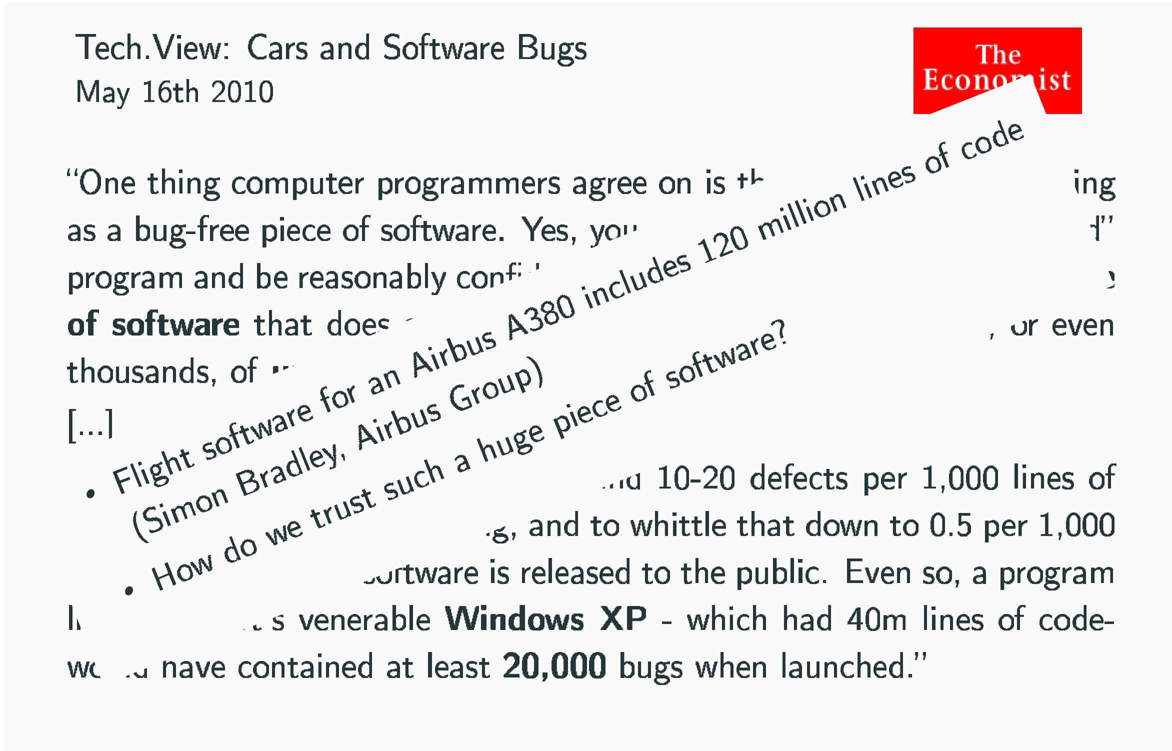
From Dependable Timed Actor Models to Executable Code

Marjan Sirjani
Professor in Software Engineering
MDH, IDT
Cyber-Physical Systems Analysis group

Center for Model-Based Cyber-Physical Product Development (MODPROD)
Feb. 5-6, 2019
Linköping University, Sweden

Thanks to Edward Lee, Hossein Hojjat, and Tom Henzinger, I used some of their slides.
Acknowledgement:
Ehsan Khamespanah, Giorgio Forcina, Luciana Provenzano, Bahar Salmani, Ali Jafari

Dependable Software?

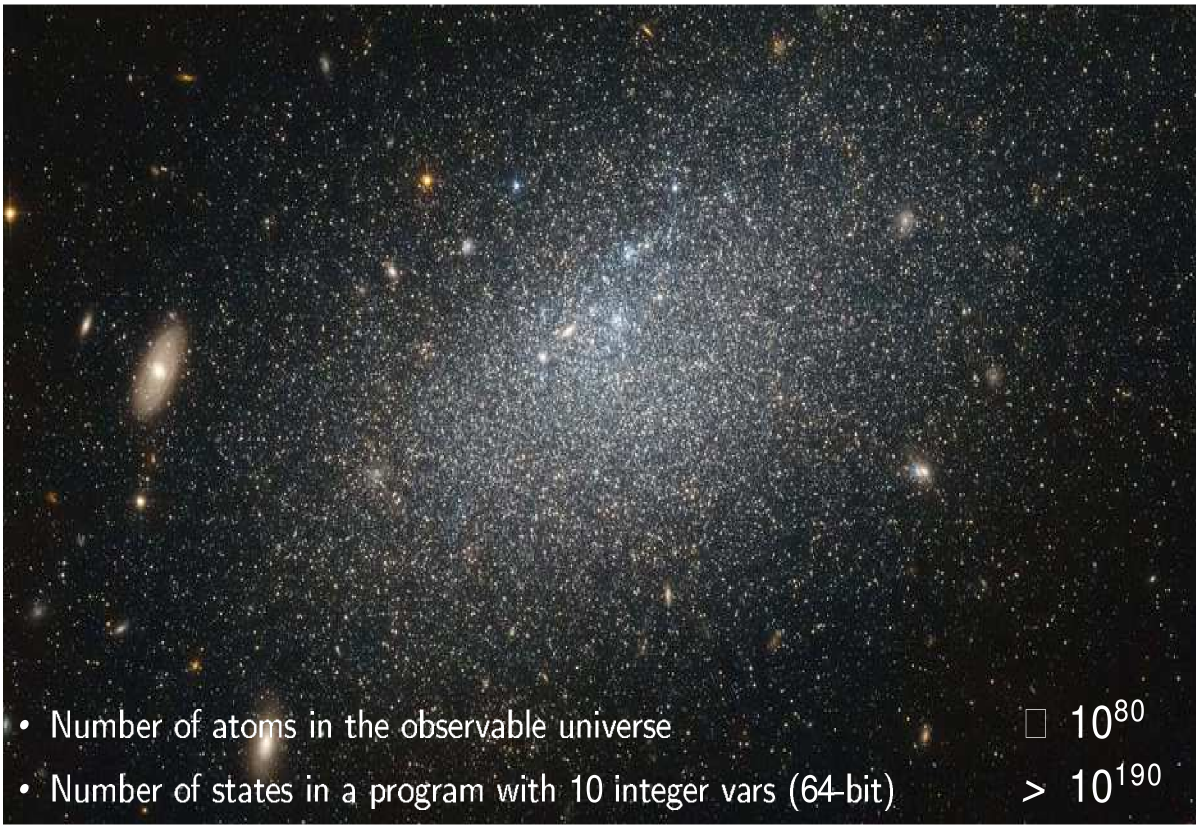


> "Since 2001, Airbus has been integrating several tool supported **formal verification** techniques into the development process of avionics software products"

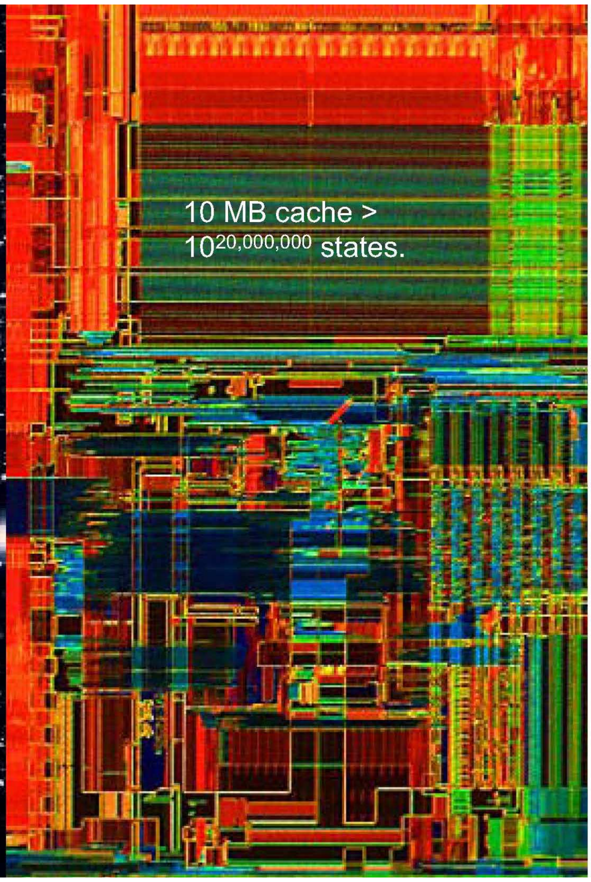
Jean Souyris et al., "Formal Verification of Avionics Software Product", FM 2009

Software truly is the most complex artifact we build routinely. It's not surprising we rarely get it right.

Tom Henzinger, 2006

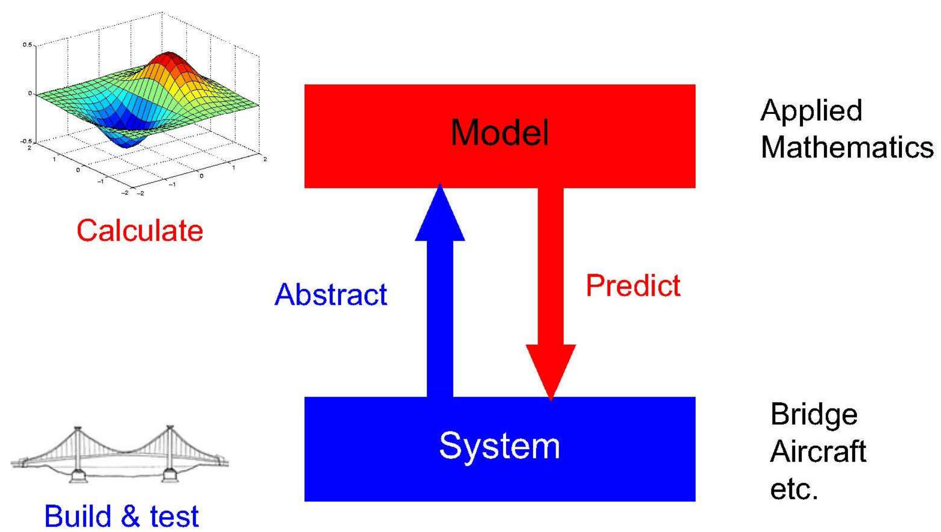

• Number of atoms in the observable universe 10^80
• Number of states in a program with 10 integer vars (64-bit) > 10^190


Between 10^69 and 10^81
atoms in the universe.


10 MB cache >
10^20,000,000 states.

	Year	Project	Lines of code
	1960s	Apollo 11 mission	145K [John D. Cressler 2016]
	1970s	Safeguard Program (US Army anti-ballistic missile system)	2M [John Lamb 1985]
	1980s	IBM air traffic control systems	2M [Computerworld 1988]
	1990s	Seawolf Submarine	3.6M [Kevin Kelly 1995]
	1990s	Boeing 777	4M [Ron J.Pehrson 1996]

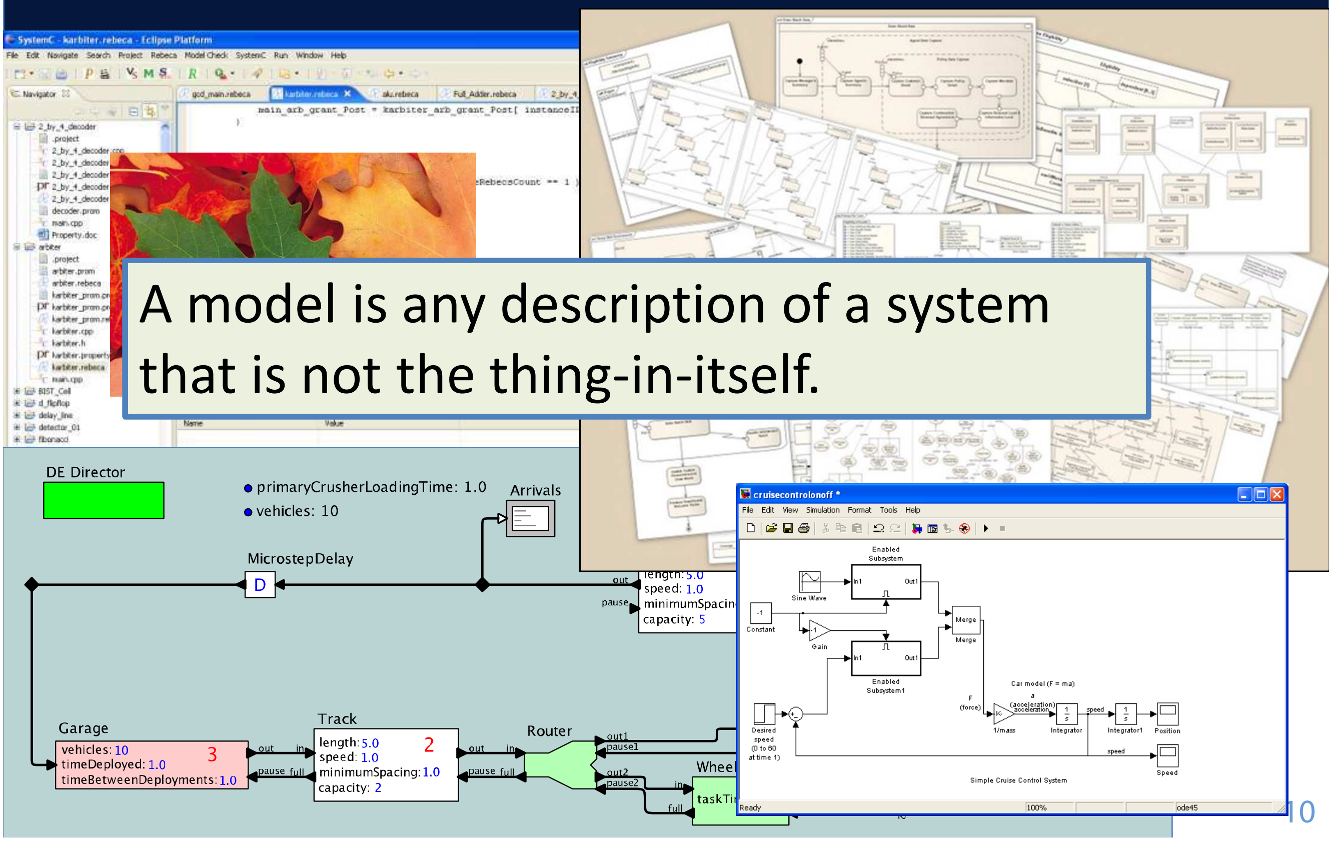
Complexity Management in Engineering



Mathematical Modeling:
A Tale of Two Cultures

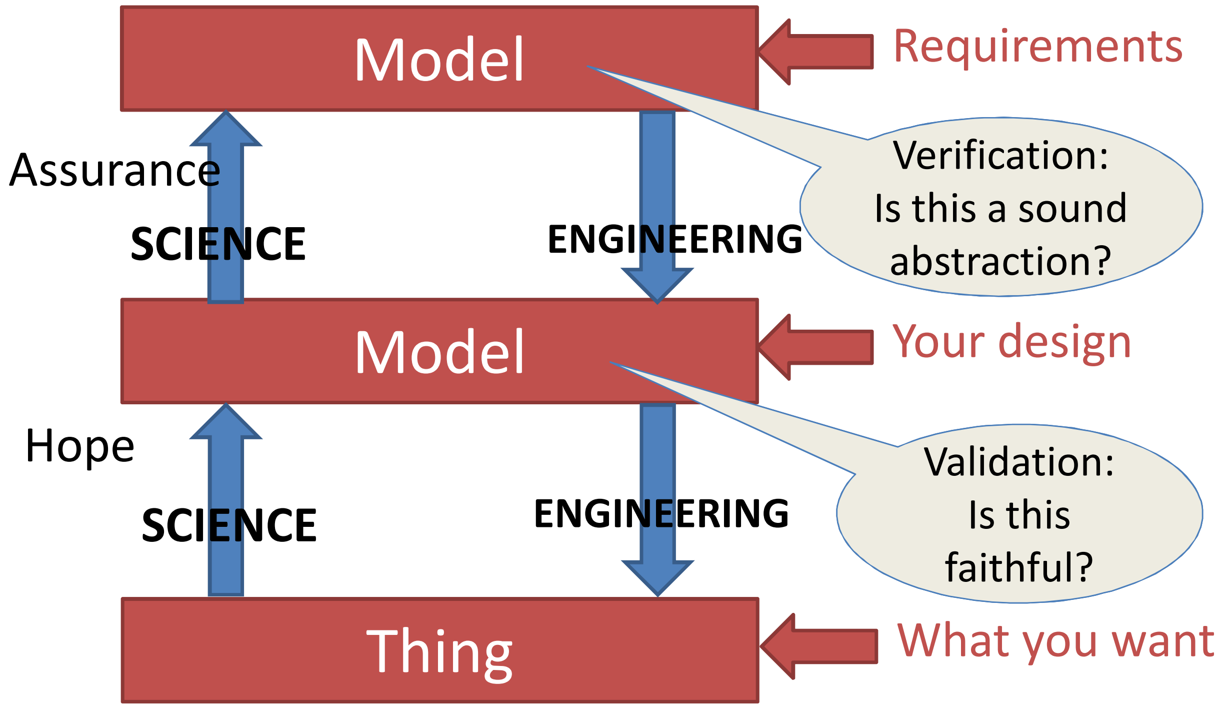
Engineering	Computer Science
Differential Equations	Mathematical Logic
Linear Algebra	Discrete Structures
Probability Theory	Automata Theory

Models



A model is any description of a system that is not the thing-in-itself.

Verification and Validation



Useful Models and Useful Things

"Essentially, all models are wrong,
but some are useful."

Box, G. E. P. and N. R. Draper, 1987: *Empirical Model-Building and Response Surfaces*. Wiley Series in Probability and Statistics, Wiley.

"Essentially, all system implementations are wrong, but some are useful."

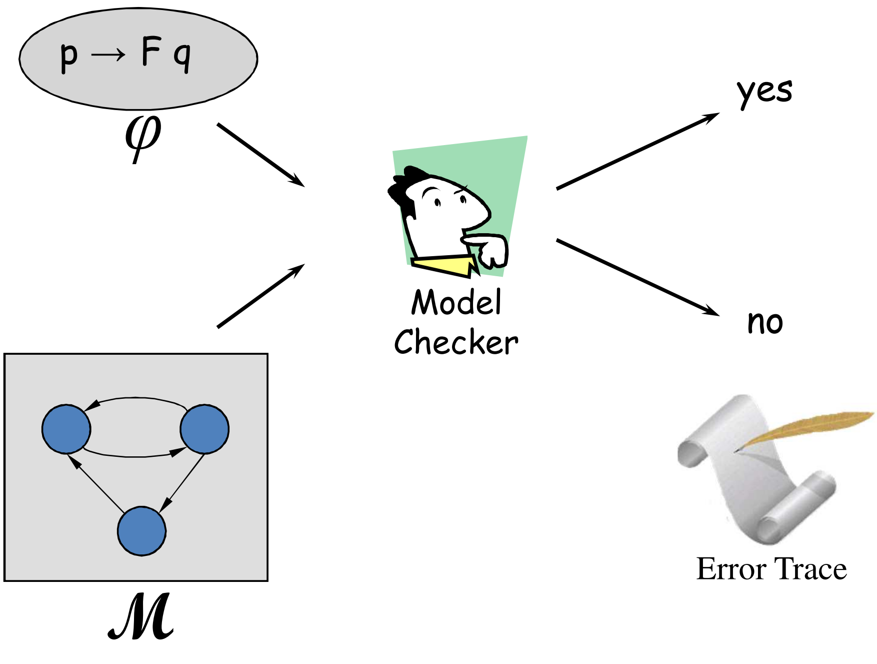
Lee and Sirjani, "What good are models," FACS 2018.

Our methodology in building software

- **Model Building: capture relevant aspects of the system formally (using logic and automata)**
- **Model Checking: implement algorithms for model analysis** [Clarke/Emerson; Queille/Sifakis1981]

Exhaustively testable pseudo-code

Model checker



Use of Formal Methods at Amazon Web Services

Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, Michael Deardeuff
Amazon.com

29th September, 2014

Exhaustively testable pseudo-code

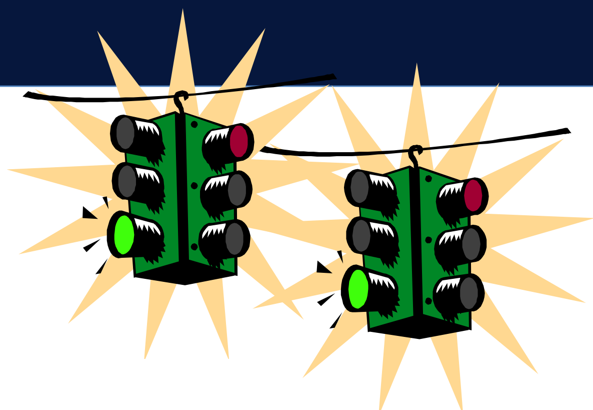
Side Benefit: A Better Way to Design Systems

- Safety properties: "what the system is *allowed* to do"
  Example: at all times, all committed data is present and correct.
  Or equivalently: at no time can the system have lost or corrupted any committed data.

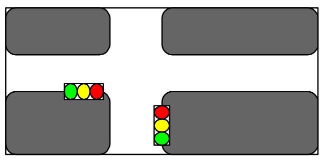
- Liveness properties: "what the system *must eventually* do"
  Example: whenever the system receives a request, it must eventually respond to that request

More Side Benefits: Improved Understanding, Productivity and Innovation

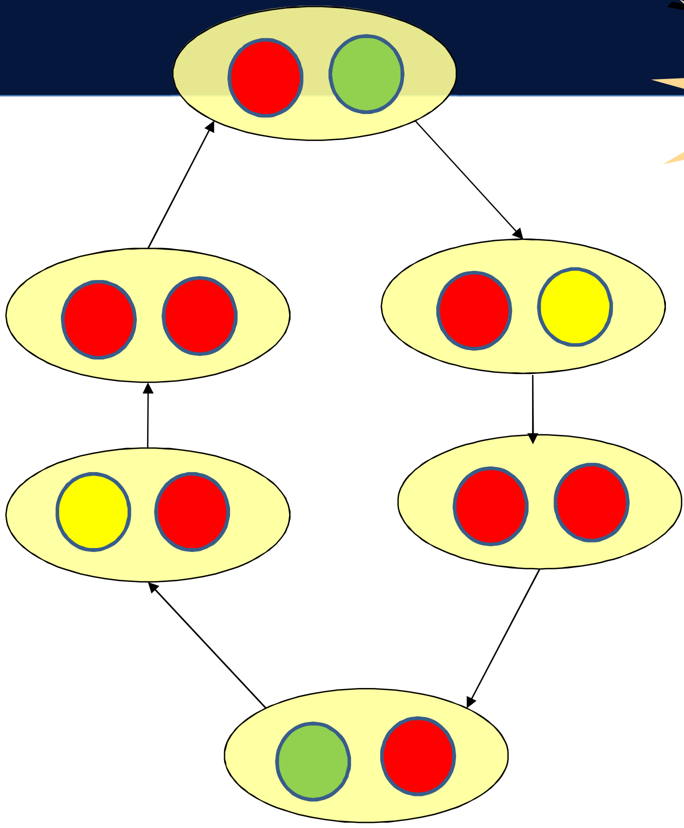
Model a traffic light

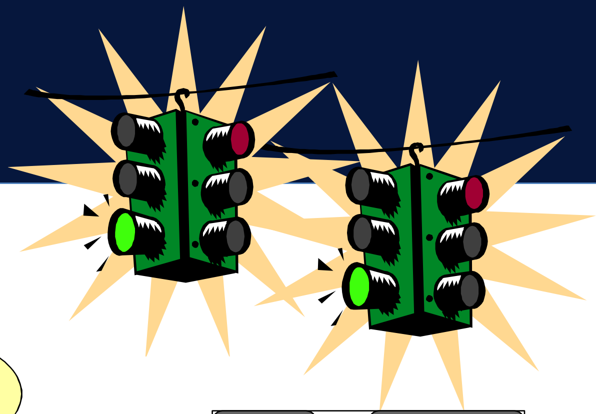


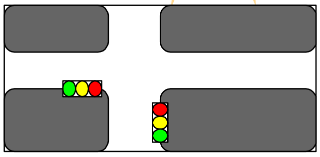
Model a crossing with two traffic lights

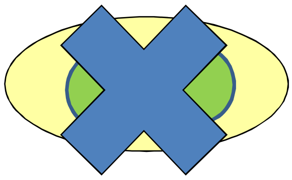


X1 = Red, X2 = Green



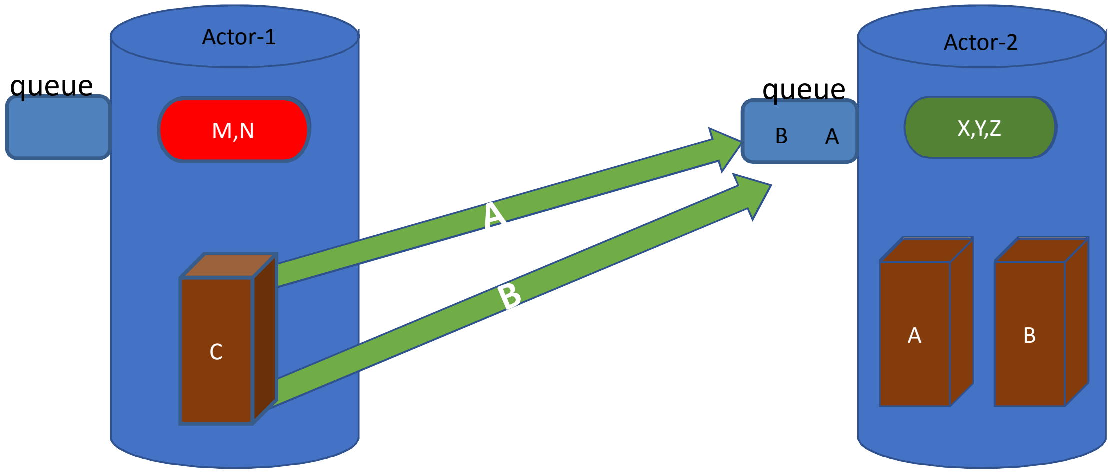






- We check the model for required properties
  - Mutual exclusion
  - Deadlock freedom
  - No starvation

ACTORS

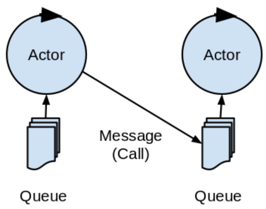


Concurrent and Distributed
Message Passing

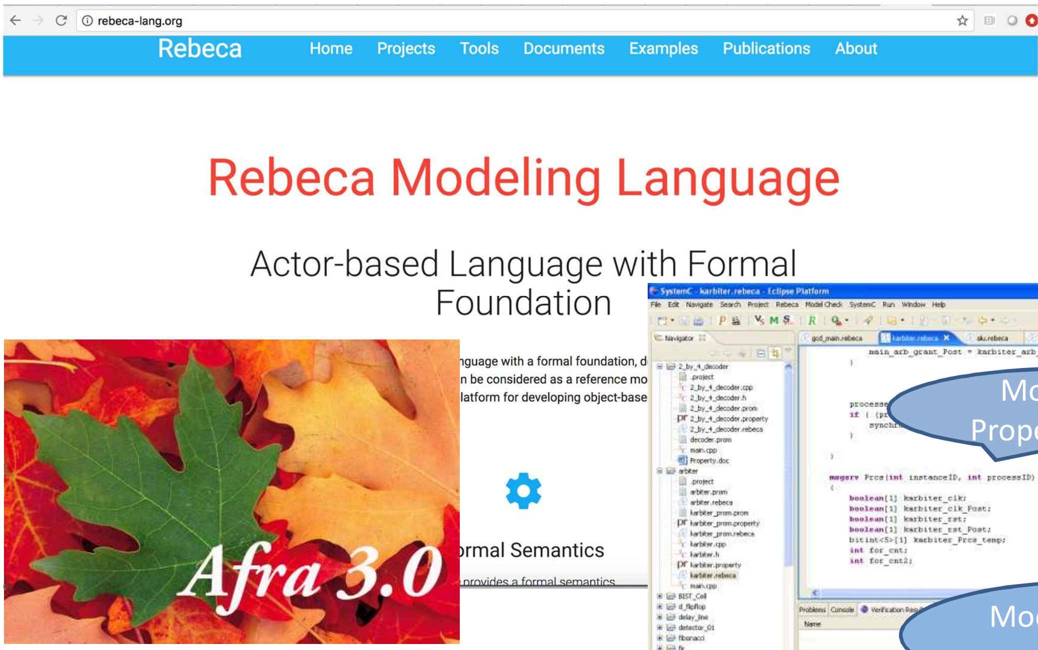
An actor: a message queue, state variables, message servers

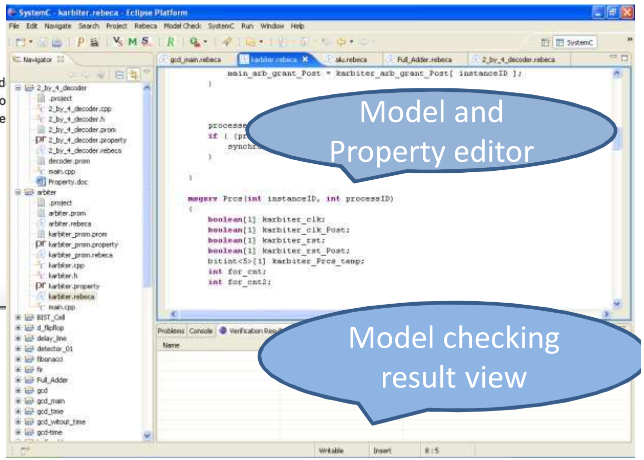
Rebeca: The Modeling Language Asynchronous and Event-driven

- **Rebeca: Re**active o**bj**e**c**t l**a**nguage (Sirjani, Movaghar. **2001**)
  - Based on Hewitt actors
  - Concurrent reactive objects (OO)
  - Java like syntax



- Communication:
  - Asynchronous message passing: non-blocking send
  - Unbounded message queue for each rebec
  - No explicit receive
- Computation:
  - Take a message from top of the queue and execute it
  - Event-driven

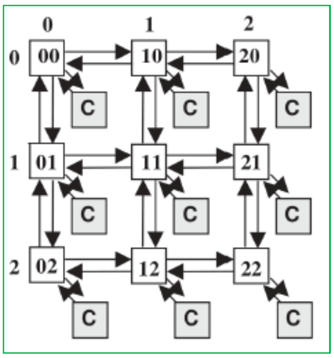


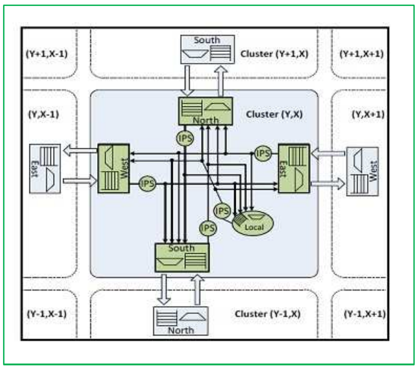


http://www.rebeca-lang.org/

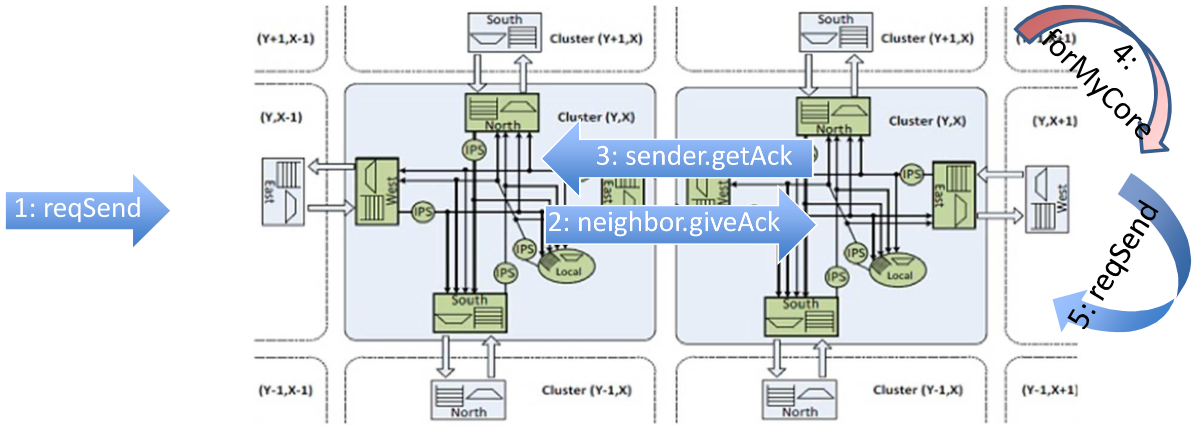
- **Ten years of Analyzing Actors: Rebeca Experience (Sirjani, Jaghouri) Invited paper at Carolyn Talcott Festschrift, 70th birthday, LNCS 7000, 2011**
- **On Time Actors (Sirjani, Khamespanah), Invited paper, Theory and Practice of Formal Methods, LNCS 9660, 2016**

Network on Chip
ASPIN: Two-dimensional mesh GALS NoC





- Explore the design space
  - Evaluate routing algorithms
  - Select best place for memory
  - Choose buffer sizes
  - …



reqSend:
//Route the Packet
neighbor.giveAck;

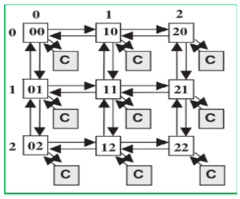
getAck:
//send the Packet
//set the flag of your port to free

giveAck:
//if I am the final Receiver
//then Consume the Packet
sender.getAck;
myCore.forMyCore;

//else if my buffer is not full
//get the Packet
sender.getAck
//and route it ahead
self.reqSend;

else (my buffer is full) wait

ASPIN: Rebeca abstract model



Actor type and its message servers

A message server

```
reactiveclass Router{
  knownrebecs
    {Router[4] neighbor, Core myCore}
  statevars{int[4] buffer;}
  Router (myId-row, myId-col) { ...
  }
  msgsrv reqSend() {
     neighbor[x]. giveAck() after(3);   . . .
  }
  msgsrv getAck() {
    // receive ack from the receiver
    // get ready for receiving the next packet
    ...
  }
  msgsrv giveAck (…) {
   //if the message is for my core use it
   myCore.forMyCore()
   //send ack to the sender
   sender.getAck() after(3);
  // if not route it to the receiver …
} }
```

```
reactiveclass Core{
   knownrebecs {Router myRouter}
   statevars{ …}
   Core ( … ) {
        ….
   }
   msgsrv forMyCore() {
     // get the Packet and use it
        . . .
   }
main(){
   Router r00(r02,r10,r01,r20)(0,0);
   Router r01(r00,r11,r02,r21)(0,1);
    …
   Core c00(r00)
   Core c01(r01)
    …
 }
```

Instances of different actors

Parameters

Known rebecs

ASPIN: Rebeca abstract model

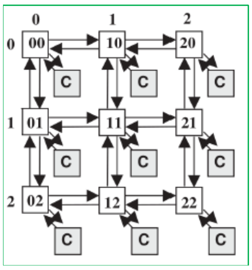
```
reactiveclass Router{
   knownrebecs {Router[4] neighbor}
   statevars{int[4] buffer;}
     Router ( … ) {
       ….
     }
     msgsrv reqSend()  {
       delay(2);
       neighbor[x]. giveAck() after(3) deadline(6);
       . . .

msgsrv giveAck (…) {
     //if the message is for my core use it
     myCore.forMyCore()
     //send ack to the sender
     sender.getAck() after(3);
    // if not and buffer not full then route it to the receiver …
    // if buffer full then busy-wait until buffer empty
    else self.giveAck() after(10),
 }
       …
}
```

Time progress because of computation delay

Deadline for the receiver

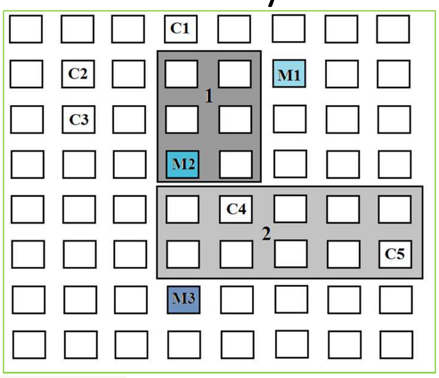
Communication delay

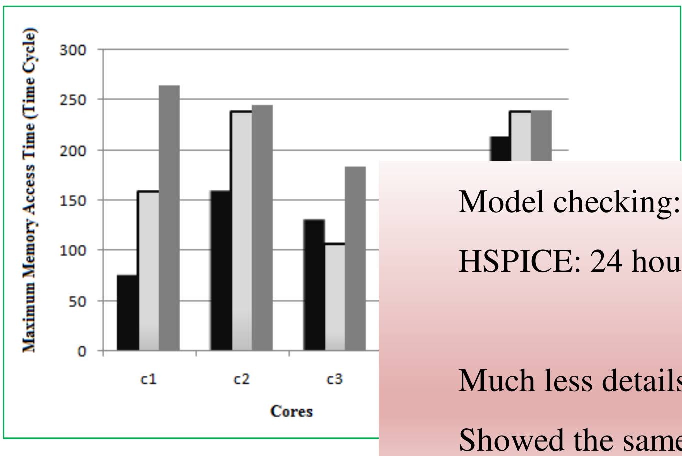
periodic tasks



Evaluation of different memory locations for ASPIN 8×8

- Consider 5 cores and their access time to the memory
- 3 choices for memory placement
- 40 packets are injected
- High congestion in area 1 and 2





Model checking: 3 seconds

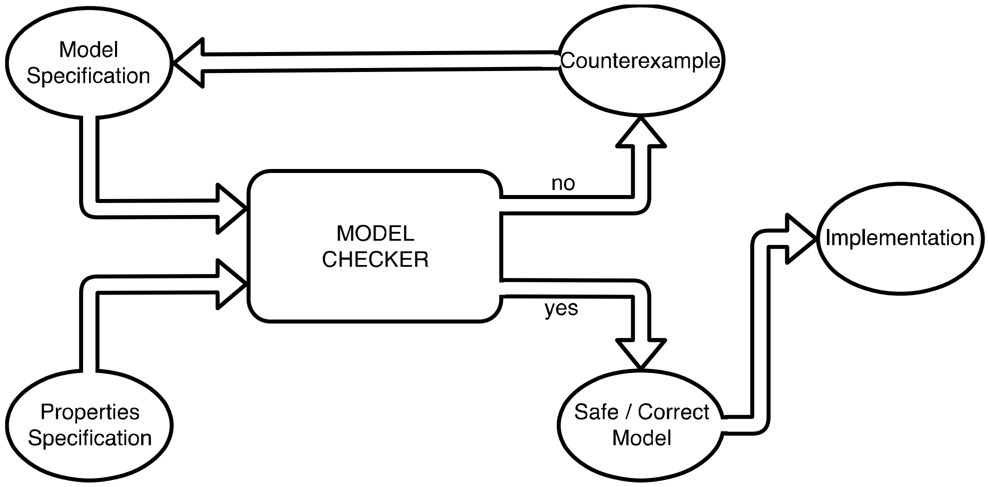
HSPICE: 24 hours

Much less details.

Showed the same trend.

…e our expectation
…a better choice
…M2
…acket injection is
… on an application
…that cores have
…ent roles)

Model Checking



From Requirements to Model

From Requirements to the Model
The Train Door System

Door Lock System

- The external doors of a train can be opened by:
  - the driver, who pushes the “external door opening button” on the driver’s desk.
  - a passenger, who pushes the “door opening button” installed on each external door.

- But, if the train is running the external doors shall be kept closed to avoid that passengers fall out of the train.

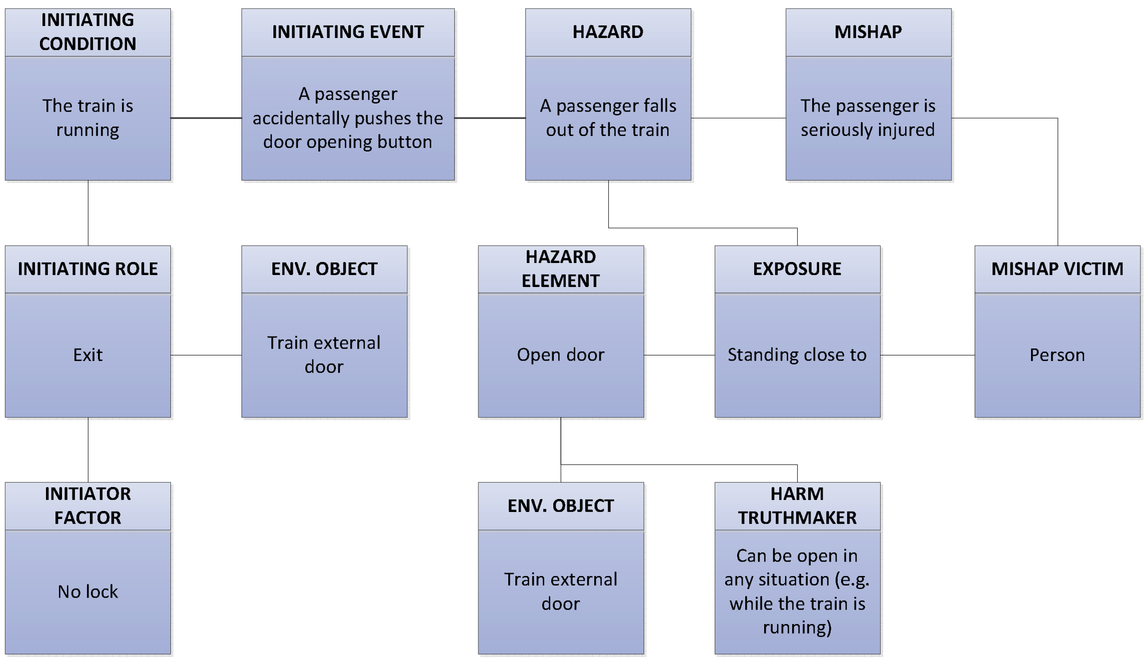
- So, the “doors lock” mechanism is put in place to keep locked all the external doors when the train is running to prevent a passenger from opening an external door out of the platform.

Properties

- **Safety**: we want to check the model if there is any possibility that a passenger can open a locked door to get off from a running train, thus causing an accident.

- **Progress**: we want to be sure that each passenger can get off the train at a platform by opening the door.

Hazard Ontology of the Door



Safety Requirements for the Train Doors Control System

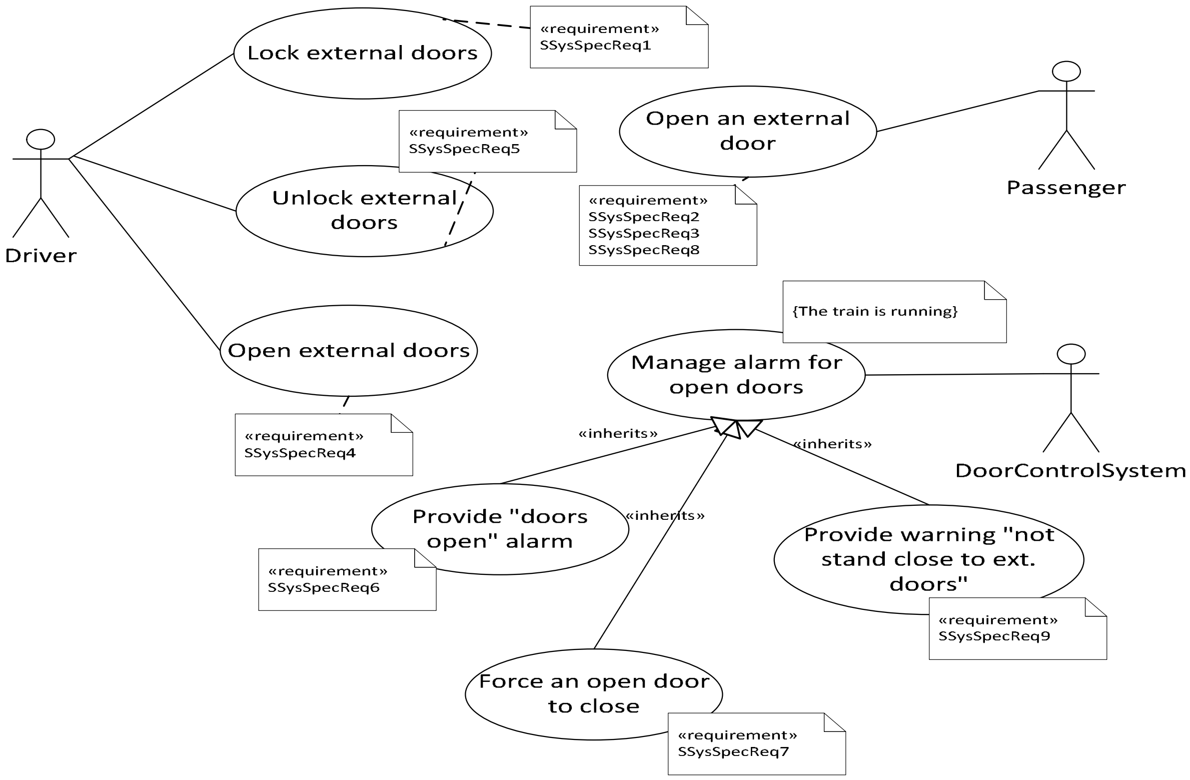
By using the Hazard Ontology, and by applying the SARE* (Safety Requirements Elicitation) approach, the analyst and safety engineer obtained a set of safety requirements (to lock the doors).

*An Ontological Approach to Elicit Safety Requirements. Luciana Provenzano, Kaj Hanninen, Jiale Zhou, and Kristina Lundqvist. Proceedings of the 24th Asia-Pacific Software Engineering Conference (APSEC'17), Nanjing, China, December 2017

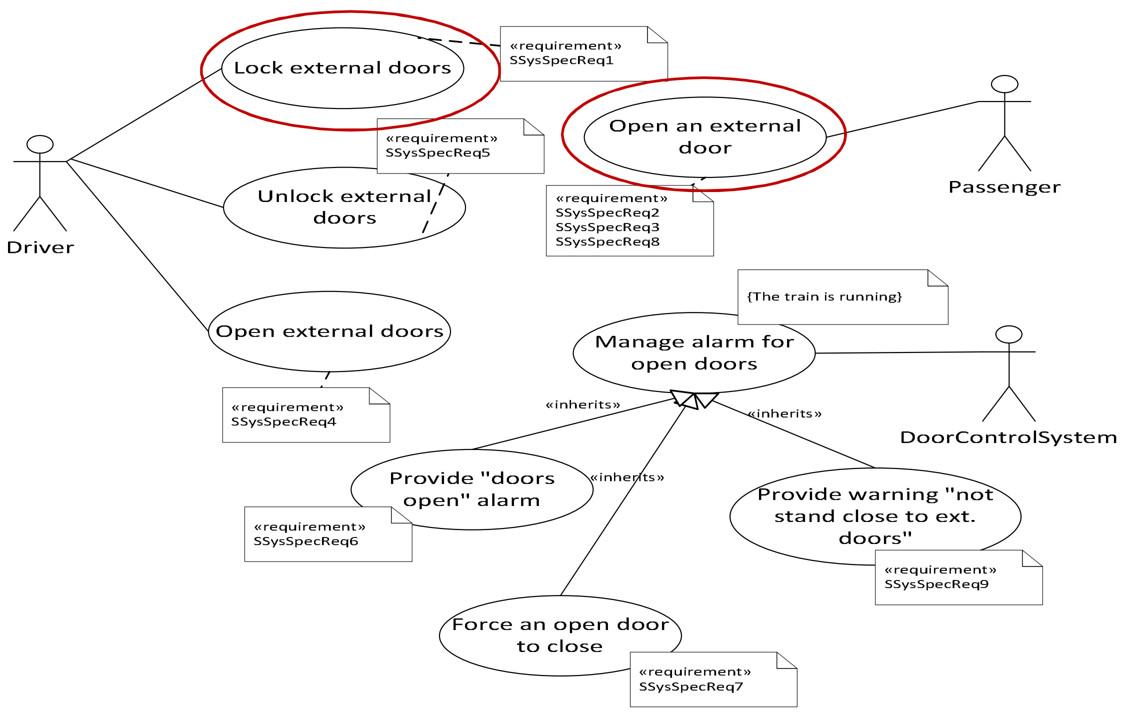
Safety Requirements Elicitation for the Train Doors Control System

REQ ID	REQ DESCRIPTION	Elicited REQ ID
SSysSpecReq1	GIVEN the train is ready to run WHEN the driver requests to lock the external doors THEN all the external doors in the train shall be closed and locked	SSysReq1
SSysSpecReq2	GIVEN an external door is locked WHEN the passenger requests to open the external door THEN the external door shall be kept closed and locked	SSysReq2
SSysSpecReq3	GIVEN an external door is unlocked WHEN the passenger requests to open the external door THEN the external door shall open	SSysReq3
SSysSpecReq4	GIVEN all the external doors on the side of the train close to the platform are unlocked WHEN the driver requests to open all the external doors THEN all the external doors on the side of the train close to the platform shall be open	SSysReq3
SSysSpecReq5	GIVEN the train arrives at a station AND the train speed is less than 0.5 km/h WHEN the driver requests to unlock all external doors that are on the train side close to the platform THEN all the external doors on that side of the train shall be unlocked	SSysReq4

From requirements to Use Case

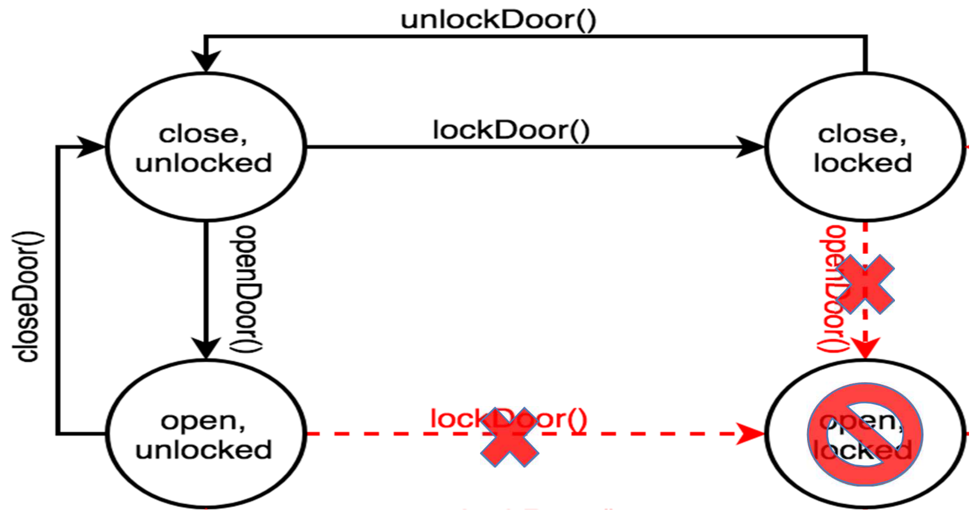


Chosen use cases

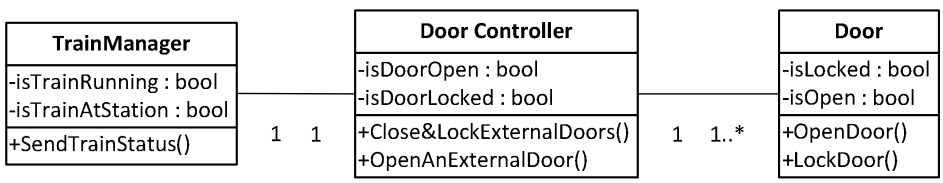


Why these two use cases?

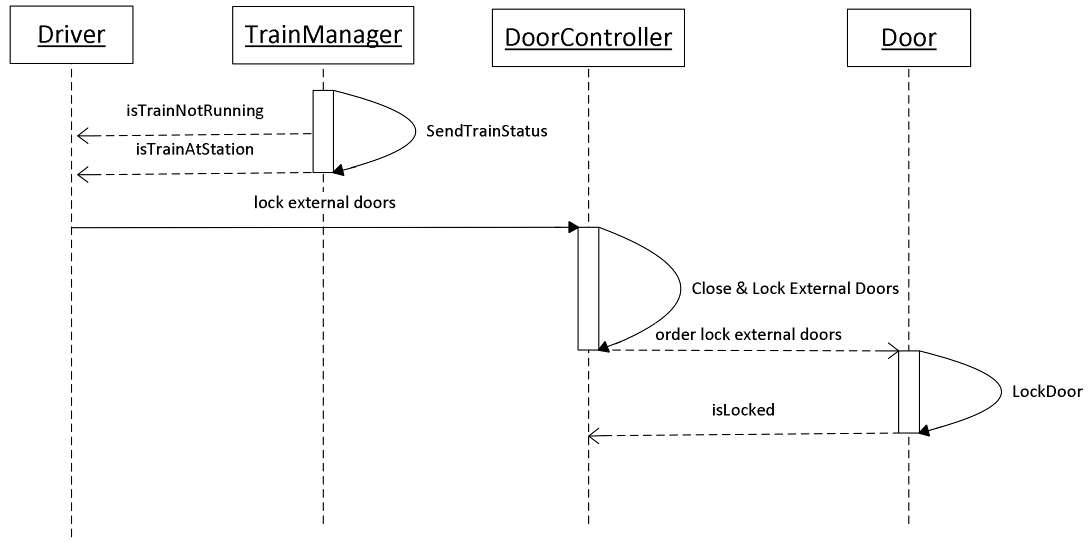
We want to verify that it is not possible to open a locked door or lock an open door.



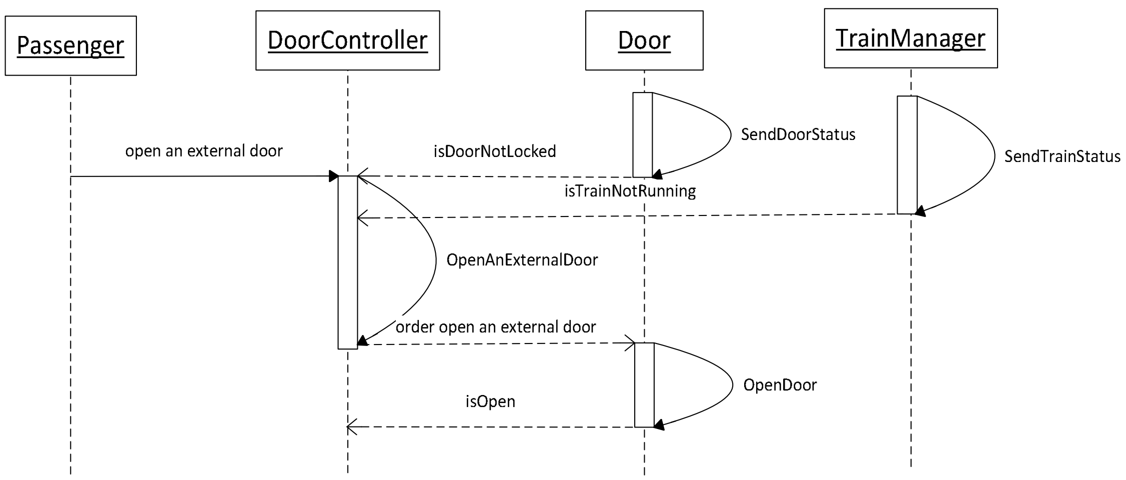
Door Control System Class diagram



Sequence Diagram: Lock External Doors



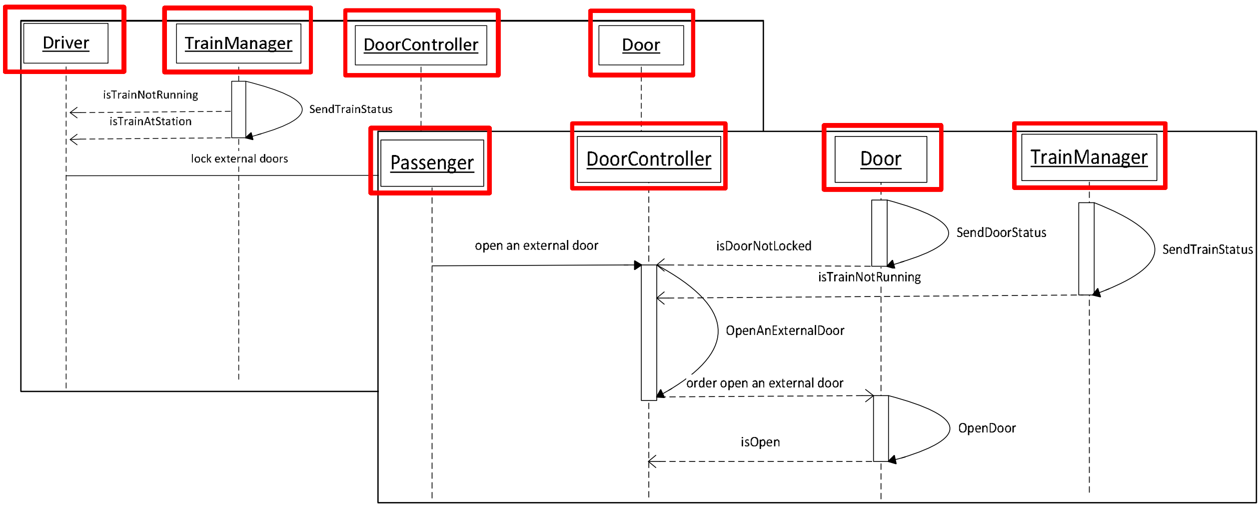
Sequence Diagram: Open an External Door



From UML to Rebeca Model

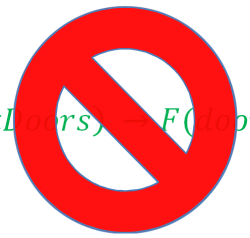
Reactive classes:

- Driver
- Passenger
- Door
- DoorController
- TrainManager



The Verification of the Rebeca Model

REQ ID	REQ DESCRIPTION	Elicited REQ ID
SSysSpecReq1	GIVEN the train is ready to run WHEN the driver requests to lock the external doors THEN all the external doors in the train shall be closed and locked	SSysReq1

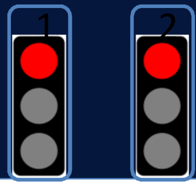


$$\varphi_1 = G((trainReady \wedge lockDoors) \rightarrow F(doorsLocked \wedge doorsClosed))$$

The Rebeca Model counter-example

Using Rebeca and Afra for Modeling and Model Checking

Traffic Lights

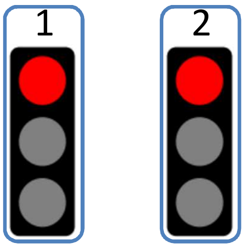



```
reactiveclass TrafficLight(5) {
knownrebecs {
TrafficLight tOther;
}
statevars {
byte Color;
}

TrafficLight(byte myI ) {
Color ! "; #$ re  $#
if (myI !!%) {
    self&'e to(reen();
    }
}

& & &
main {
TrafficLight
traffic%(traffic))*(%);
TrafficLight
traffic)(traffic%)*());
}
```

```
msgsrv 'e to(reen() {
Color ! %; #$ green $#
self&(reento+ellow();
}

msgsrv (reento+ellow() {
Color ! ); #$ yellow $#
self&+ellowto'e ();
}

msgsrv +ellowto'e () {
Color ! "; #$ re  $#
tOther&'e to(reen();
}
```


Rebeca Model: Traffic Lights

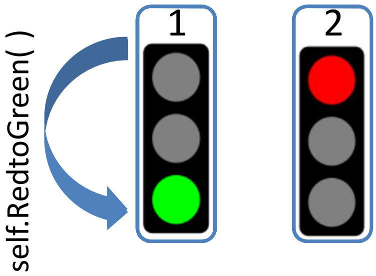
```
reactiveclass TrafficLight(5) {
knownrebecs {
TrafficLight tOther;
}
statevars {
byte Color;
}

TrafficLight(byte myId) {
Color = 0; /* red */
if (myId==1) {
    self.RedtoGreen();
    }
}
msgsrv RedtoGreen() {
Color ! %;
self&(reento+ellow();
}
msgsrv (reento+ellow() {
Color ! );
self&+ellowto'e ();
}
msgsrv +ellowto'e () {
Color ! ";
tOther&'e to(reen();
}
}

main {
TrafficLight traffic%(traffic))*(%);
TrafficLight traffic)(traffic%)*());
}
```




```
reactiveclass TrafficLight(5) {
    knownrebecs {
        TrafficLight tOther;
    }
    statevars {
        byte Color;
    }

    TrafficLight(byte myId) {
        Color = 0;  /* red */
        if (myId==1) {
            self.RedtoGreen();
            }
    }
```

Rebeca Model: Traffic Lights

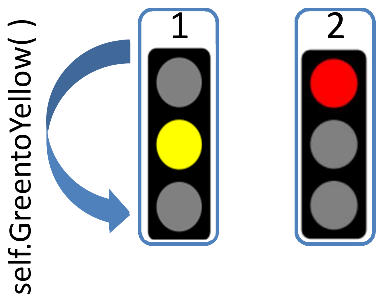
```
reactiveclass TrafficLight(5) {
knownrebecs {
TrafficLight tOther;
}
statevars {
byte Color;
}

TrafficLight(byte myI ) {
Color ! "; #$ re  $#
if (myI !!%) {
    self&'e to(reen();
    }
}
msgsrv RedtoGreen() {
Color = 1;
self.GreentoYellow();
}
msgsrv (reento+ellow() {
Color ! );
self&+ellowto'e ();
}
msgsrv +ellowto'e () {
Color ! ";
tOther&'e to(reen();
}
}

main {
TrafficLight traffic%(traffic))*(%);
TrafficLight traffic)(traffic%)*());
}
```




```
msgsrv RedtoGreen() {
    Color = 1;  #$ green $#
    self.GreentoYellow();
}
```

Rebeca Model: Traffic Lights

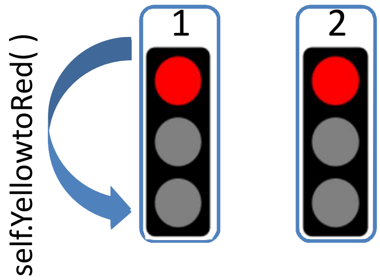
```
reactiveclass TrafficLight(5) {
knownrebecs {
TrafficLight tOther;
}
statevars {
byte Color;
}

TrafficLight(byte myI ) {
Color ! "; #$ re  $#
if (myI !!%) {
    self&'e to(reen();
    }
}
msgsrv 'e to(reen() {
Color ! %;
self&(reento+ellow();
}
msgsrv GreentoYellow() {
Color = 2;
self.YellowtoRed();
}
msgsrv YellowtoRed() {
Color ! ";
tOther&'e to(reen();
}
}

main {
TrafficLight traffic%(traffic))*(%);
TrafficLight traffic)(traffic%)*());
}
```




```
msgsrv GreentoYellow() {
    Color = 2; #$ yellow $#
    self.YellowtoRed();
}
```

Rebeca Model: Traffic Lights

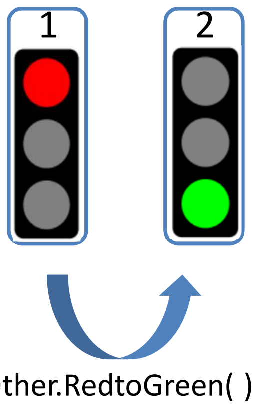
```
reactiveclass TrafficLight(5) {
knownrebecs {
TrafficLight tOther;
}
statevars {
byte Color;
}

TrafficLight(byte myI ) {
Color ! "; #$ re  $#
if (myI !!%) {
    self&’e to(reen();
    }
}
msgsrv ’e to(reen() {
Color ! %;
self&(reento+ellow();
}
msgsrv (reento+ellow() {
Color ! );
self&+ellowto’e ();
}
msgsrv YellowtoRed() {
Color = 0;
tOther.RedtoGreen();
}
}

main {
TrafficLight traffic%(traffic))*(%);
TrafficLight traffic)(traffic%)*());
}
```




```
msgsrv YellowtoRed() {
    Color = 0;  #$ re  $#
    tOther.RedtoGreen();
}
```

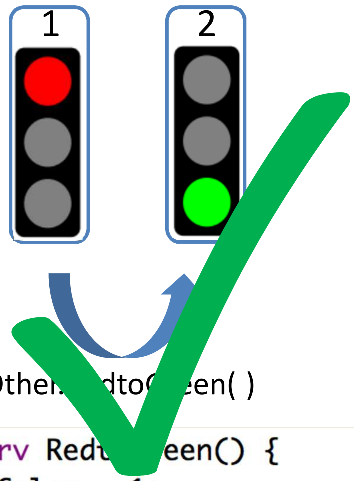
Rebeca Model: Traffic Lights

```
reactiveclass TrafficLight(5) {
knownrebecs {
TrafficLight tOther;
}
statevars {
byte Color;
}

TrafficLight(byte myI ) {
Color ! "; #$ re  $#
if (myI !!%) {
    self&’e to(reen();
    }
}
msgsrv RedtoGreen() {
Color = 1;
self.GreentoYellow();
}
msgsrv GreentoYellow() {
Color ! );
self&+ellowto’e ();
}
msgsrv +ellowto’e () {
Color ! ";
tOther&’e to(reen();
}
}

main {
TrafficLight traffic%(traffic))*(%);
TrafficLight traffic)(traffic%)*());
}
```




```
msgsrv RedtoGreen() {
    Color = 1;  #$ green $#
    self.GreentoYellow();
}
```

Rebeca Model: Traffic Lights

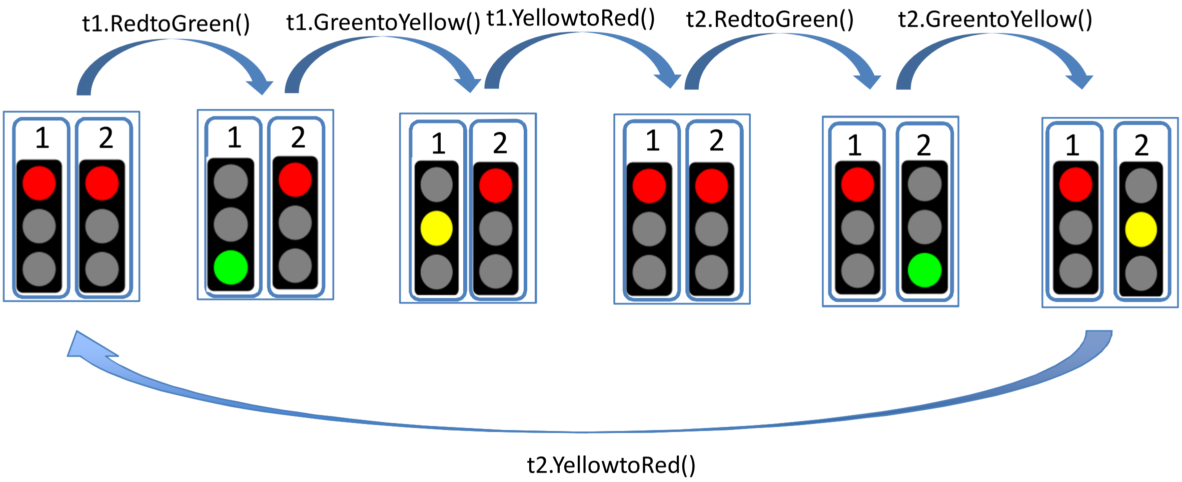


tOther.RedtoGreen( )

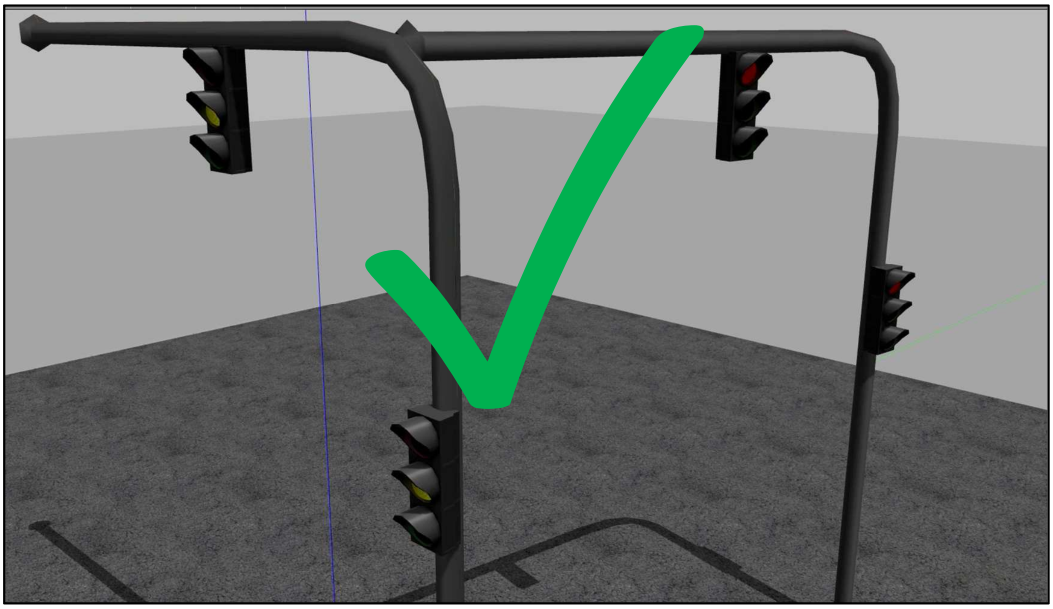
```
msgsrv RedtoGreen() {
    Color = 1; #$ green $#
    self.GreentoYellow();
}
```

$\varphi_1 = G(\neg(green_1 \wedge green_2))$ → NO CONCURRENT GREEN

Safe Rebeca Model: State-space

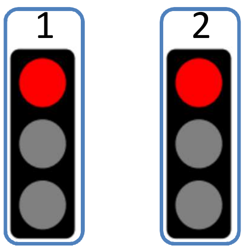


Rebeca Model: Traffic Lights



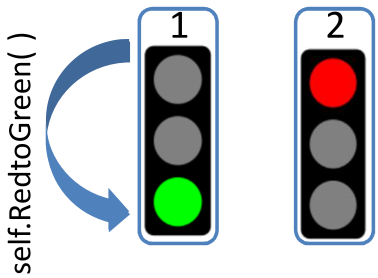
Impatient Rebeca Model: Traffic Lights

```
reactiveclass TrafficLight(5) {
knownrebecs {
TrafficLight tOther;
}
statevars {
byte Color;
}

TrafficLight(byte myI ) {
Color ! "; #$ re  $#
if (myI !!%) {
    self&'e to(reen();
    }
}
msgsrv 'e to(reen() {
Color ! %;
self&(reento+ellow();
}
msgsrv (reento+ellow() {
Color = 2;
self.YellowtoRed();
tOther.RedtoGreen();
}
msgsrv +ellowto'e () {
Color ! ";
}
}

main {
TrafficLight traffic%(traffic))*(%);
TrafficLight traffic)(traffic%)*());
}
```



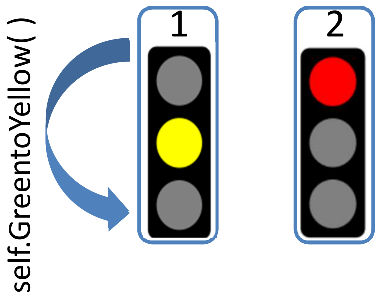

```
reactiveclass TrafficLight(5) {
    knownrebecs {
        TrafficLight tOther;
    }
    statevars {
        byte Color;
    }

    TrafficLight(byte myId) {
        Color = 0;  /* red */
        if (myId==1) {
            self.RedtoGreen();
            }
    }
```

Impatient Rebeca Model: Traffic Lights




```
msgsrv RedtoGreen() {
    Color = 1;
    self.GreentoYellow();
}
```

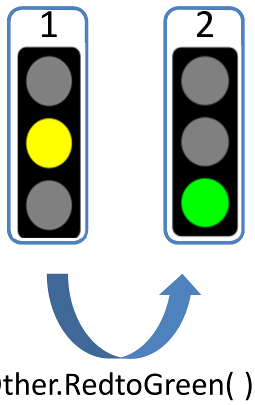
Impatient Rebeca Model: Traffic Lights




```
msgsrv GreentoYellow() {
    Color = 2;
    self.YellowtoRed();
    tOther.RedtoGreen();
}
```

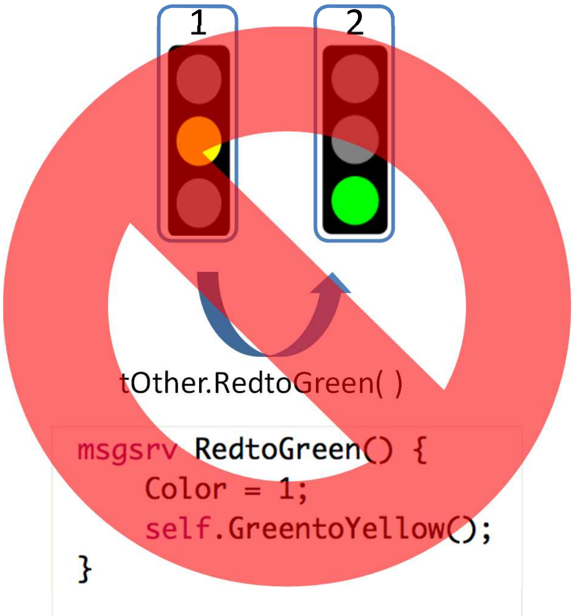
- **What will happen here?**

Impatient Rebeca Model: Traffic Lights

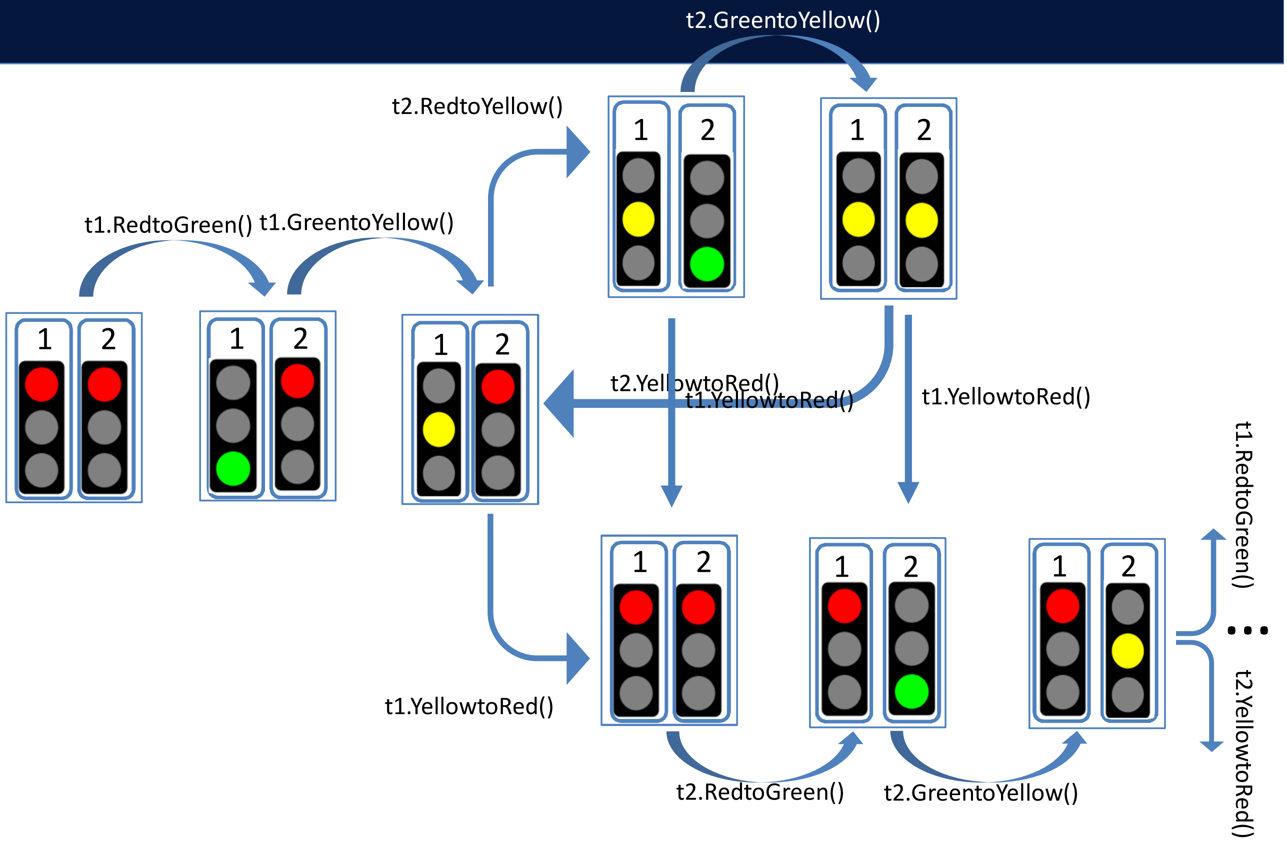



```
msgsrv RedtoGreen() {
    Color = 1;
    self.GreentoYellow();
}
```

Impatient Rebeca Model: Traffic Lights



Impatient Rebeca Model: State-space

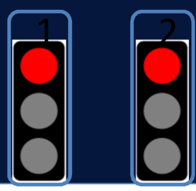


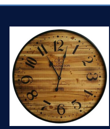
Impatient Rebeca Model: Traffic Lights

$\varphi_1 = G(\neg(green_1 \wedge green_2))$ → NO CONCURRENT GREEN ✓

$\varphi_n = G(\neg(yellow_1 \wedge green_2))$ → NO YELLOW AND GREEN ✗

Timed Traffic Lights



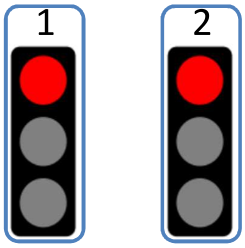


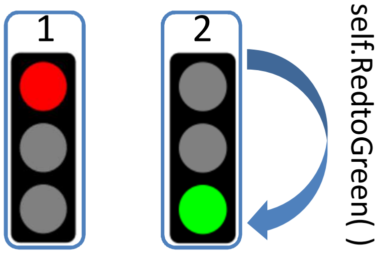
```
reactiveclass TrafficLight(5) {
knownrebecs {
TrafficLight tOther;
}
statevars { byte Color;
}
TrafficLight(byte myI ) {
  Color ! "; #$ re  $#
  if (myI !!%) {
    self&'e to(reen() after(2);
    }
  else self&'e to(reen()

}
& & &
main {
TrafficLight
traffic%(traffic))*(%);
TrafficLight
traffic)(traffic%)*());
}
```

```
msgsrv 'e to(reen() {
Color ≐ ϒ; #$ green $#
delay(2);
self&(reento+ellow();
}

msgsrv (reento+ellow() {
Color ! ); #$ yellow $#
delay(2);
self.Yellowto'e ();
}

msgsrv +ellowto'e () {
Color ≐ ϋ; #$ re  $#
delay(2);
self&'e to(reen();
}
```

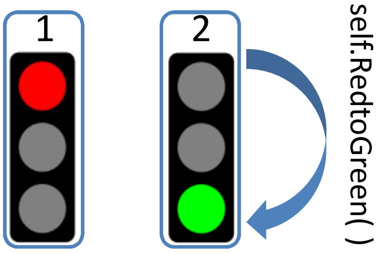
Timed Rebeca Model: Traffic Lights

```
reactiveclass TrafficLight(5) {
knownrebecs {
TrafficLight tOther;
}
statevars {
byte Color;
}

TrafficLight(byte myI ) {
  Color ! "; #$ re  $#

 if (myI !!%) {
    self&’e to(reen() after());
    }
 else self&’e to(reen()
 }
```





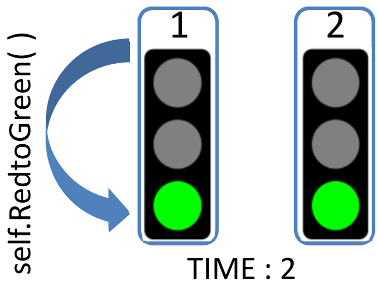
Timed Rebeca Model: Traffic Lights




```
msgsrv ’e to(reen() {
Color ! %; #$ green $#
 elay());
self&(reento+ellow();
}
```

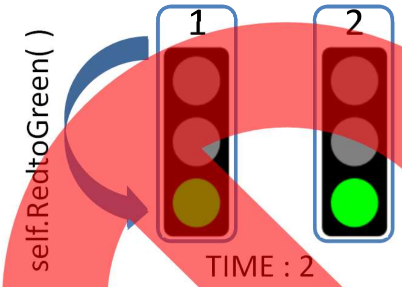
- **What will happen here?**

Timed Rebeca Model: Traffic Lights




```
msgsrv ’e to(reen() {
Color ! %; #$ green $#
 elay());
self&(reento+ellow();
}
```

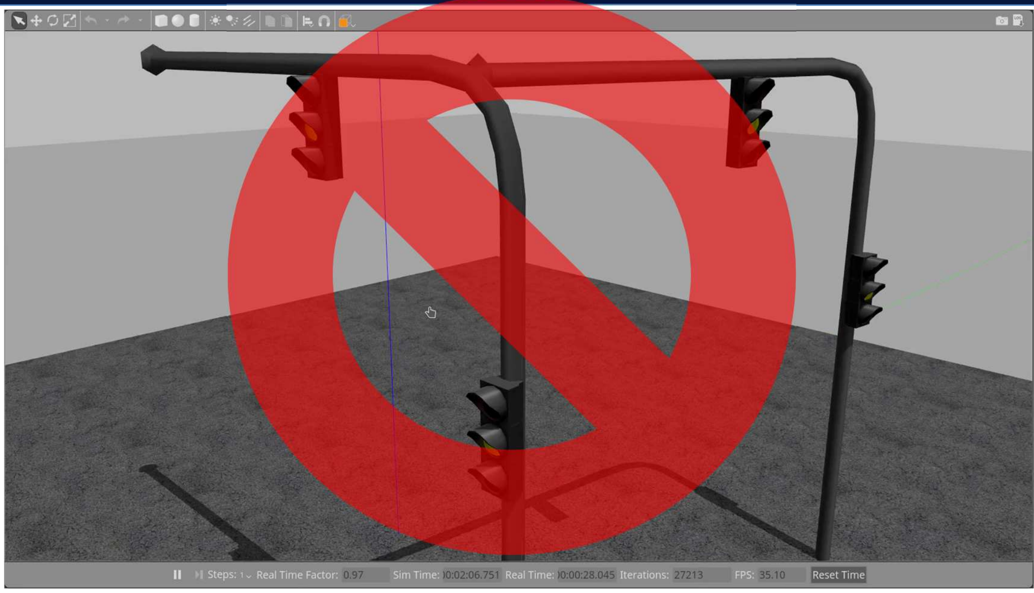
Timed Rebeca Model: Traffic Lights




```
msgsrv RedtoGreen() {
    Color = 1; //Green
    //temp = now;
    //temp = 0;
    driftedDelay = 2;
    delay(driftedDelay);
    self.GreentoYellow();
}
```

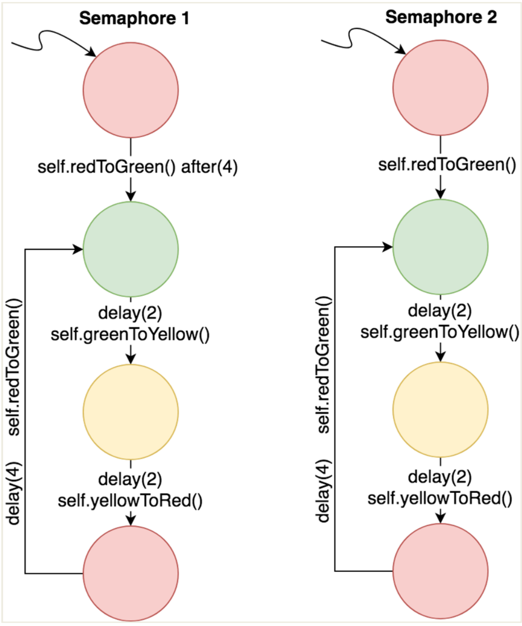
$\varphi_1 = G(\neg(green_1 \wedge green_2))$ → NO CONCURRENT GREEN ✗

Timed Rebeca Model: Traffic Lights

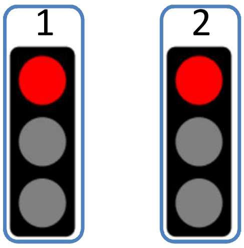


$\varphi_1 = G(\neg(green_1 \wedge green_2))$ → NO GREEN AND GREEN ✗

Timed Rebeca Model: Traffic Lights
Different shift in time



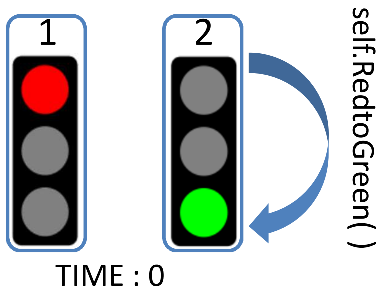
Timed Rebeca Model: Traffic Lights
Different shift in time



TIME : 0

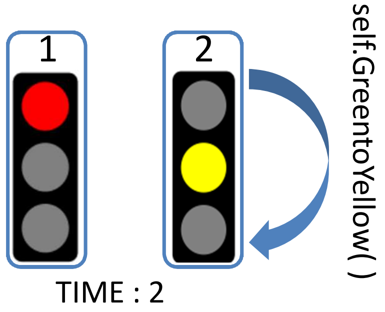
```
reactiveclass TrafficLight(5) {
    knownrebecs {
        TrafficLight tOther;
    }

    statevars {
        byte Color;
        int driftedDelay;
        int temp;
    }

    TrafficLight(byte myId) {
        Color = 0;  //Red
        if (myId==1) {
            self.RedtoGreen() after(4);
        }
        else self.RedtoGreen();
    }
```

Timed Rebeca Model: Traffic Lights
Different shift in time



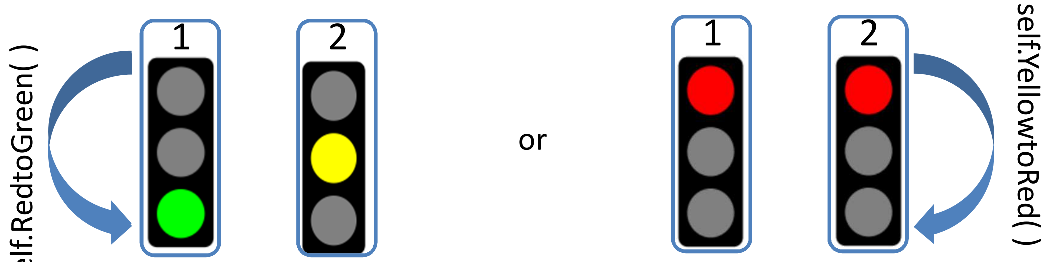

```
msgsrv ’e to(reen() {
Color ! %; #$ green $#
 elay());
self&(reento+ellow();
}
```

Timed Rebeca Model: Traffic Lights
Different shift in time




```
msgsrv (reento+ellow() {
Color ! ); #$ yellow $#
 elay());
self&+ellowto’e ();
}
```

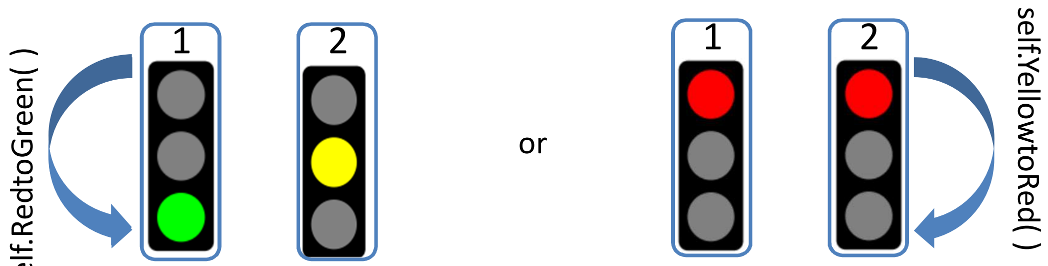
Timed Rebeca Model: Traffic Lights
Different shift in time




```
msgsrv ’e to(reen() {
Color ! %; #$ green $#
 elay());
self&(reento+ellow();
}
```

```
msgsrv +ellowto’e () {
Color ! "; #$ re  $#
 elay());
self&’e to(reen();
}
```

Timed Rebeca Model: Traffic Lights
Different shift in time

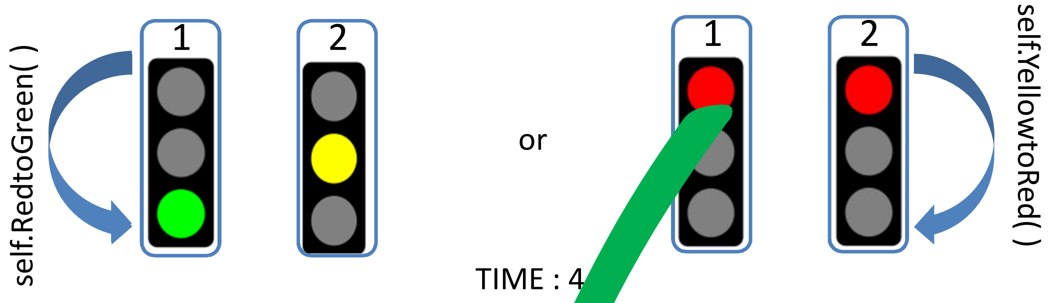



```
msgsrv RedtoGreen() {
    Color = 1; //Green
    driftedDelay = 2;
    delay(driftedDelay);
    self.GreentoYellow();
}
```

```
msgsrv YellowtoRed() {
    Color = 0; //Red
    driftedDelay = 4;
    delay(driftedDelay);
    self.RedtoGreen();
}
```

$\varphi_1 = G(\neg(green_1 \wedge green_2)) \rightarrow$ NO CONCURRENT GREEN ✓

Timed Rebeca Model: Traffic Lights
Different shift in time

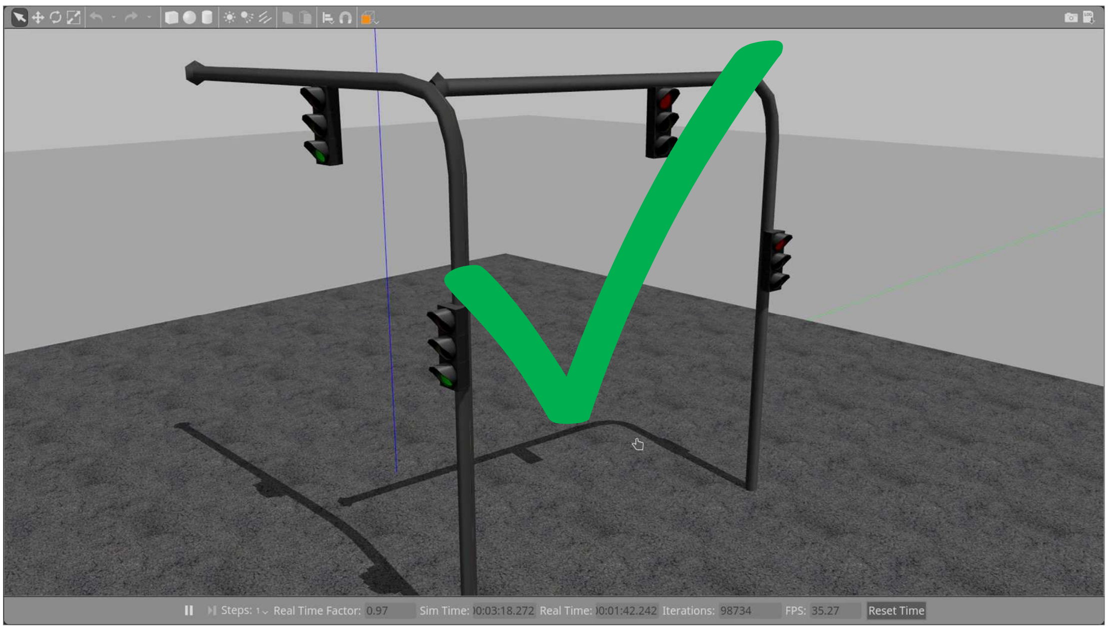



```
msgsrv RedtoGreen() {
    Color = 1; //Green
    driftedDelay = 2;
    delay(driftedDelay);
    self.GreentoYellow();
}
```

```
msgsrv YellowtoRed() {
    Color = 0; //Red
    driftedDelay = 4;
    delay(driftedDelay);
    self.RedtoGreen();
}
```

$\varphi_1 = G(\neg(green_1 \ \wedge \ green_2))$ → NO CONCURRENT GREEN ✓

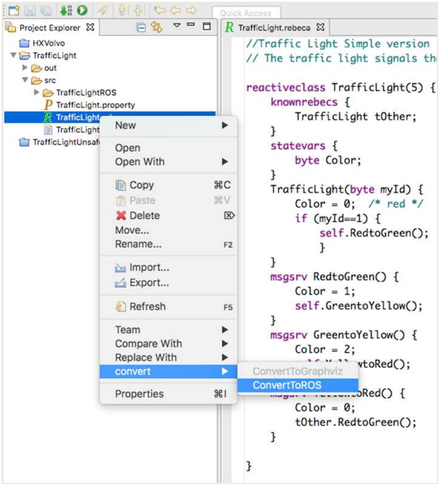
Timed Rebeca Model: Traffic Lights
Different shift in time



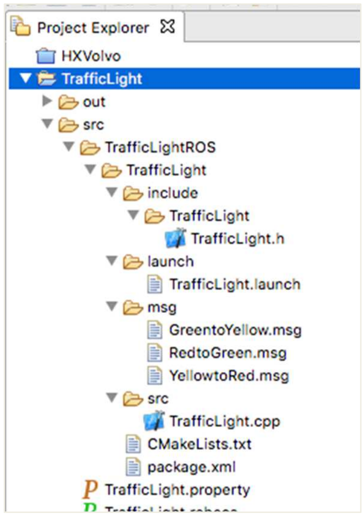
$\varphi_1 = G(\neg(green_1 \ \wedge \ green_2))$ → NO CONCURRENT GREEN ✓

Rebeca Traffic Lights Model to ROS

Automatic conversion from Rebeca specification to ROS with Afra 3.0



reactiveclass → ROS class (robot)
rebecs → ROS nodes
message servers → ROS topics
message parameters → ROS msgs
sending messages → publish on topic
receive messages → subscribe on topic

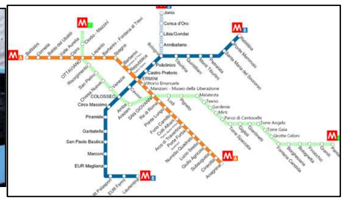


Flow Management

Flow Managemnet of Track-based Applications

Warehouse Management System

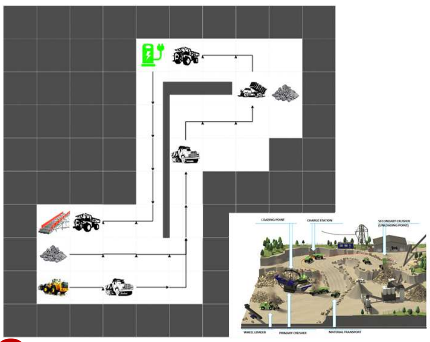
Public Transportation System



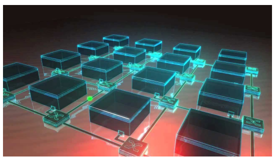
Transporting Shuttles in a Close Environment

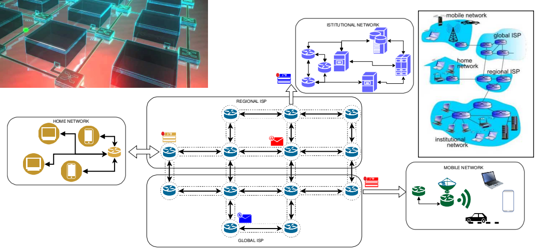
Airport and Airspace Systems

Flow Management: An Abstract View

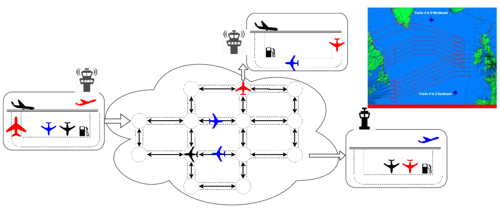


Quarry

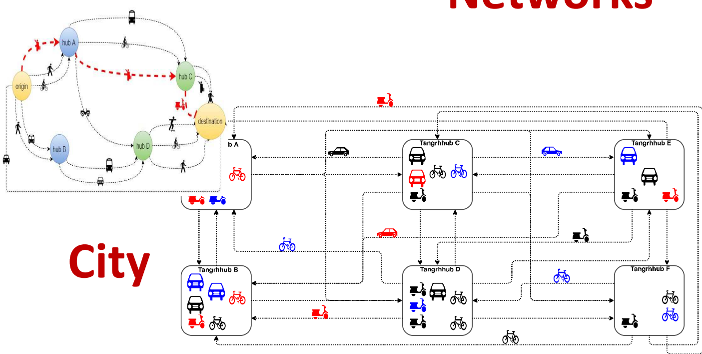




Networks

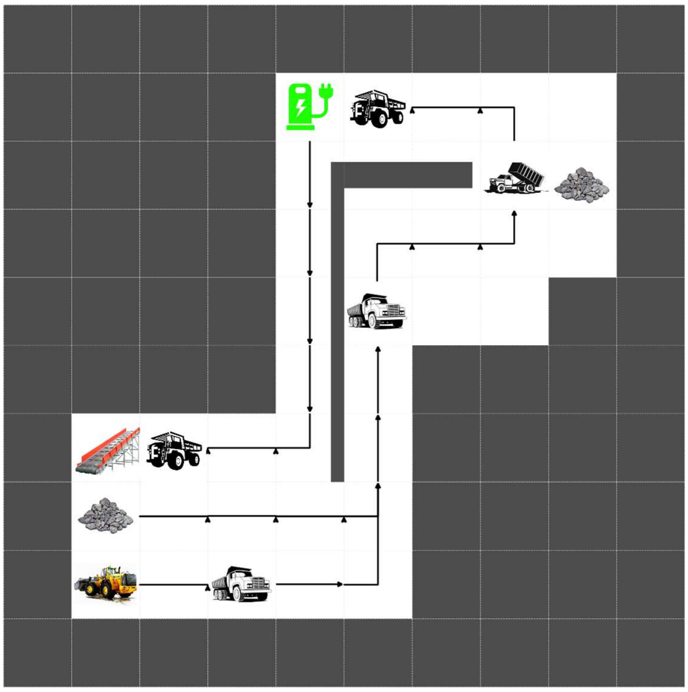


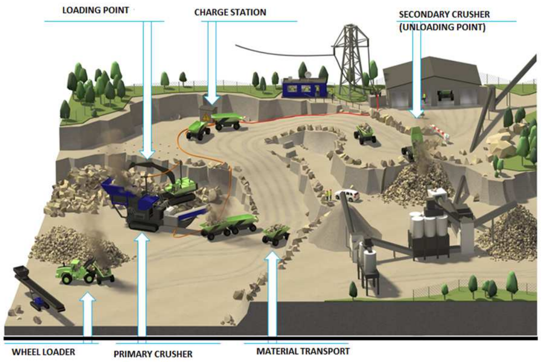
Air Space



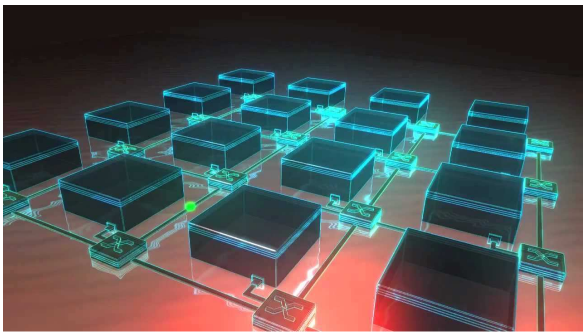
City

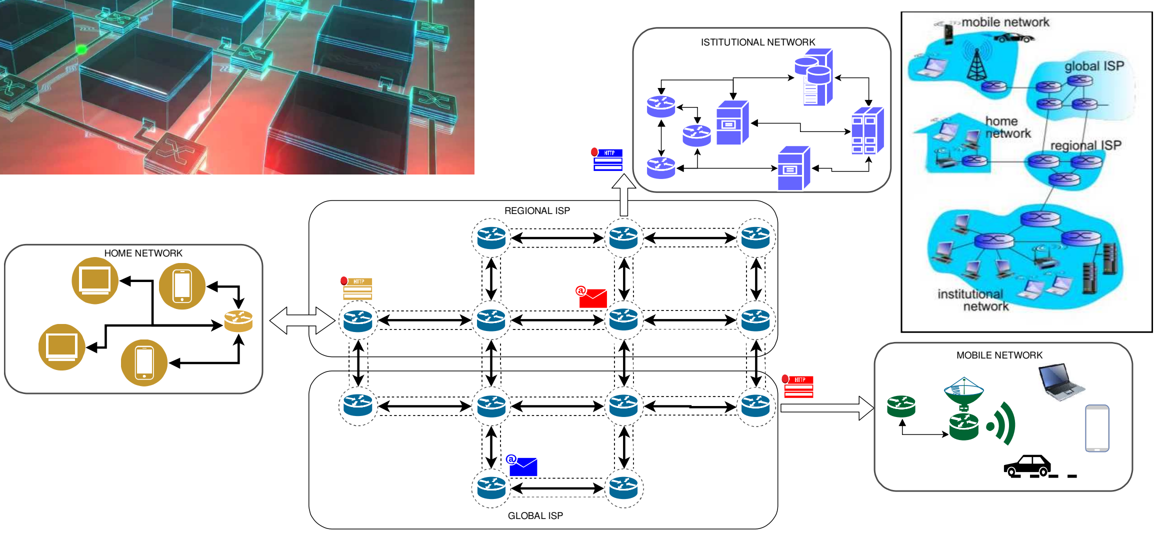
Quarry



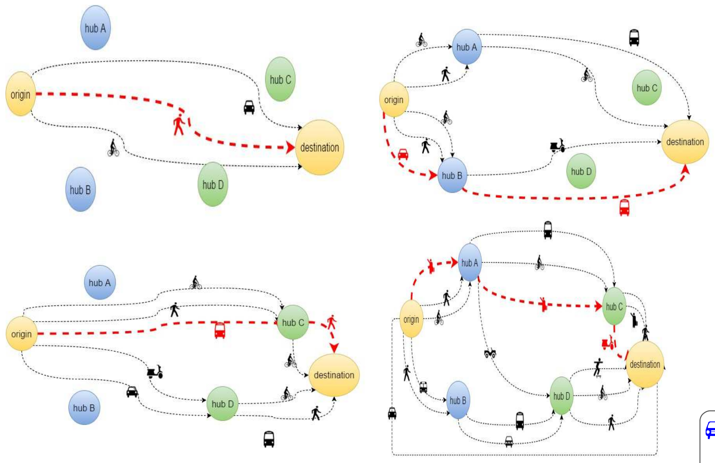


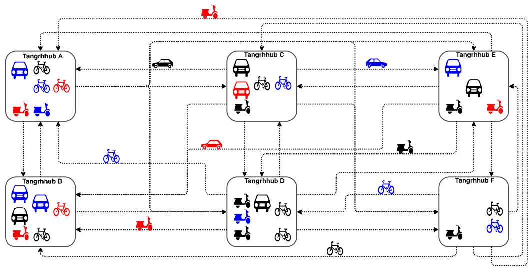
Network on Chip (NoC)



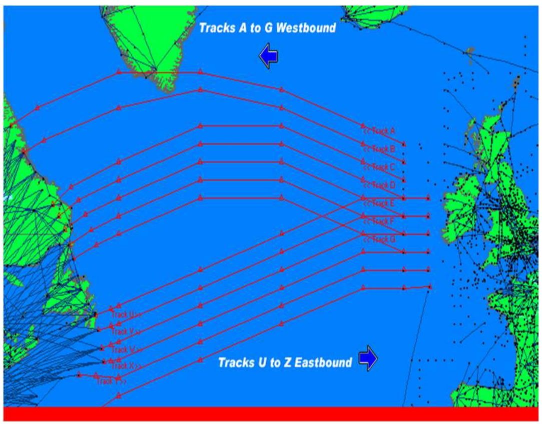


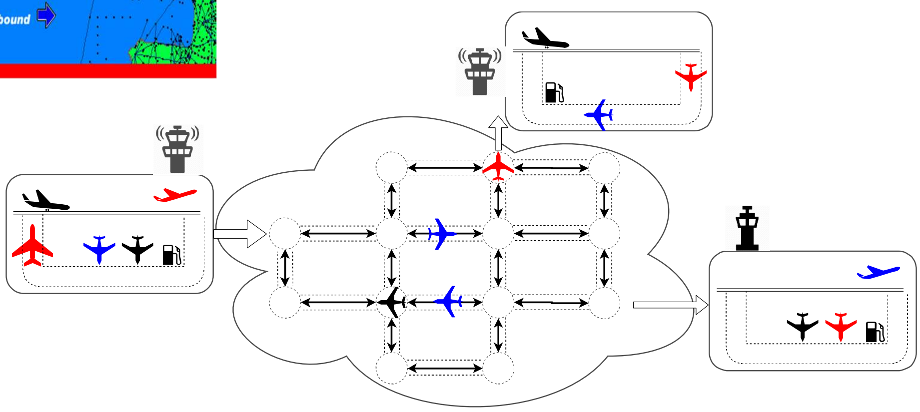
Smart Transport Hubs



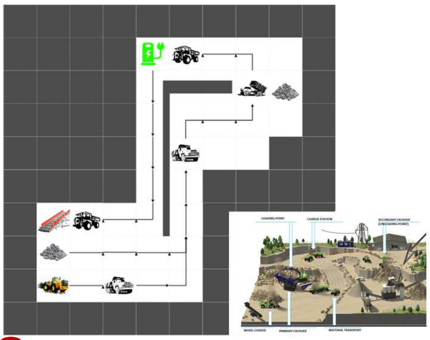


Air Space

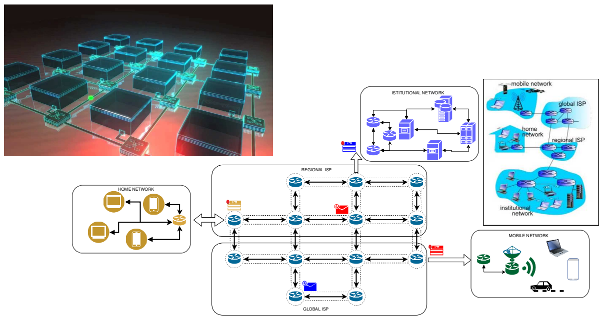




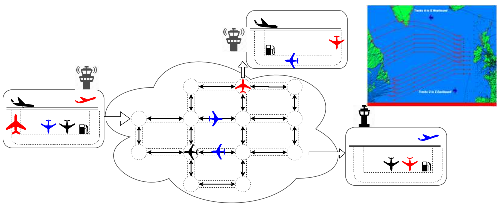
Track-based Flow Management



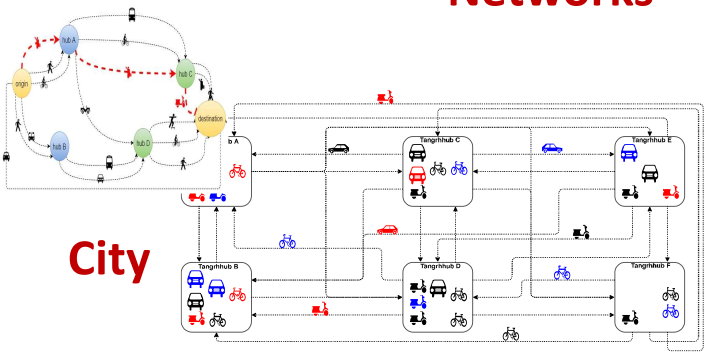
Quarry



Networks



Air Space



City

Similar Pattern:
Flow of objects on tracks

Topology

- Sources
- Destinations
- Intermediate Destinations
  - Charging stations
  - Bus stations
  - Hubs

Configuration, design variables and constraints

- Capacity
  - Bandwidth
- Speed
- Latency / Time
- Cost

Goals

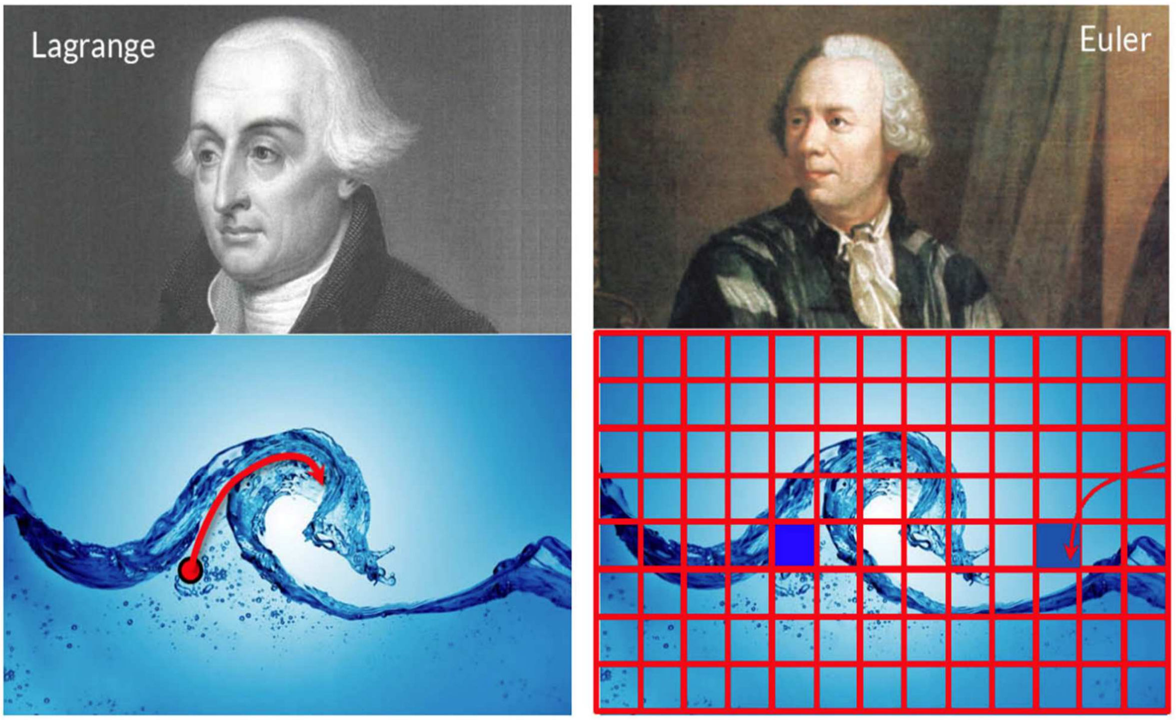
- Minimum Time
- Minimum Fuel
- Maximum Throughput
- …

Analysis

- Safety
- Optimization and Performance Analysis

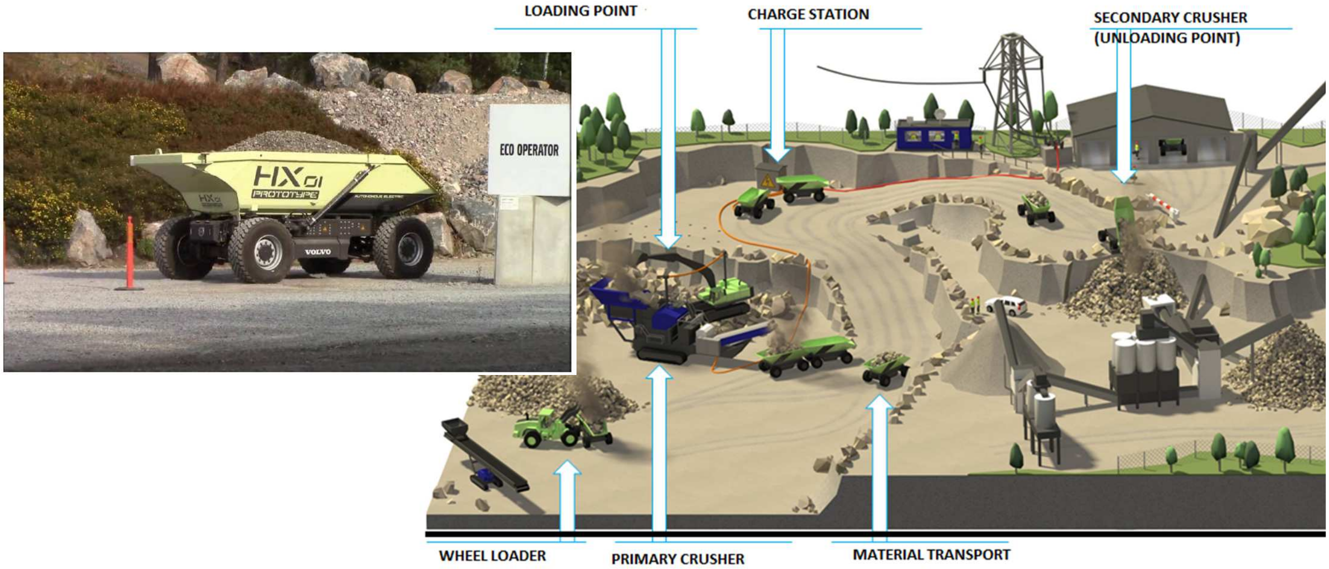
- Self-Adaptation

In Physics (classical field theory)

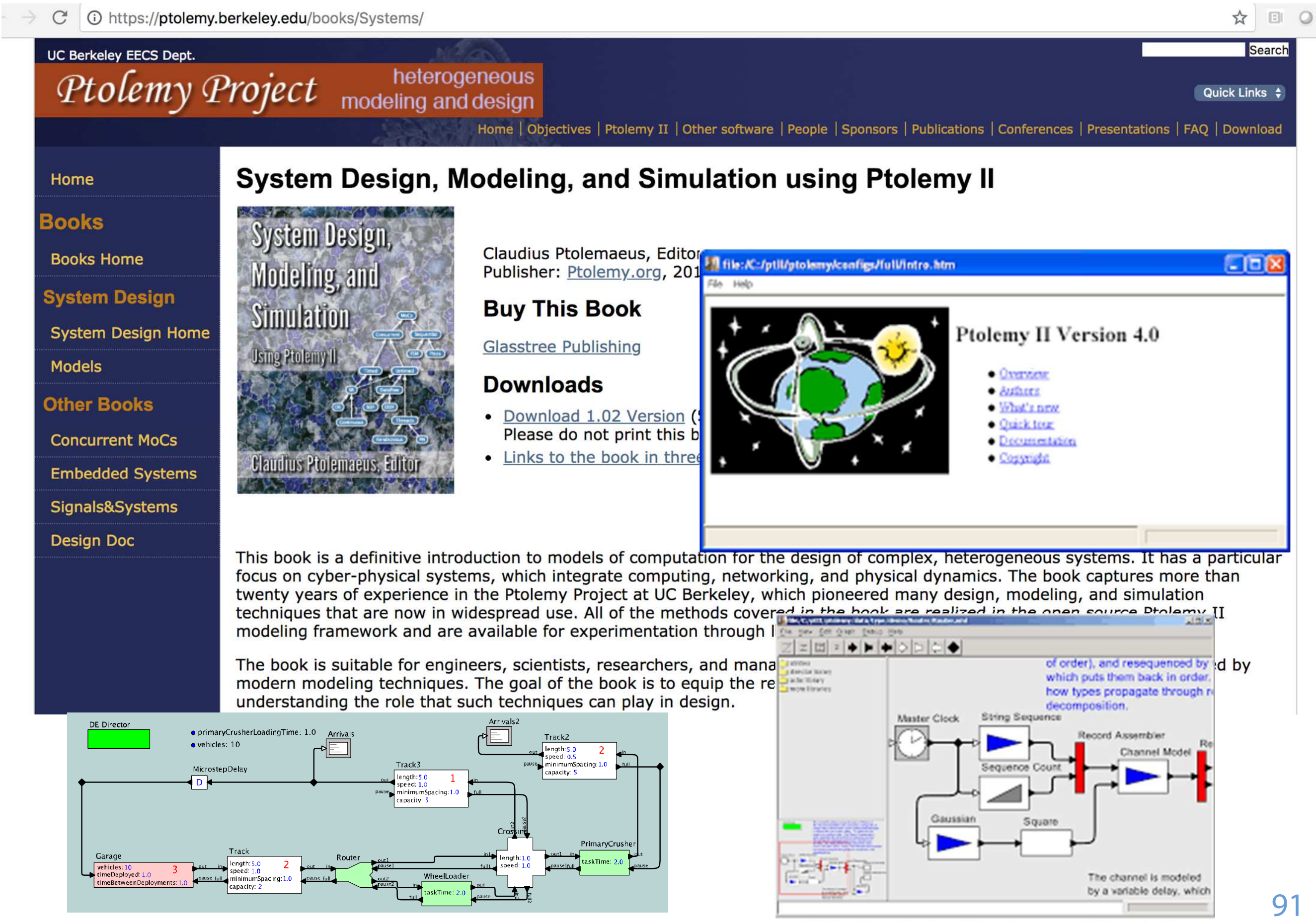


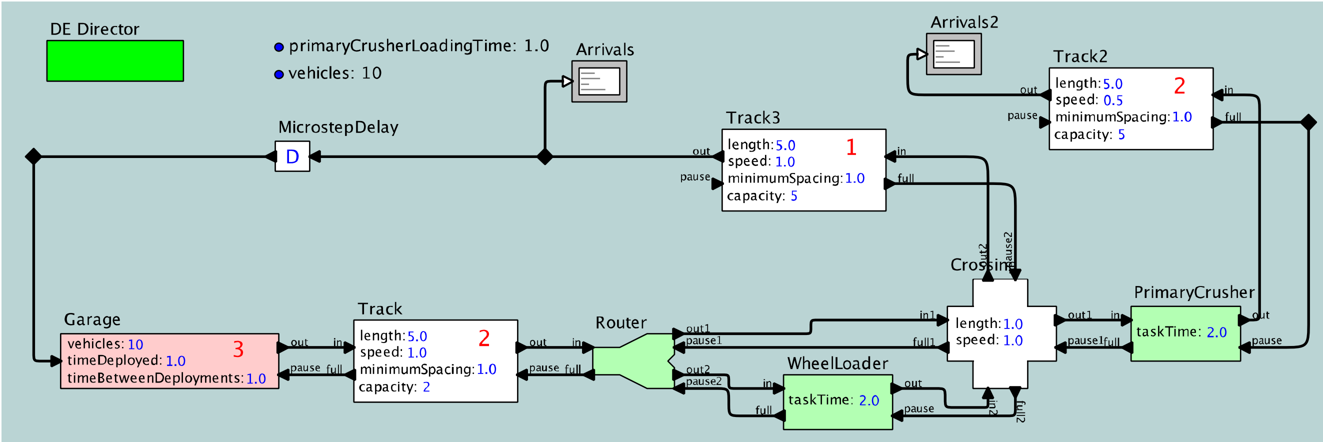
May need a copyright

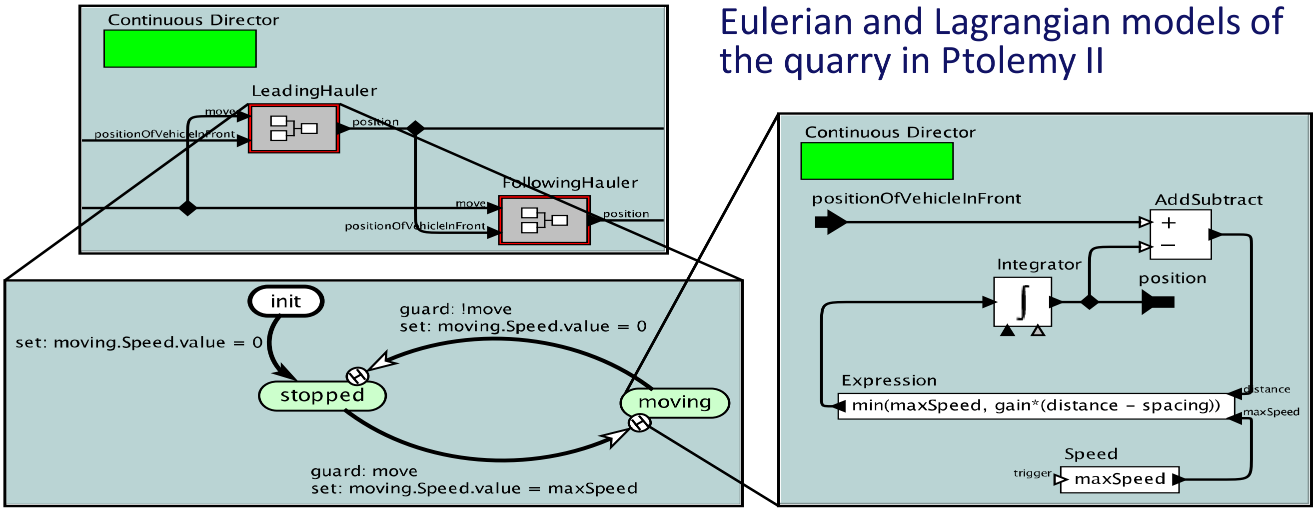
VCE Automated Quarry



Courtesy of Volvo CE

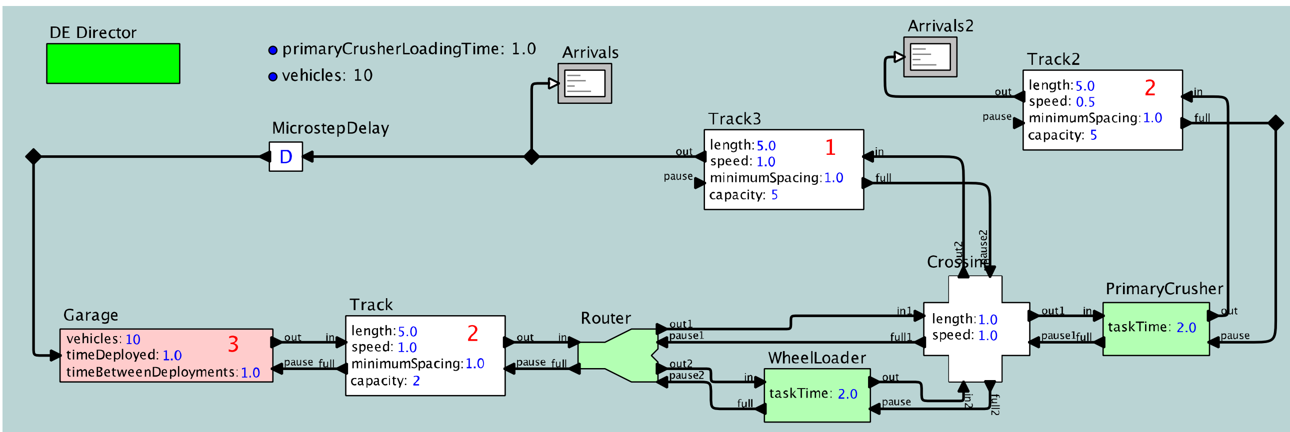






Eulerian and Lagrangian models of the quarry in Ptolemy II

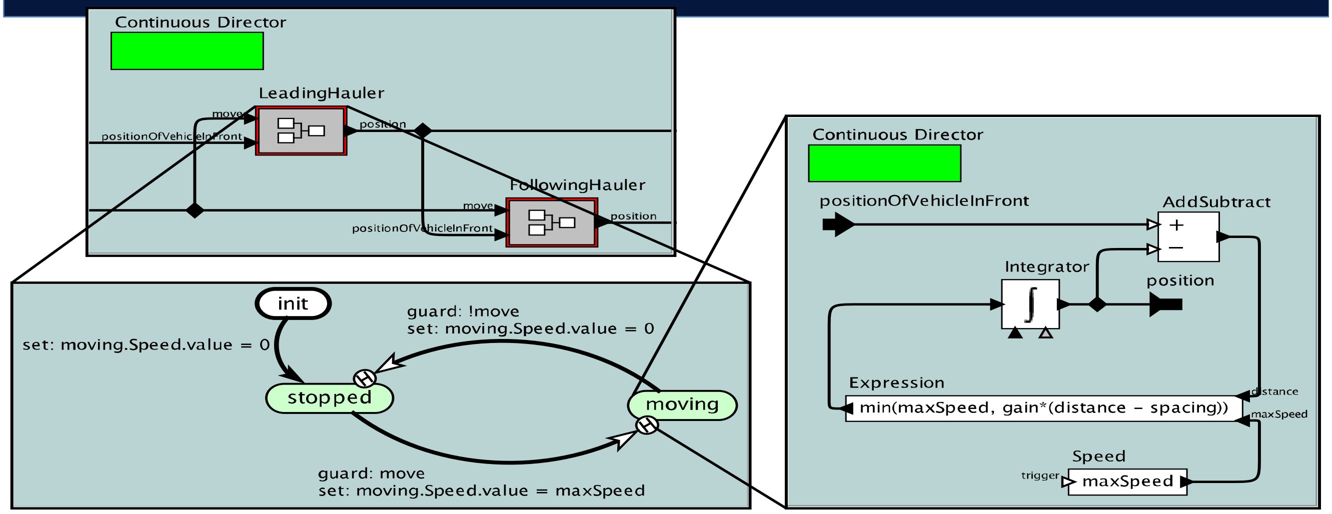
Eulerian Model
Actors are Tracks and Worksites



Use this model to study or design:

- Trajectory planning
- Resource optimization
- Affects of disruptions

Lagrangian Model
Actors are Haulers



Use this model to study or design:

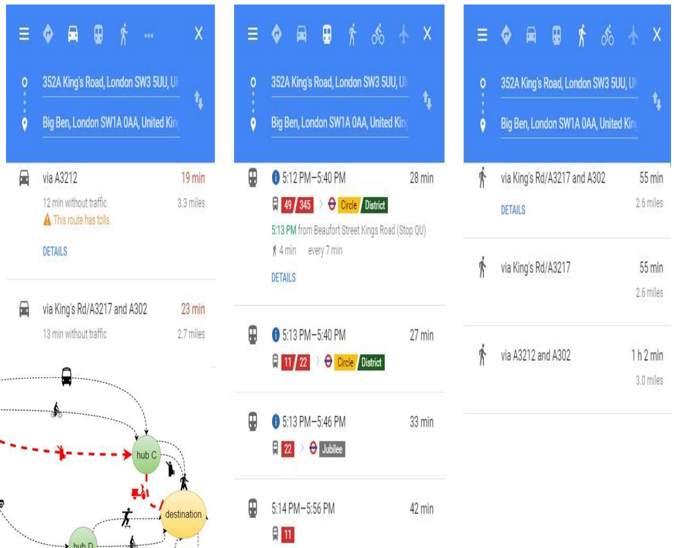
- Collision avoidance
- Sensor performance
- Battery usage

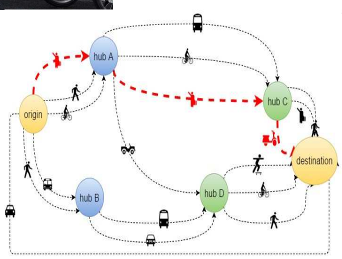
Other Projects

SmartHub Project

(Unicam Smart Mobility Lab, Andrea Polini and Francesco De Angelis)

- Smart Hubs are Local **container** of one or more smart mobility services

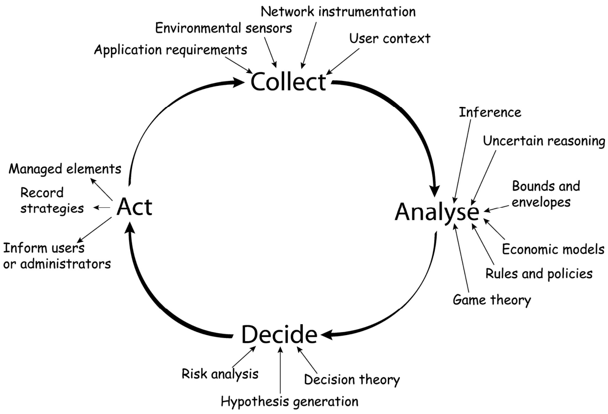




Goals in Smart Transportation Hub

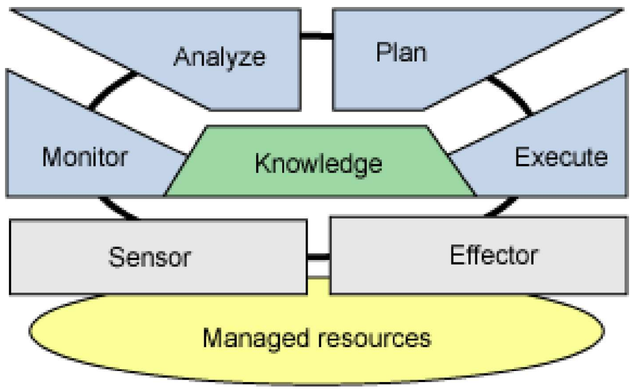
- Minimize:
  – number of service disruptions
  – number of mobility resources in smarthubs
  – cost of mobility for commuters
  – travel time for commuters
  – travel distance for commuters

Adaptive Track-based Traffic Control



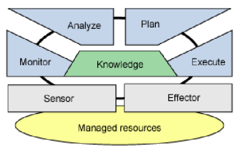
Typical Autonomic Control Loop

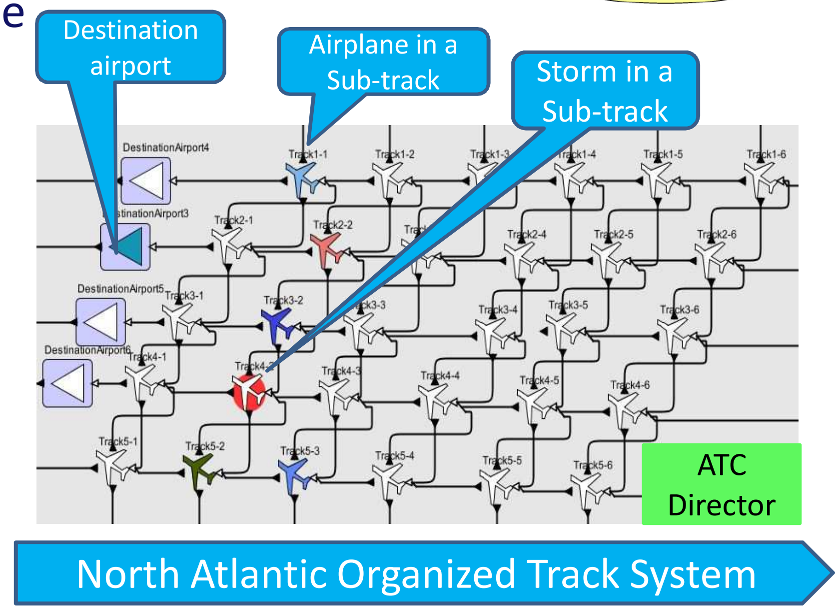
IBM MAPE-K

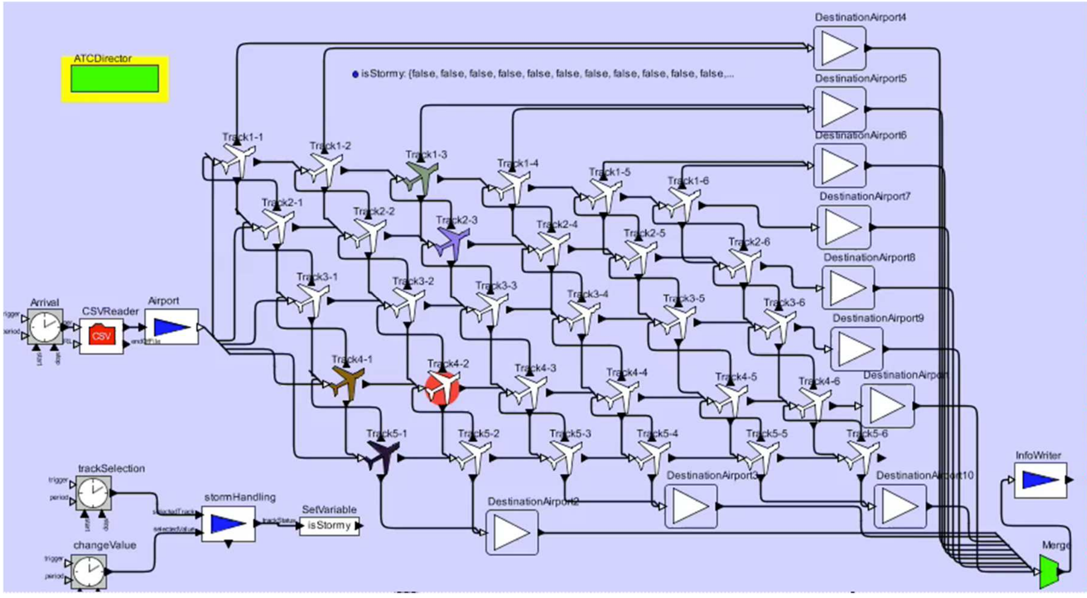


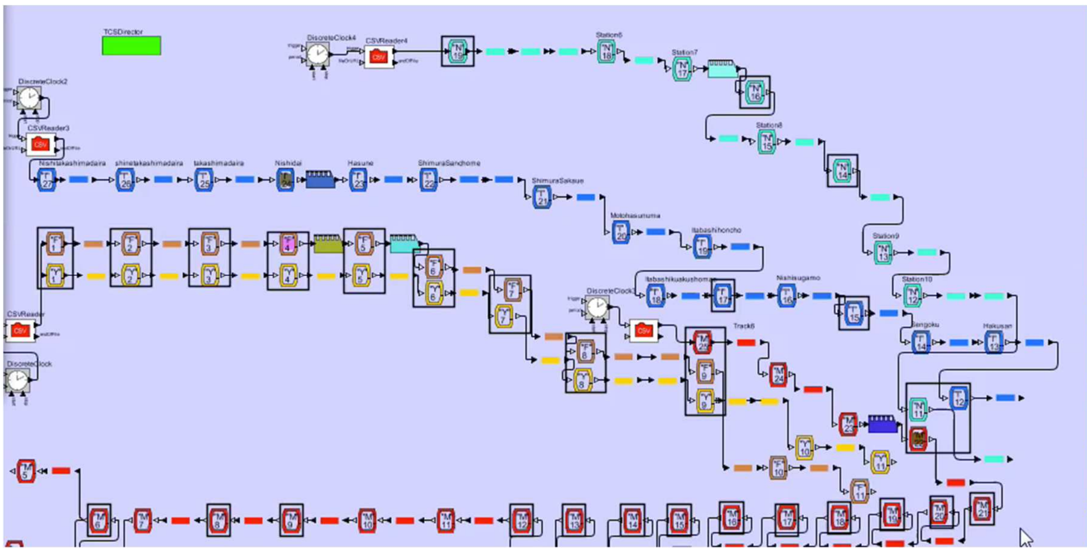
Dependable Self-Adaptive Actors

- Coordinated Actors in Ptolemy
- Model Change and Handle Rerouting
- Use model@runtime

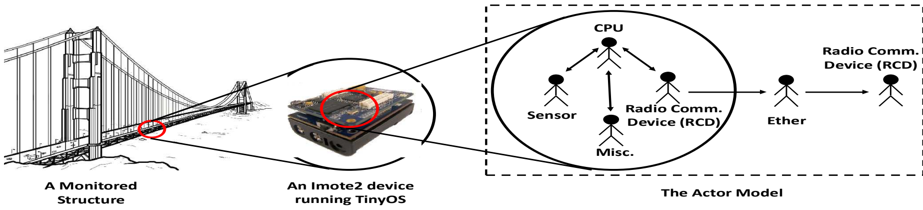




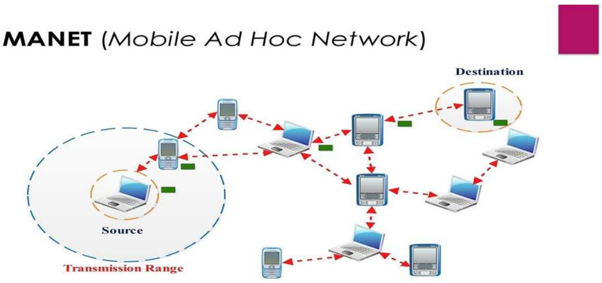

ATCDirector
isStormy: {false, false, false, false, false, false, false, false, false, false, false,...
Track1-1
Track1-2
Track1-3
Track1-4
Track1-5
Track1-6
Track2-1
Track2-2
Track2-3
Track2-4
Track2-5
Track2-6
Track3-1
Track3-2
Track3-3
Track3-4
Track3-5
Track3-6
Track4-1
Track4-2
Track4-3
Track4-4
Track4-5
Track4-6
Track5-1
Track5-2
Track5-3
Track5-4
Track5-5
Track5-6
DestinationAirport4
DestinationAirport5
DestinationAirport6
DestinationAirport7
DestinationAirport8
DestinationAirport9
DestinationAirport2
DestinationAirport3
DestinationAirport10
Arrival
CSVReader
CSV
Airport
trackSelection
changeValue
stormHandling
SetVariable
isStormy
InfoWriter
Merge


TCSDirector
DiscreteClock4
CSVReader4
Station6
Station7
Station8
Station9
Station10
DiscreteClock2
CSVReader3
Nishitakashimadaira
shintakashimadaira
takashimadaira
Nishidai
Hasune
ShimuraSanchome
ShimuraSakaue
Motohasunuma
Itabashihoncho
Nishisugamo
DiscreteClock3
Track6
CSVReader
DiscreteClock

- **Schedulability Analysis - Wireless Sensor Networks**



- **Correctness of Network Protocols**



References

- For publications, see

http://rebeca-lang.org/publications

- For projects, see

http://rebeca-lang.org/projects