idx: id_process
getFront[idx]?
idx==0
resMsg=dequeue()
idx: id_process, m: message
enq_pub[idx][m]?
idx==0 and isFull()==true
replaceOld(m)
idx: id_process, m: message
enq_pub[idx][m]?
idx==0 and isFull()==false
enqueue(m)
Init
Overflow
idx: id_process, m: message
enq_pub[idx][m]?
idx==0 and isFull()==true
replaceOld(m)
QueuePub2
idx: id_process

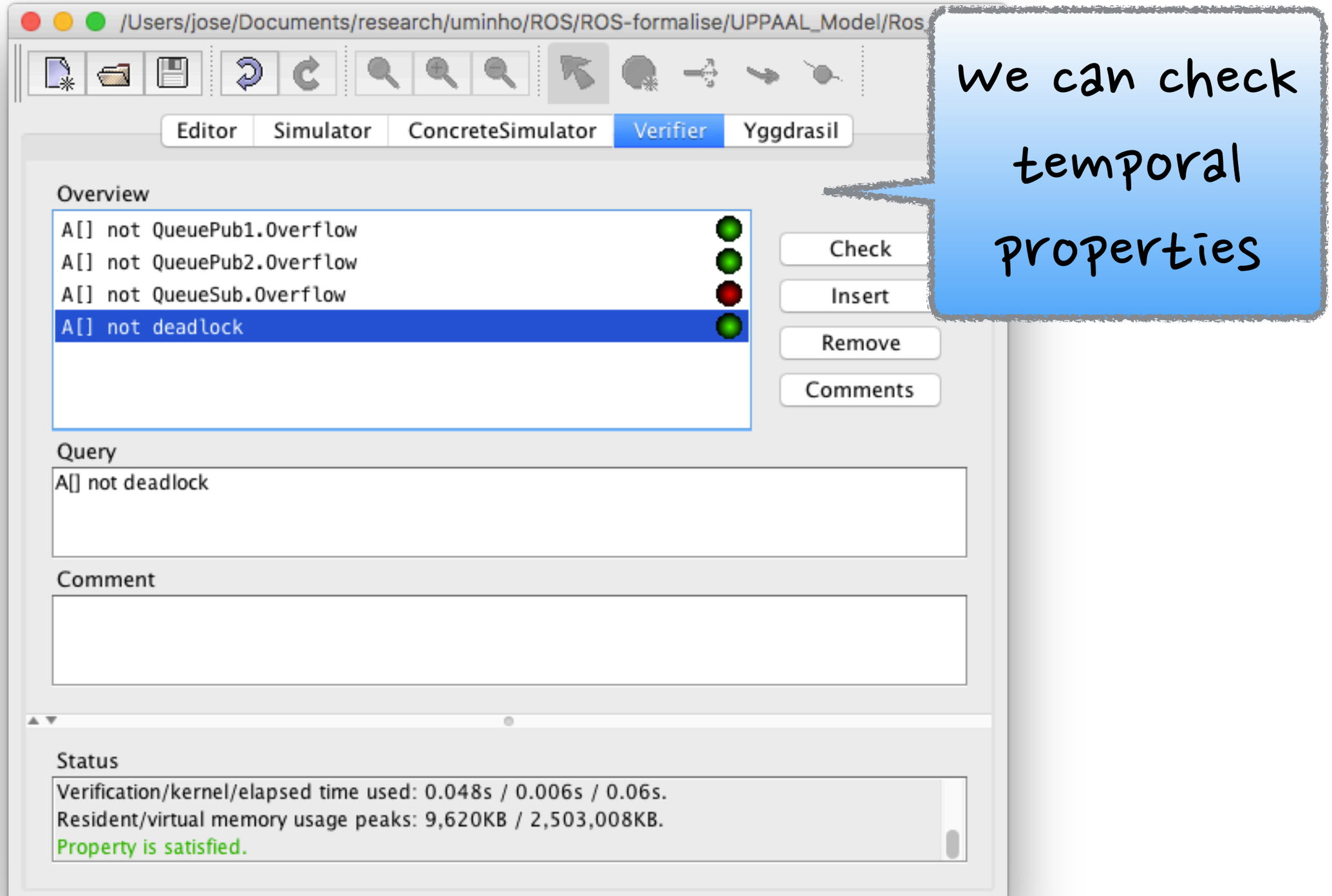
UPPAAL

Simulate
(discover traces)

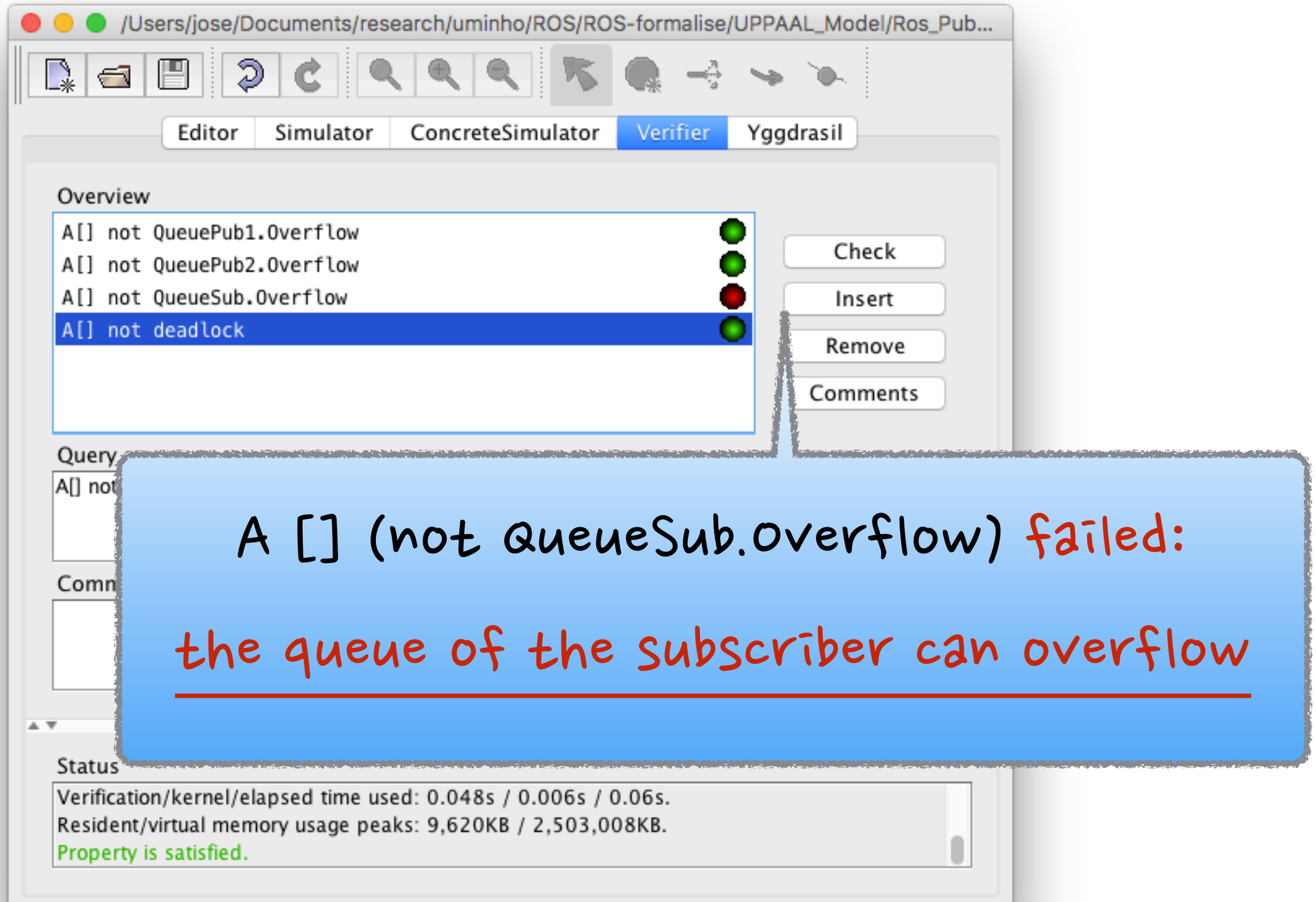


run and view
8 automata in
parallel

UPPAAL


/Users/jose/Documents/research/uminho/ROS/ROS-formalise/UPPAAL_Model/Ros
Editor
Simulator
ConcreteSimulator
Verifier
Yggdrasil
Overview
A[] not QueuePub1.Overflow
A[] not QueuePub2.Overflow
A[] not QueueSub.Overflow
A[] not deadlock
Check
Insert
Remove
Comments
Query
A[] not deadlock
Comment
Status
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


/Users/jose/Documents/research/uminho/ROS/ROS-formalise/UPPAAL_Model/Ros_Pub...
Editor
Simulator
ConcreteSimulator
Verifier
Yggdrasil
Overview
A[] not QueuePub1.Overflow
A[] not QueuePub2.Overflow
A[] not QueueSub.Overflow
A[] not deadlock
Check
Insert
Remove
Comments
Query
A[] not
Comm
Status
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

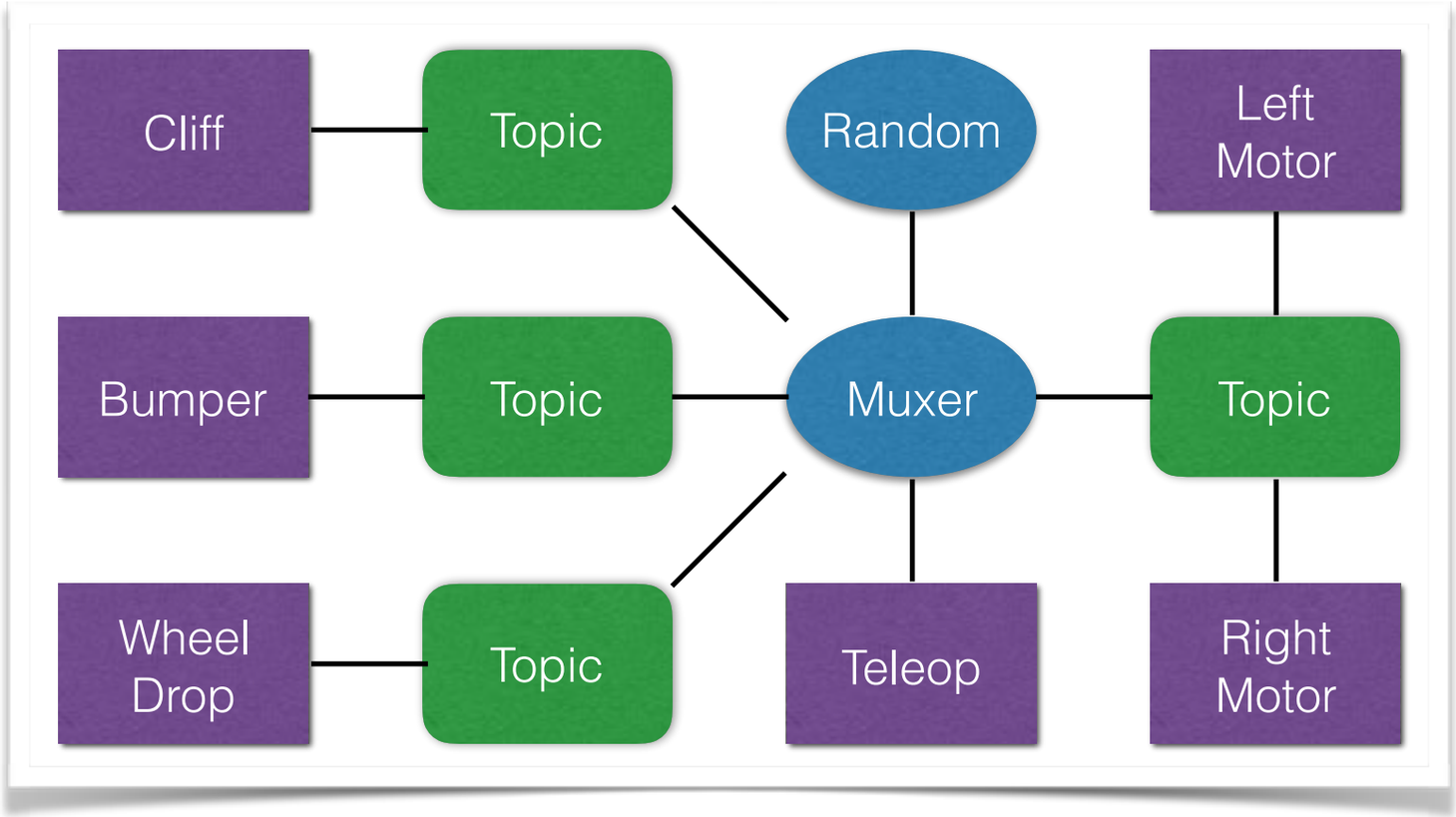
Queue Sizes			Transmission Time		Callback Time		Publishing Time-gap		Spin Time-gap	Properties		
$Q_{1\to1}$	$Q_{2\to1}$	$Q_{3\leftarrow1}$	Tmin	Tmax	CBmin	CBmax	PubTime (P_1)	PubTime (P_2)	SubTime (P_3)	Pr_1	Pr_2	Pr_3
5	5	5	1	2	1	2	4	3	4	✗	✗	✓
							4	3	5	✗	✗	✗
							4	4	9	✓	✓	✗
					4	5	4	3	1	✗	✗	✗
							4	4	8	✓	✓	✓
							4	4	9	✓	✓	✗
			3	4	1	2	8	7	15	✗	✗	✗
							8	8	17	✓	✓	✓
							8	8	18	✓	✓	✗
10	10	10	1	2	1	2	3	4	9	✗	✗	✓
							3	4	10	✗	✗	✗
							4	4	18	✓	✓	✗
					4	5	3	4	9	✗	✗	✓
							3	4	10	✗	✗	✗
							4	4	18	✓	✓	✗

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:

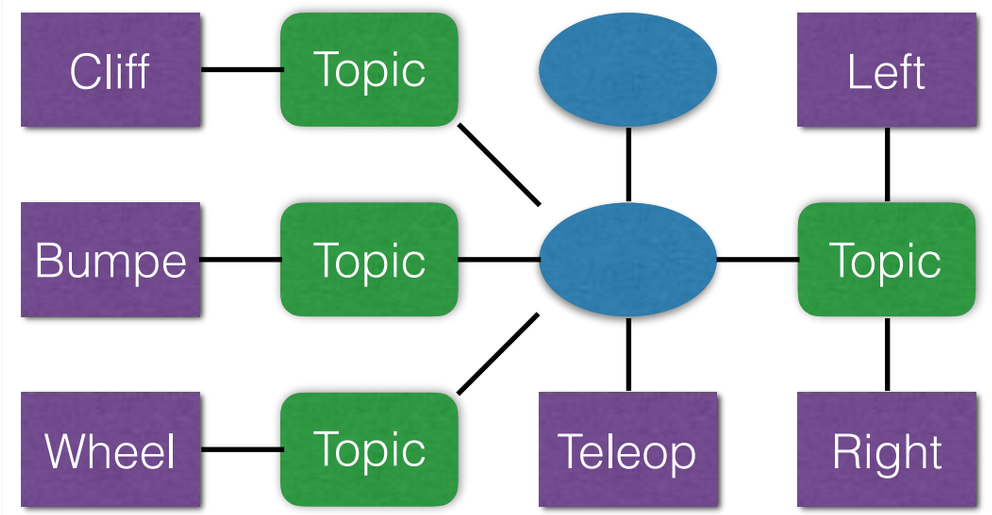
- Sensor messages get lost (too many)
- Remote never manages to go through (sensors have priority)

Wrapping up

ROS program

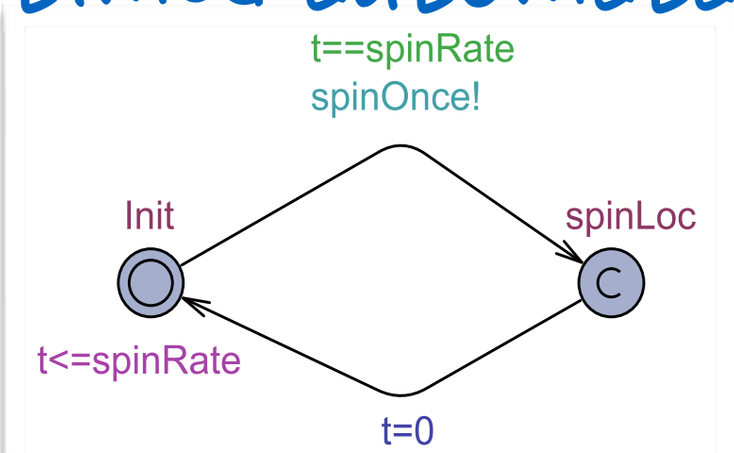
Extract

communicating components



model as

timed automata

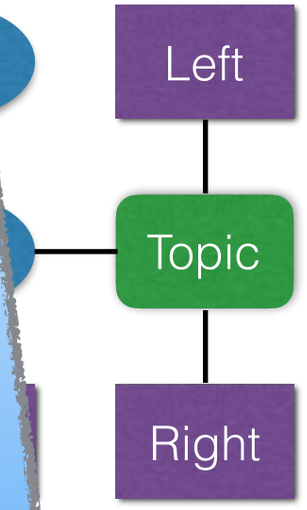


Find parameters that obey timed properties

Wrapping up

ROS program

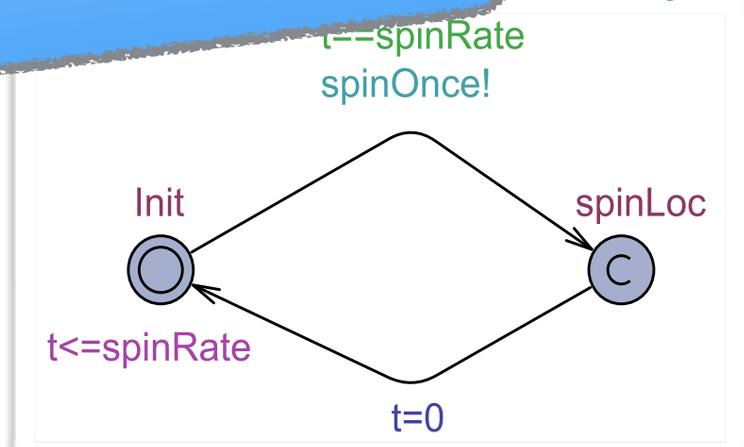
omponents



Future work:

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

mata



Find parameters that obey timed properties