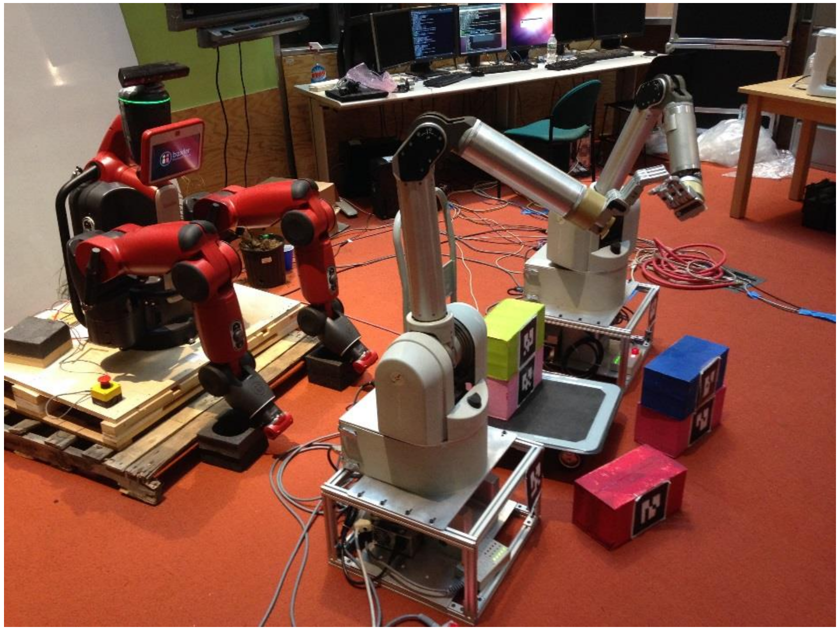
# Cognitive Robotics

## I.e., How to create "thinking" robots



Slide contributions:
Jacob Beal

Prof. Brian Williams and Steve Levine (TA)

*Introduction*

February 3rd, 2015.

# Today’s Assignments

## Problems Sets:

- None!  Problem Set #1 out next Monday, due in 1½ weeks .

## Readings:

- **Today**: Williams, B. C., *et. al.,* “Model-based Programming of Fault-Aware Systems,” AI Magazine, 24, pp. 61-75, 2004.
- **Next:** Williams, B. C. and Ragno, R. “Conflict-directed A* and its Role in Model-based Embedded Systems,” Special Issue on Theory and Applications of Satisfiability Testing, Journal of Discrete Applied Math, 155, pp. 1562-1595, 2003.

## Note:

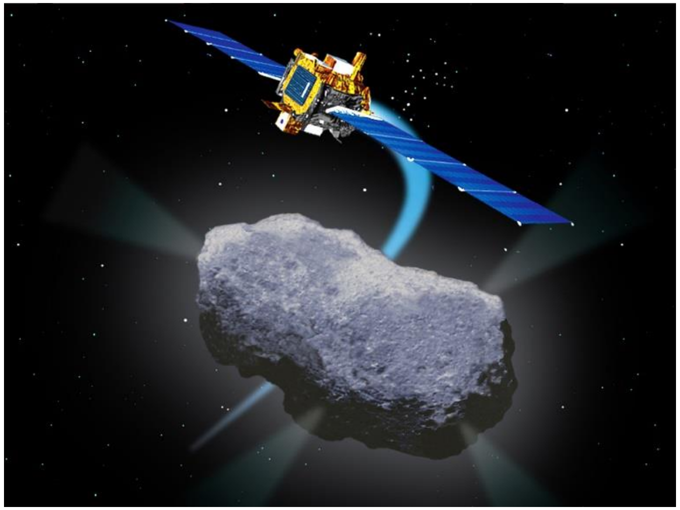
- Problem sets, readings and lecture slides posted on 16.412J course site.
- Background: open courseware, 16.410/13 Principles of Autonomy and Decision Making.

# Outline

- **Course in a Nutshell**
- Course Structure and Logistics
- Programming Cognitive Robots
- Self-Adaptive and Self-Repairing Systems
- Programs that Monitor State

# Coordinating Network Embedded Systems

- We are creating vast networks of embedded systems that perform critical functions over long periods of time.
- These long-lived systems achieve robustness by coordinating a complex network of devices.
- Programming these systems robustly is becoming an increasingly daunting task.





00:00 Go to $x_1,y_1$
00:20 Go to $x_2,y_2$
00:40 Go to $x_3,y_3$
…
04:10 Go to $x_n,y_n$

**Command script**

**Commands**

**Plant**

*"Put out the fires at Locations 1 and 2, and return to the base in an hour. Avoid no fly zones. Here is a map, including reservoirs and gas stations."*

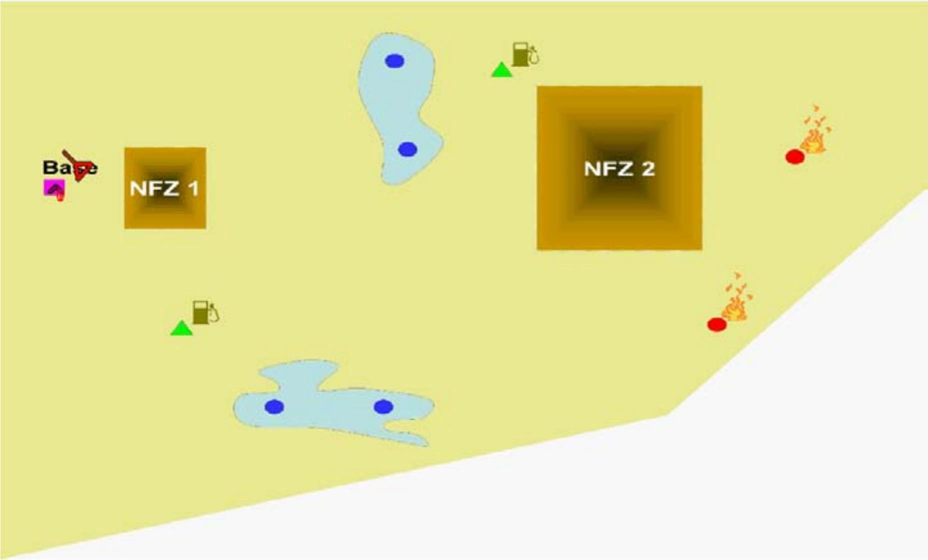
**Qualitative State Plan**

**Model-based Executive**

**Observations**

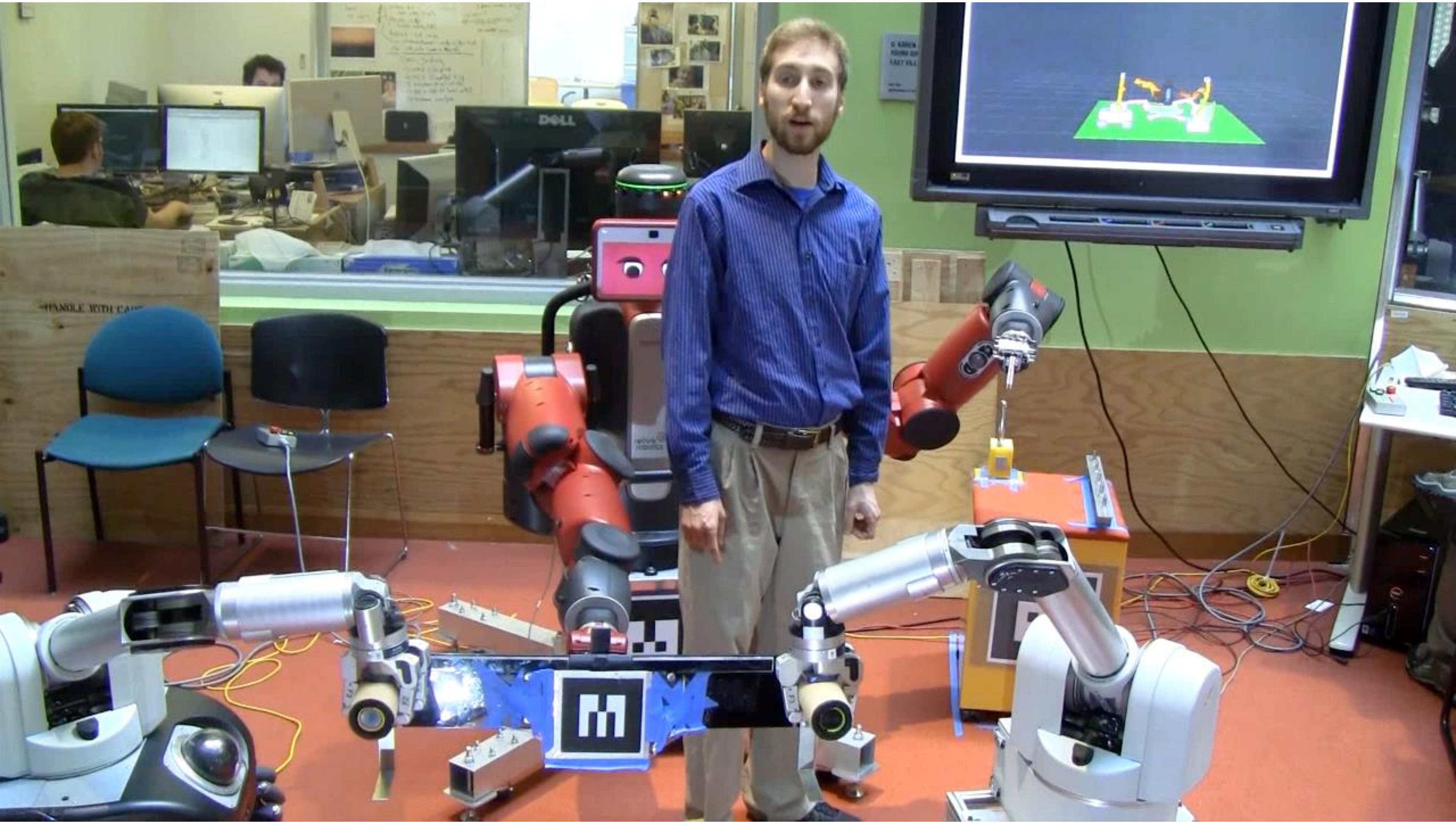
**Commands**

**Plant**




[Leaute & Williams, AAAI 05]

# Robustness through Collaboration

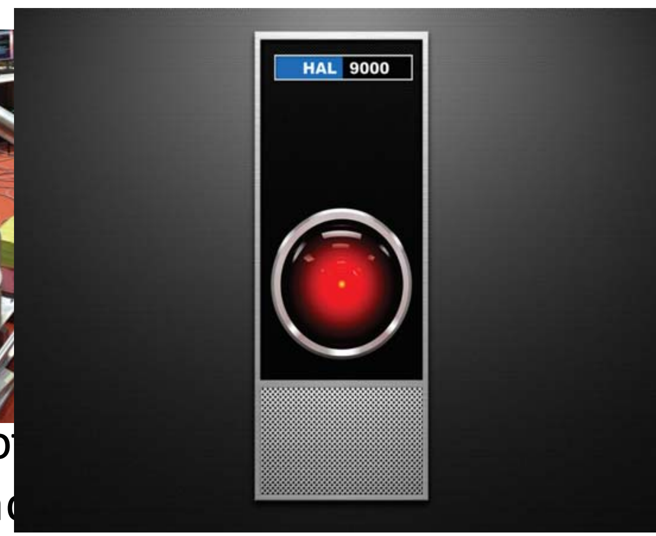

DELL

# Examples of Cognitive Robotic Systems

es:
rected
nt
nsitive
rative
formation

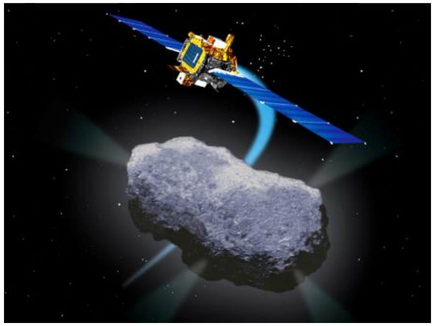
Information-gath
Scouts


Human-Robo
For Manufa




Grids
Homes

# Programming Cognitive Systems





1. Embedded programming languages elevated to the goal-level through partial specification and operations on state (RMPL).
2. Language executives that achieve robustness by reasoning over constraint-based models and by bounding risk (Enterprise).
3. Interfaces to support human interaction fluidly and at the cognitive level (Uhura, Pike ...).

# Outline

- Course in a Nutshell
- **Course Structure and Logistics**
- Programming Cognitive Robots
- Self-Adaptive and Self-Repairing Systems
- Programs that Monitor State

# Crash Course in Autonomy

## I.e., Programming Cognitive Robots


Prof. Brian Williams & Eric Timmons
Erez Karpas, Andrew Wang, Peng Yu, Steve Levine, Pedro Santana, Simon Fang, Enrique Gonzales, David Wang and Peng Yu.

*Introduction*

January 12, 2015.

# About

- How to program cognitive robots
  in terms of goals, to perform complex tasks.

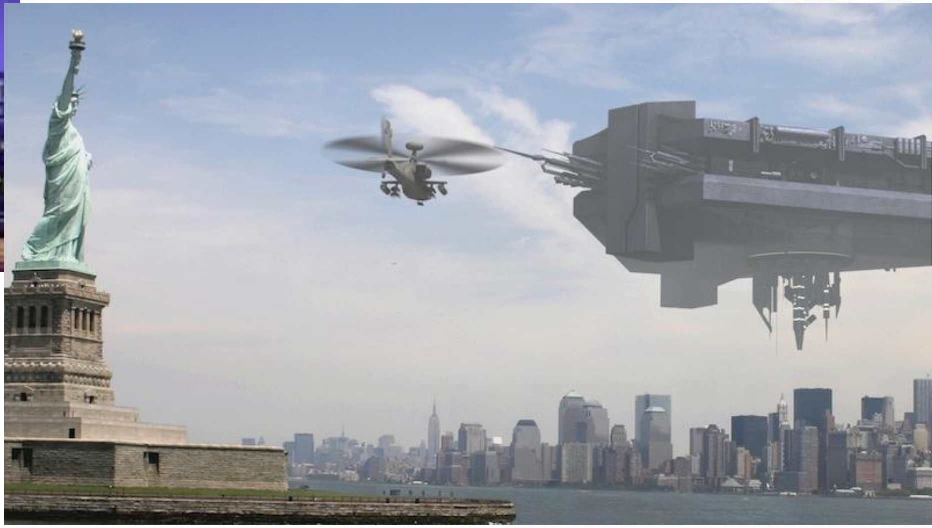
- Intuitions underlying how robots “reason.”

- ~~Exposure to basic computational concepts.~~

➢ 16:412J: State-of-the-art reasoning methods.

# Driven by a Grand Challenge





# Embedded in simulations and hardware

Working out-of-the-box with your ‘15 holiday gift

# 16:412J: Driven by a Grand Challenge

RobOrienteering

# 16:412J: Embedded in simulations and hardware

Works with a lot of elbow grease and a bit of luck

Traditional Programming

TODAY

TIME

- Programs that Monitor State
- Programs with Flexible Time
- Programs with Goal States
- Programs with Continuous State
- Programs that Collaborate
- Advanced lectures…
- Risk-bounded Programming
- Grand Challenge

“Cognitive” Programming

END OF SEMESTER

Reactive Model-based Programming Language (RMPL)

+ Python

# You’ll walk out knowing how to

- program cognitive robots by specifying goals and models
- Research, describe, and implement decision-making algorithms that enable robots to monitor, plan and coordinate in the real world
- implement complex missions with real robot.

# Be warned – bleeding edge!

A clipart of blooded knife removed due to copyright restrictions.

Policies, grading, and assignments

# COURSE LOGISTICS

# Websites & emails

- Stellar website
- Piazza (for course-related discussion)
- Staff email list

# Lectures & Office Hours

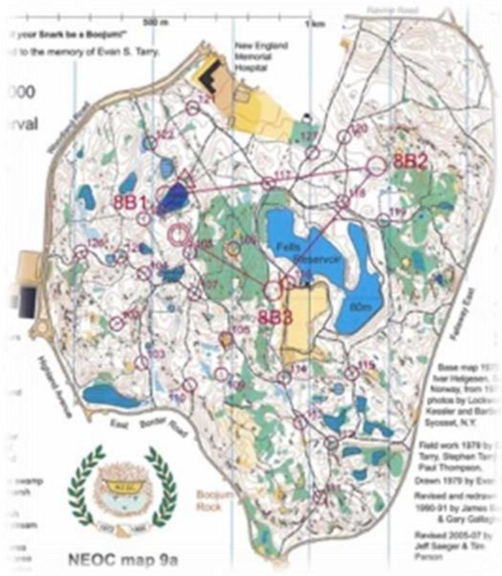
- **Students must attend lectures**
  - *Vital* to learning course material
  - Plan on attending all lectures
  - Expected to do assigned readings before lecture
- Randomly-scheduled 5-minute mini quizzes
  - Not difficult / stressful

# Assignments: Problem sets

- Modeling exercises
- Using existing autonomy tools
- Implementing algorithms (Python)

# Assignments: Grand Challenge

- Course culminates in grand challenge!
  - Orienteering theme
  - Simulation & real hardware

# Assignments: Advanced Lectures

- What it's like doing research in autonomy
- Teams of 5-6 will:
  - Present full 80 minute lecture on researched topic
  - Implement topic (Python)
  - Release code, API, and tutorial / documentation for class
    - Will be used in grand challenge

# Electronics use policy in lecture

Computers, tablets: ***for note taking only!***

Please do not:

- Check email or facebook, surf web, watch adorable cat videos 😿, etc.

Research shows it also **distracts others** nearby

# Grading

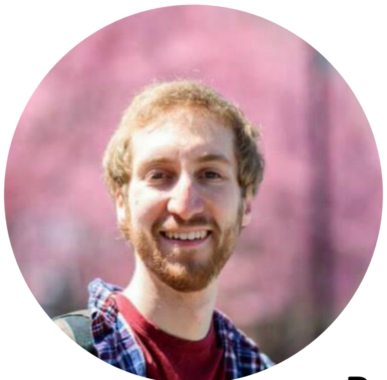
| Item | Weight |
|---|---|
| Participation & attendance (mini quizzes) | 10% |
| Problem sets | 40% |
| Advanced lecture & implementation | 30% |
| Grand challenge | 20% |

* Staff reserves the right to consider other factors & adjust formula

# Collaboration Policy

- Collaboration allowed, such that you:
  - Acknowledge collaborators
  - Involved in all aspects of work (no dividing up)
  - Write your own solutions
- Advanced lectures & grand challenge
  - Working in teams, expect equal contributions
- Course bibles prohibited

# TA: Steve Levine



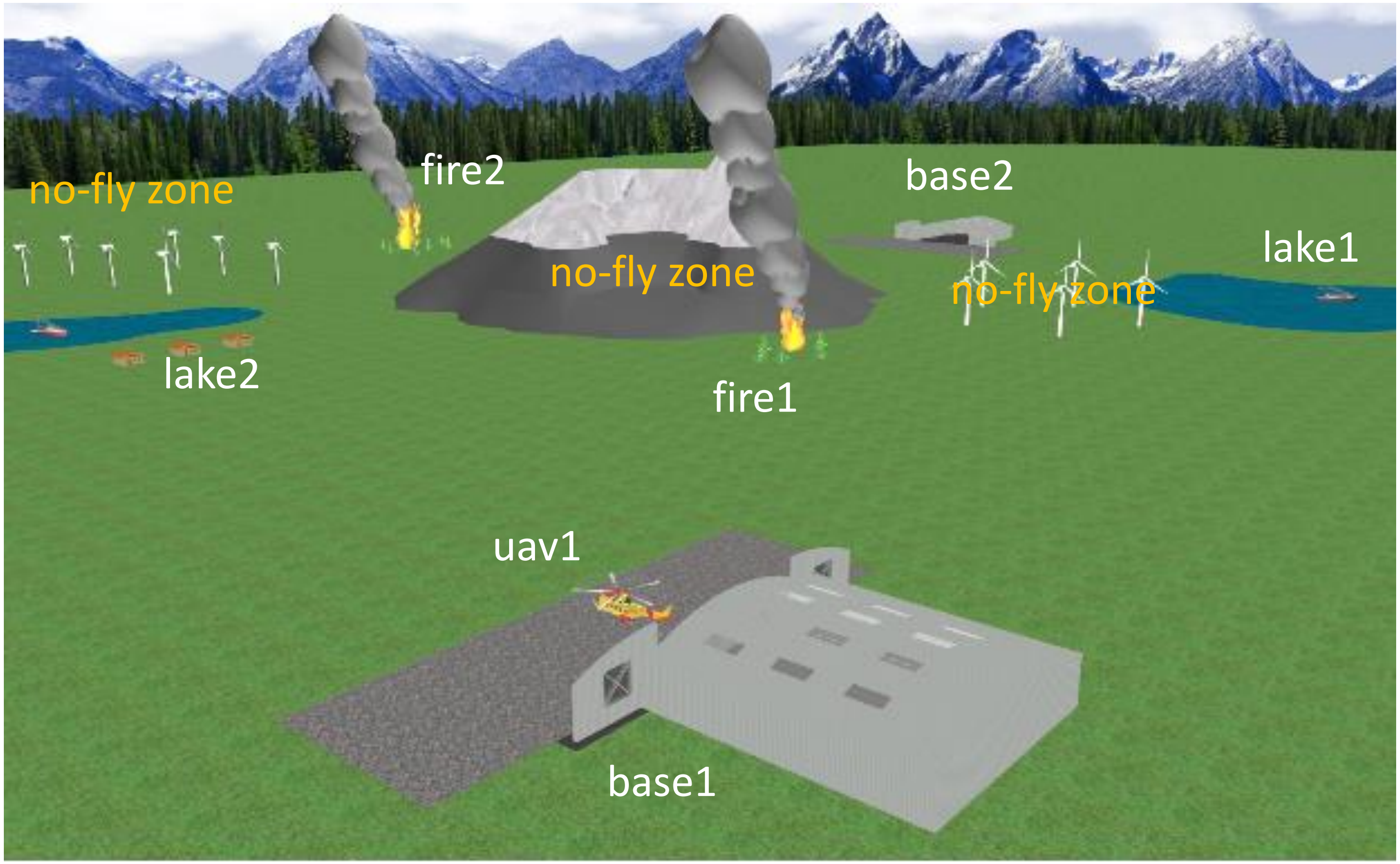
$4^{th}$ year Ph.D student in CS (MERS group)
**Research**:
Intent recognition & adaptation for robots

B.S. from MIT in ‘11, course 6
M.Eng from MIT in ‘12, course 6
$9^{th}$ year (!) at MIT! So old!

# Outline

- Course in a Nutshell
- Logistics
- **Programming Cognitive Robots**
- Self-Adaptive and Self-Repairing Systems
- Programs that Monitor State

# Programs on State - Firefighting Scenario



# Firefighting in RMPL: Traditional Imperative program

```
class Main{
  UAV uav1;
  Lake lake1;
  Lake lake2;
  Fire fire1;
  Fire fire2;
      ...

 method run() {
  sequence{
    uav1.takeoff();
    uav1.fly_base1_to_lake1();
    uav1.load_water(lake1);
    uav1.fly_lake1_to_fire1();
    uav1.drop_water_high_altitute(fire1);
    uav1.fly_fire1_to_lake1();
    uav1.load_water(lake1);
    uav1.fly_lake1_to_fire1();
    uav1.drop_water_low_altitute(fire1);
    uav1.fly_fire1_to_lake2();
    uav1.load_water(lake2);
    uav1.fly_lake2_to_fire2();
    uav1.drop_water_high_altitute(fire2);
    uav1.fly_fire2_to_lake2();
    uav1.load_water(lake2);
    uav1.fly_lake2_to_fire2();
    uav1.drop_water_low_altitute(fire2);
    uav1.fly_fire2_to_base1();
    uav1.land();
  }
}}
```

# Firefighting in RMPL: A Program on State

```
class Main{
 UAV uav1;
 Lake lake1;
 Lake lake2;
 Fire fire1;
 Fire fire2;
     ...

 method run() {
   sequence{
     (fire1 == out);
     (fire2 == out);
     (uav1.flying == no &&
      uav1.location == base_1_location);
   }
 }

}
```

# Firefighting in RMPL: Setup & Initial Conditions

```
class Main{
  UAV uav1;
  Lake lake1;
  Lake lake2;
  Fire fire1;
  Fire fire2;
     ...

 method run() {
   sequence{
     (fire1 == out);
     (fire2 == out);
     (uav1.flying == no &&
      uav1.location == base_1_location);
   }
 }

}
```

```
Main (){
     uav1 = new UAV();
     uav1.location= base_1_location;
     uav1.flying = no;
     uav1.loaded = no;

     lake1 = new Lake();
     lake1.location = lake_1_location;

     lake2 = new Lake();
     lake2.location = lake_2_location;

     fire1 = new Fire();
     fire1.location = fire_1_location;
     fire1 = high;

     fire2 = new Fire();
     fire2.location = fire_2_location;
     fire2 = high;
}
```

# Firefighting in RMPL: Model of Actions

```
class UAV {
  Roadmap location;
  Boolean flying;
  Boolean loaded;

  primitive method takeoff()
    flying == no => flying == yes;

  primitive method land()
    flying == yes => flying == no;

  primitive method load_water(Lake lakespot)
    ((flying == yes) && (loaded == no) && (lakespot.location == location)) => loaded == yes;

  primitive method drop_water_high_altitude(Fire firespot)
    ((flying == yes) && (loaded == yes) && (firespot.location == location) && (firespot == high))
     => ((loaded == no) && (firespot == medium));

  primitive method drop_water_low_altitude(Fire firespot)
    ((flying == yes) && (loaded == yes) && (firespot.location == location) && (firespot == medium))
     => ((loaded == no) && (firespot == out));

  #MOTION_PRIMITIVES(location, fly, flying==yes)
}
```

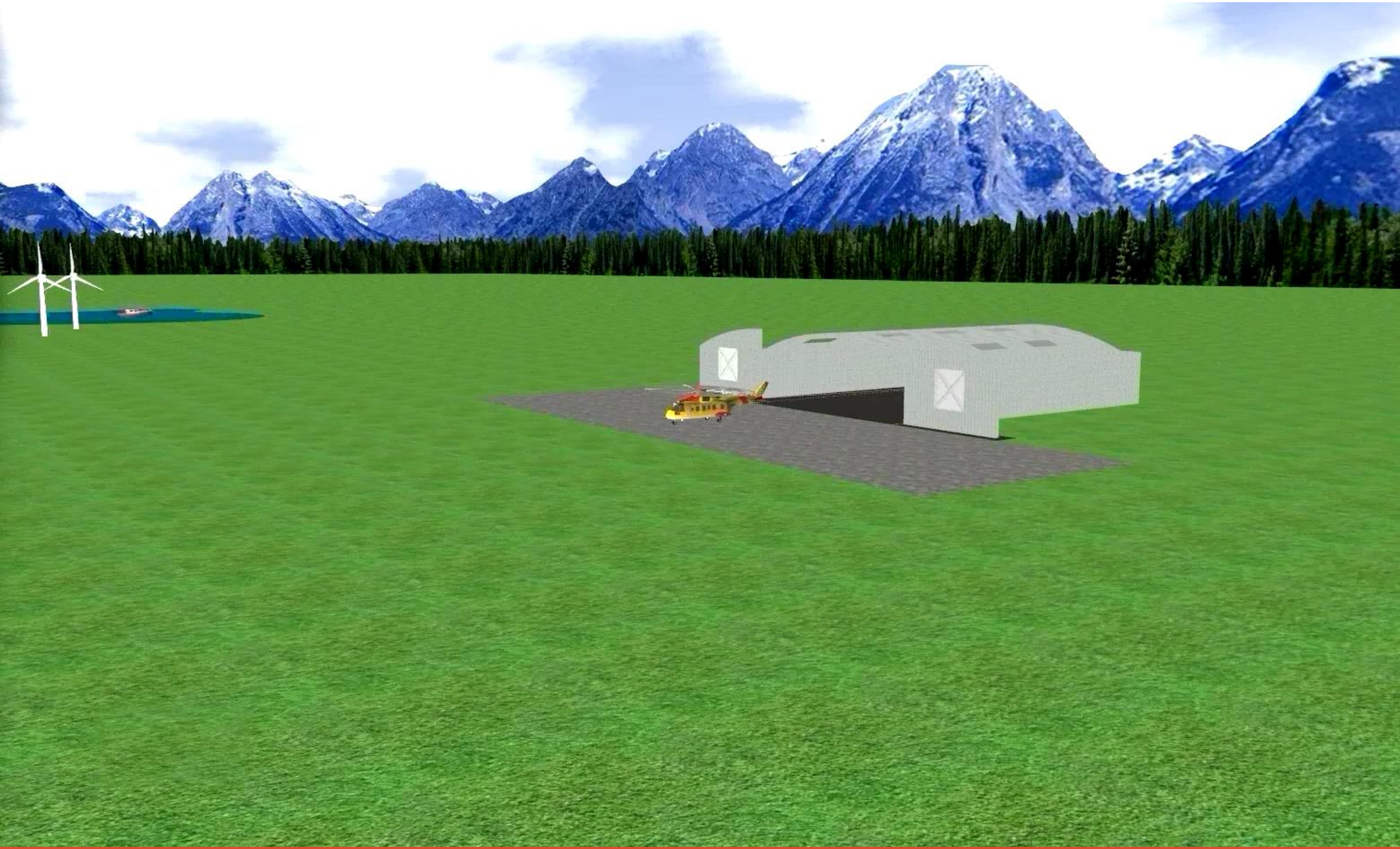
```
class Lake {
   Roadmap location;
}

class Fire{
   initial value high;
   value medium;
   value out;
   Roadmap location;
}
```

# Firefighting in RMPL: Model of Actions

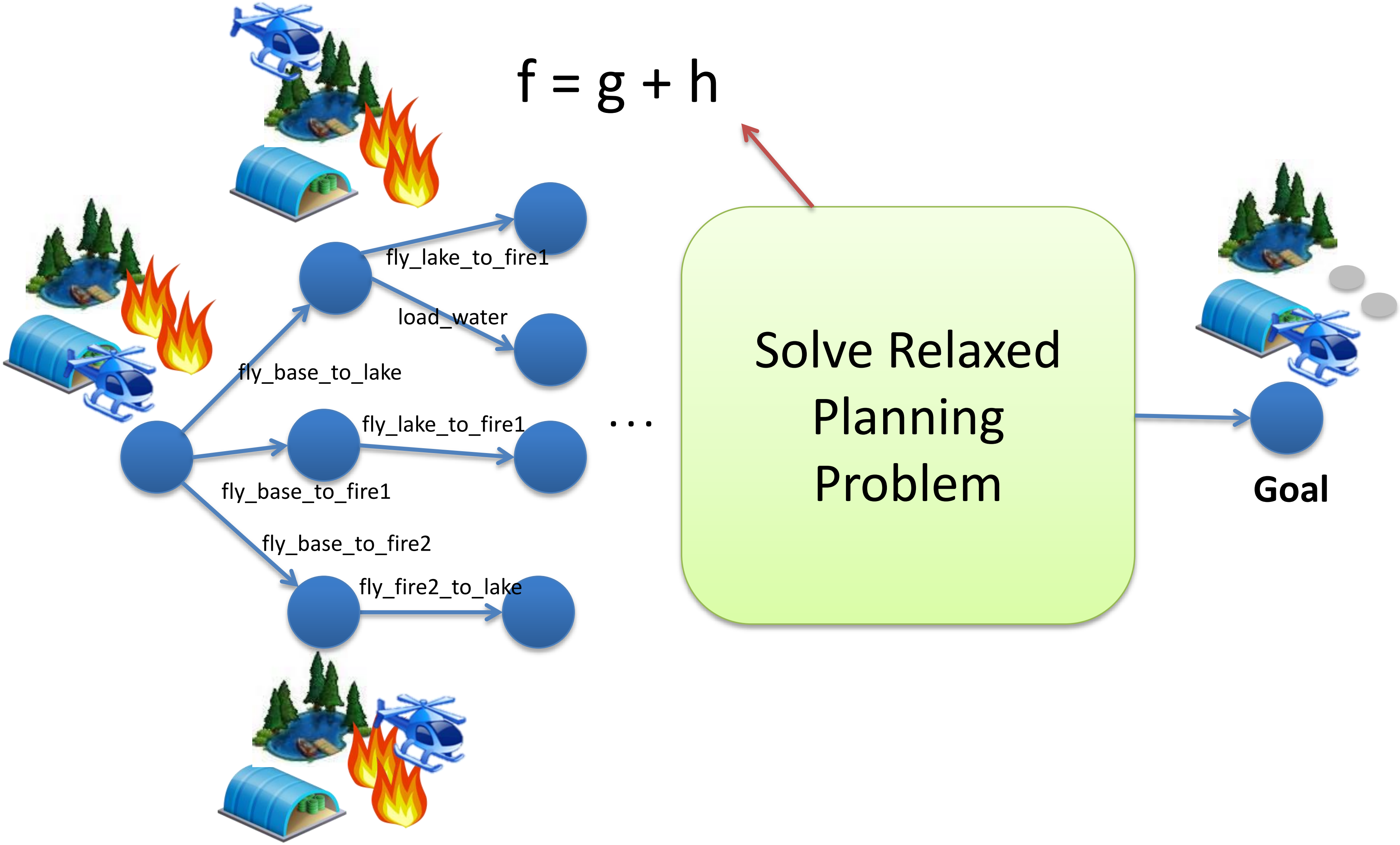
```
class UAV {
  Roadmap location;
  Boolean flying;
  Boolean loaded;

  primitive method takeoff()
    flying == no => flying == yes;

  primitive method land()
    flying == yes => flying == no;

  primitive method load_water(Lake lakespot)
    ((flying == yes) && (loaded == no) && (lakespot.location == location)) => loaded == yes;

  primitive method drop_water_high_altitude(Fire firespot)
    ((flying == yes) && (loaded == yes) && (firespot.location == location) && (firespot == high))
     => ((loaded == no) && (firespot == medium));

  primitive method drop_water_low_altitude(Fire firespot)
    ((flying == yes) && (loaded == yes) && (firespot.location == location) && (firespot == medium))
     => ((loaded == no) && (firespot == out));

  #MOTION_PRIMITIVES(location, fly, flying==yes)
}
```

```
class Lake {
   Roadmap location;
}

class Fire{
   initial value high;
   value medium;
   value out;
   Roadmap location;
}
```

# Simulation Testing: Firefighting Scenario



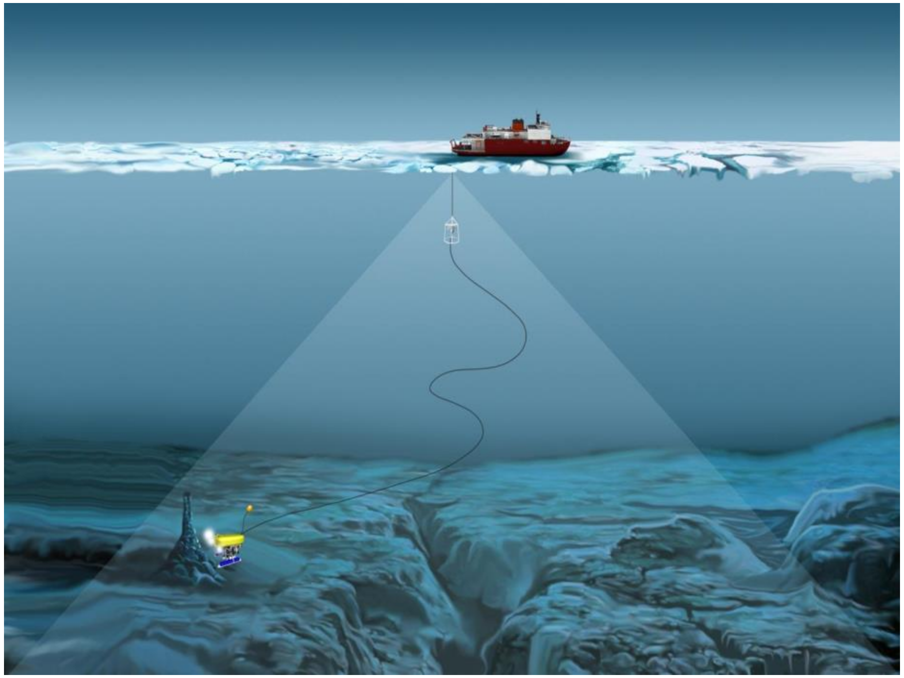
# Decision-making algorithm: Activity Planning



# Outline

- Course in a Nutshell
- Logistics
- Programming Cognitive Robots
- **Self-Adaptive and Self-Repairing Systems**
- Programs that Monitor State

# “Autonomous” vehicles explore far away places .. but often end in disaster!





# What you should *really* learn from immune systems about adversarial design

*Jacob Beal*

NDIST
December, 2015

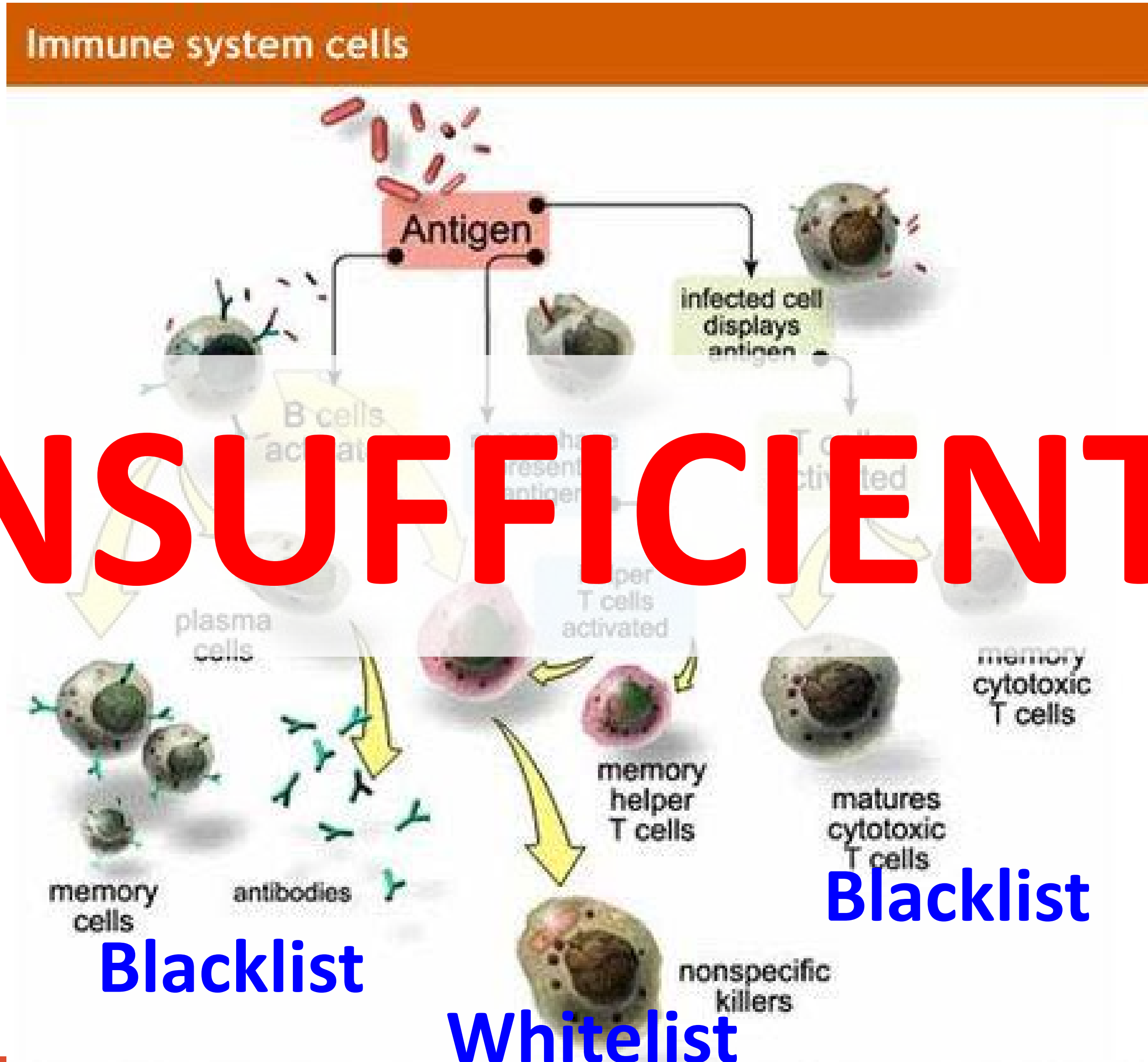


Raytheon
BBN Technologies

*Not for release: images shamelessly ganked from web*

# The Immune System

*(as noticed by computer scientists)*




Complements of Jake Beal

**INSUFFICIENT**

**Blacklist**

**Blacklist**

**Whitelist**

(Source: the Human Immune Response System www.uta.edu/chagas/images/immunSys.jpg)

# But there's more to the immune system…

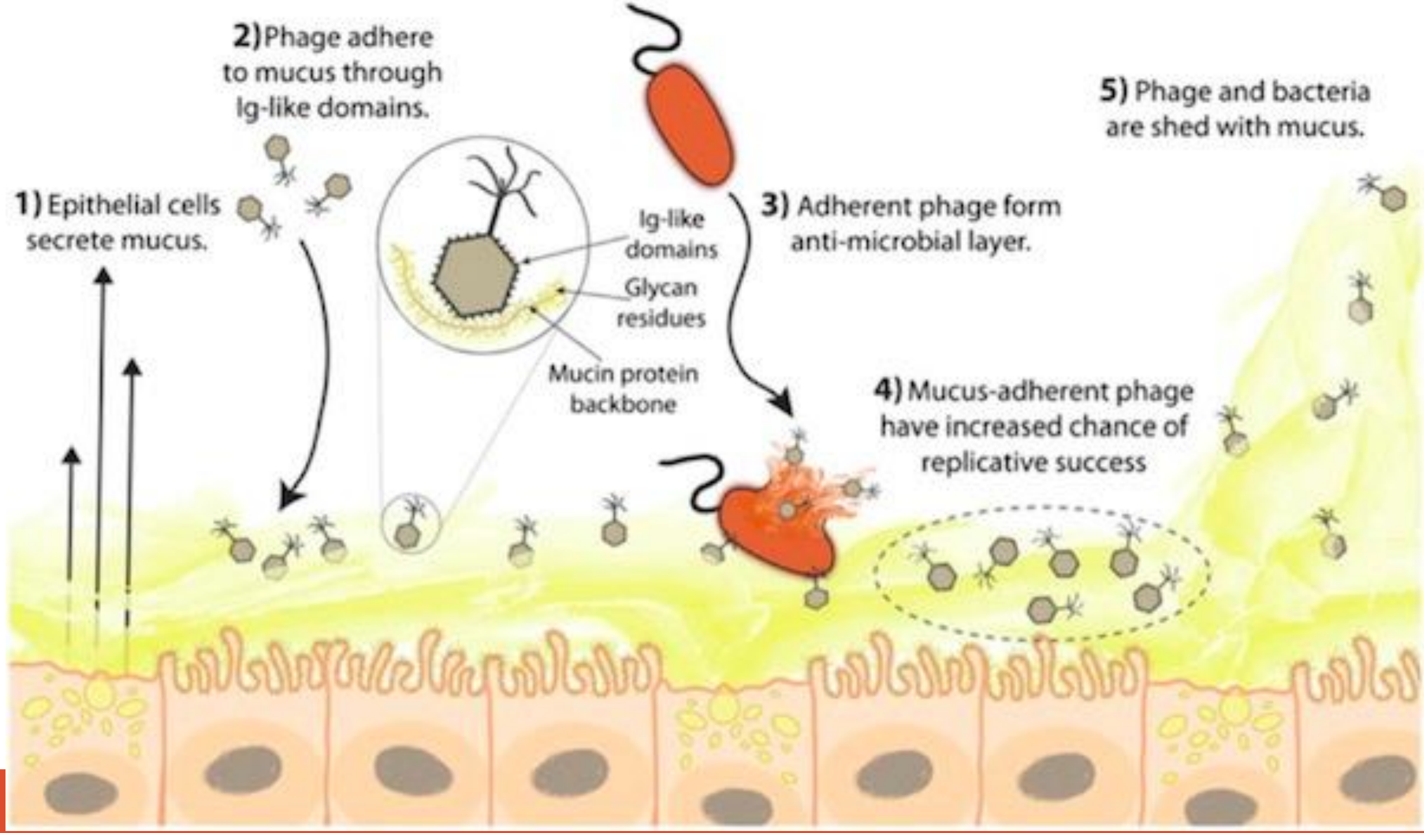
Complements of Jake Beal

- Physical barriers
- Inhospitable environments
- Tolerance
- Death

# Physical Barriers

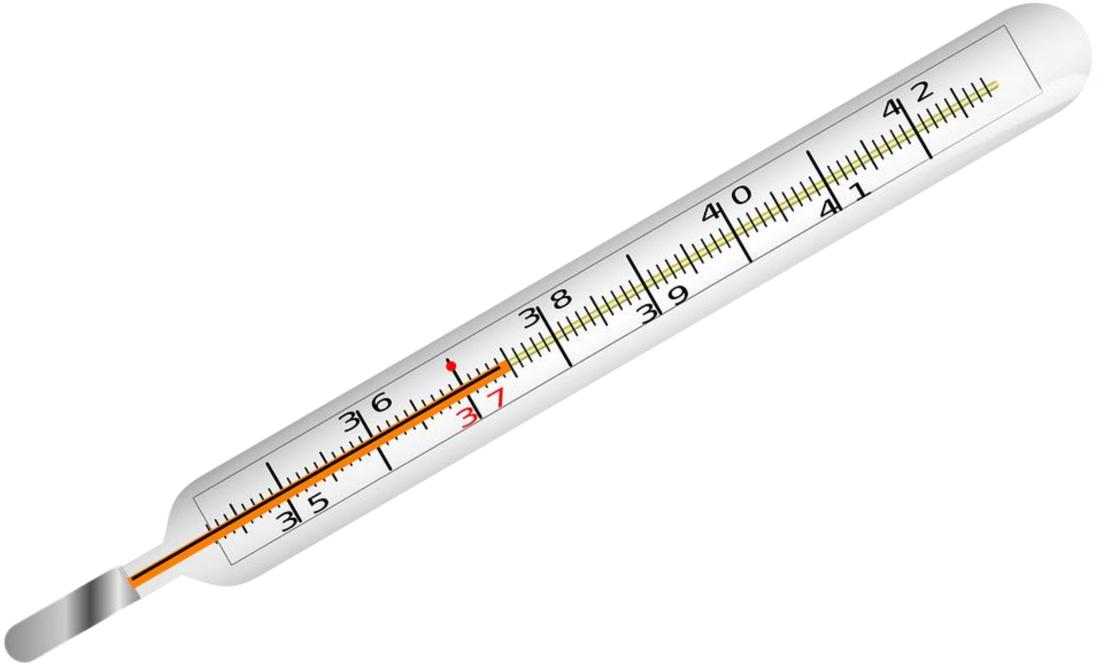
Complements of Jake Beal

- Skin, gastrointestinal wall, blood/brain barrier
- Saliva, tears, mucus (nose, lungs, gut, . . .)
- Defense by discarding



# Inhospitable environments

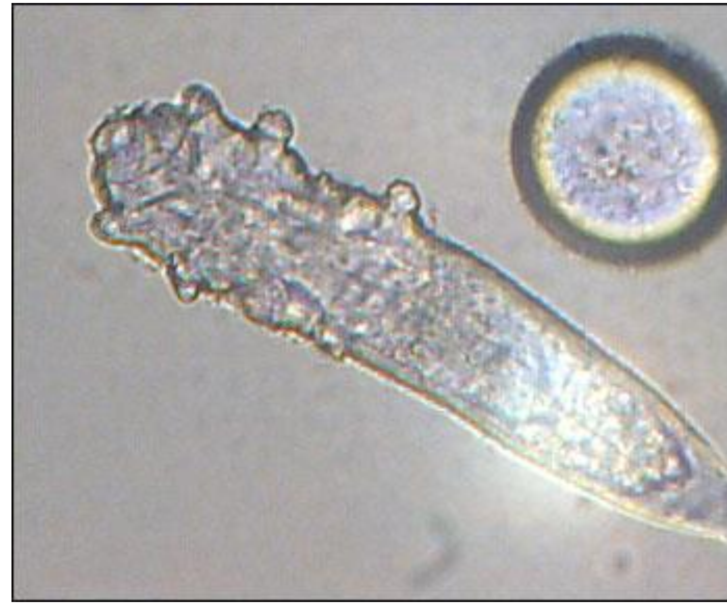
- Fever: high temperature inhibits bacterial growth
- Enzymes in saliva, tears, nasal secretions, perspiration, milk, sperm
- . . .
- Skin is acidic



# Tolerance

Complements of Jake Beal

- Follicular mites
- Parasitic worms
- Dead viruses in the genome
- Microbiome
  - Gut commensals
  - Flora for various cavities



*Human follicular mite*

*>3000 cryptic drug-like molecule gene clusters!*

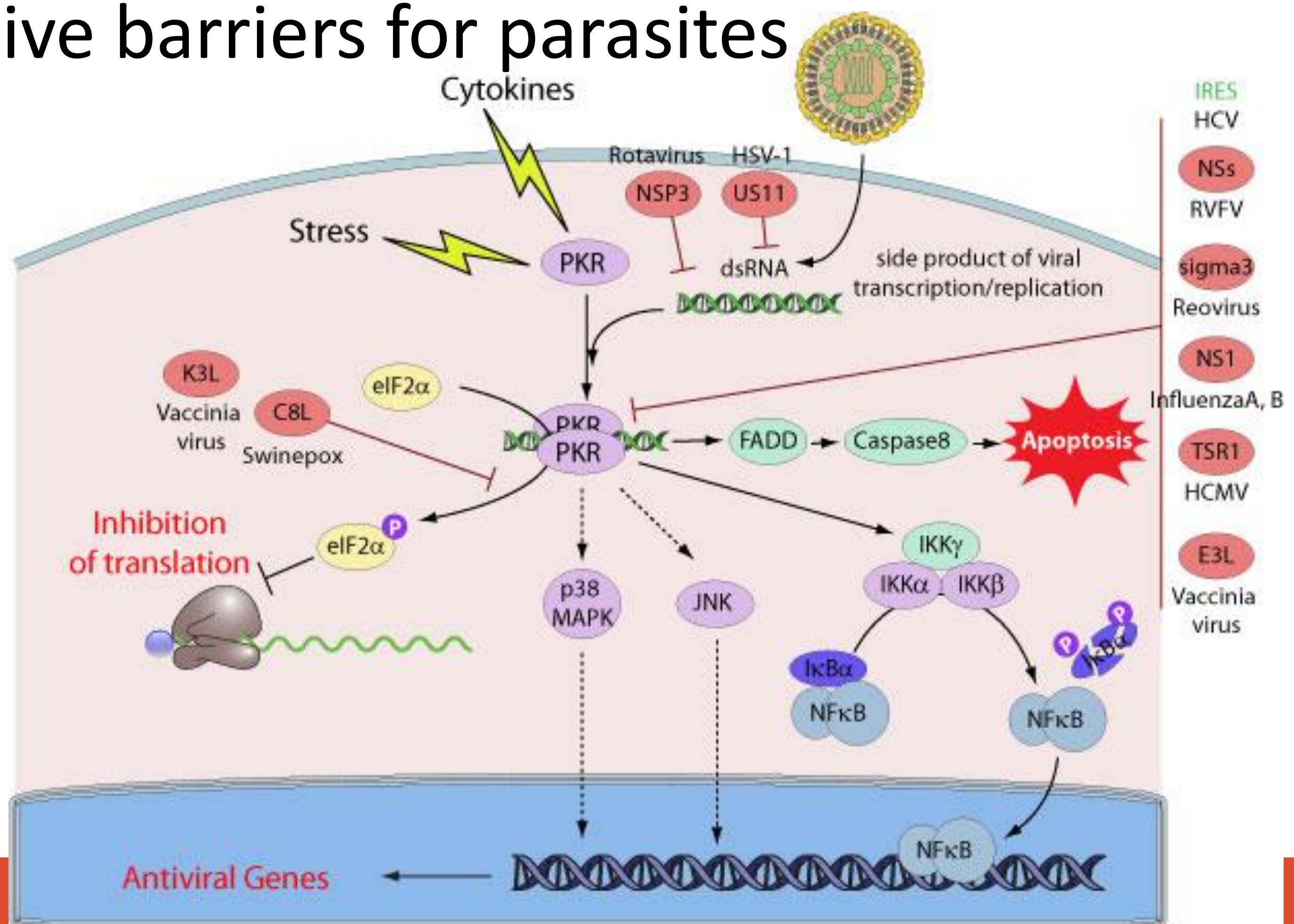
*(Fischbach, Science, 2015)*

**Natural flora occupy niches and consume resources**

# Death

Complements of Jake Beal

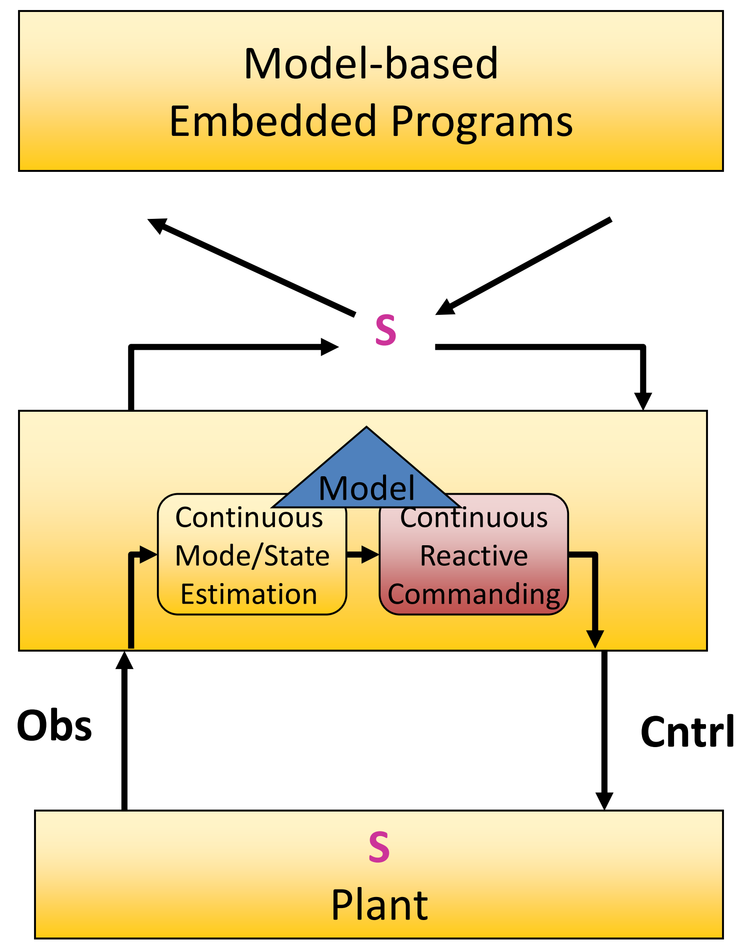
- Cell self-inhibition, suicide
- Reproductive barriers for parasites



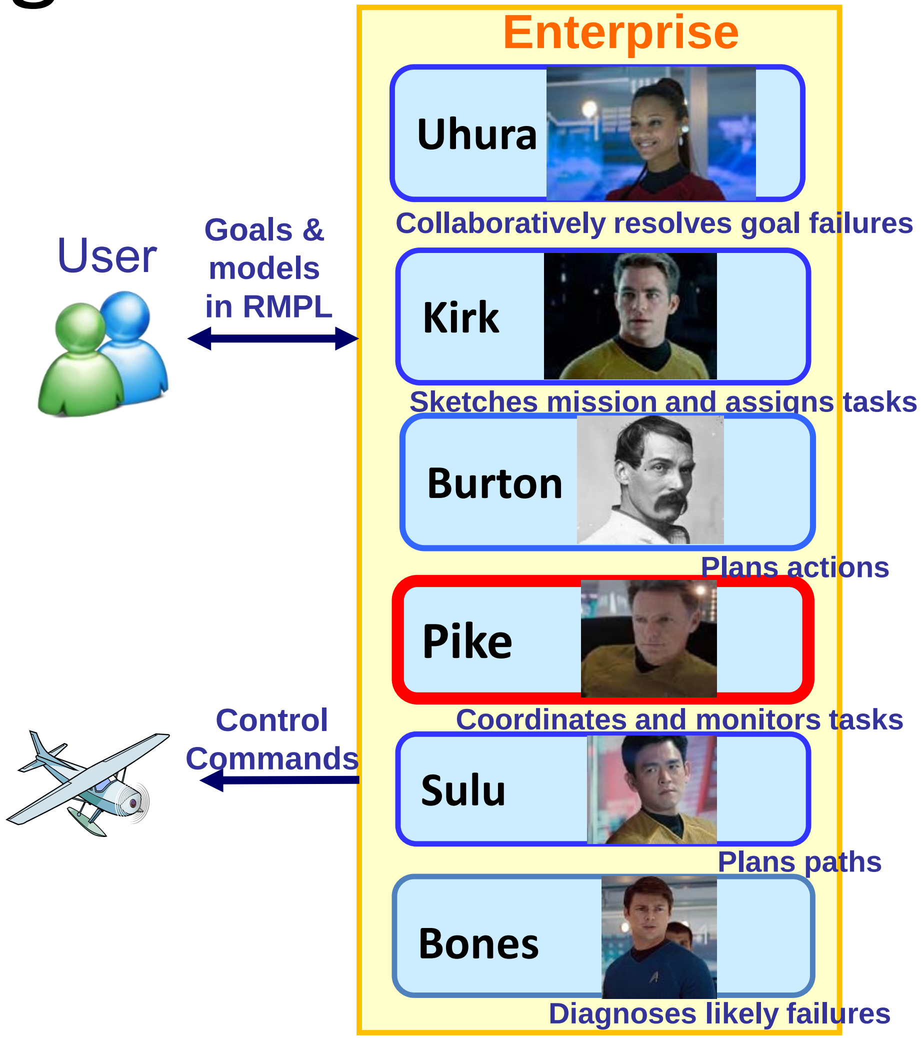
# Model-based Programs Offer Layers of Defense

Languages that achieve robustness through decision layers that are:

- Suspicious
  - Monitor states and goals.
- Adapt to disturbance
  - Adjust timing
  - Select contingencies
- State Aware
  - Plan to achieve goals states.
- Precise
  - Achieve continuous goal states.
- Collaborate
  - Executes programs, revise goals and plan with humans.
- Manage Risk
  - Executes programs in uncertain environments with bounded risk.



# A single "cognitive system" language and executive.



# Outline

- Course in a Nutshell
- Logistics
- Programming Cognitive Robots
- Self-Adaptive and Self-Repairing Systems
- **Programs that Monitor State**

# When things go wrong…

# Execution monitoring: detecting problems

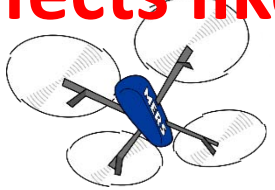
- Something unexpected things happen
  - Not modeled in plan!
  - How to react?
- **Execution monitoring**: *detecting* when things go wrong
- **Replanning**: fixing the problem (later in this course)

# Volcano erupti

```
method run() {
    sequence {
        uav.launch();
        uav.fly_to_base_station();
        uav.pick_up_med_kit();
        uav.fly_to_hikers();
        uav.drop_off_med_kit();
    }
}
```

Base Station

**Actions have preconditions & effects like before**

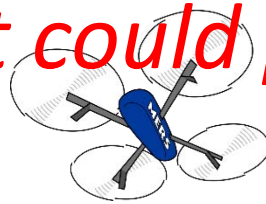


**Program** sequence of **actions** in **RMPL**

# Volcano erupti

Base Station

*What could possibly go wrong??*



Launch Q

Q fly to Base Station

Q pick up med kit

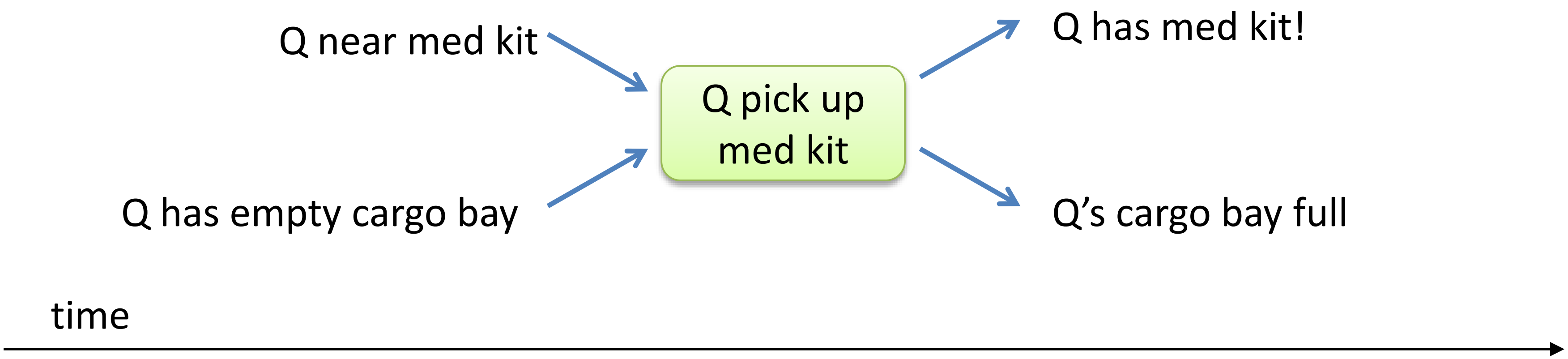
Q fly to hikers

Q give med kit to climbers

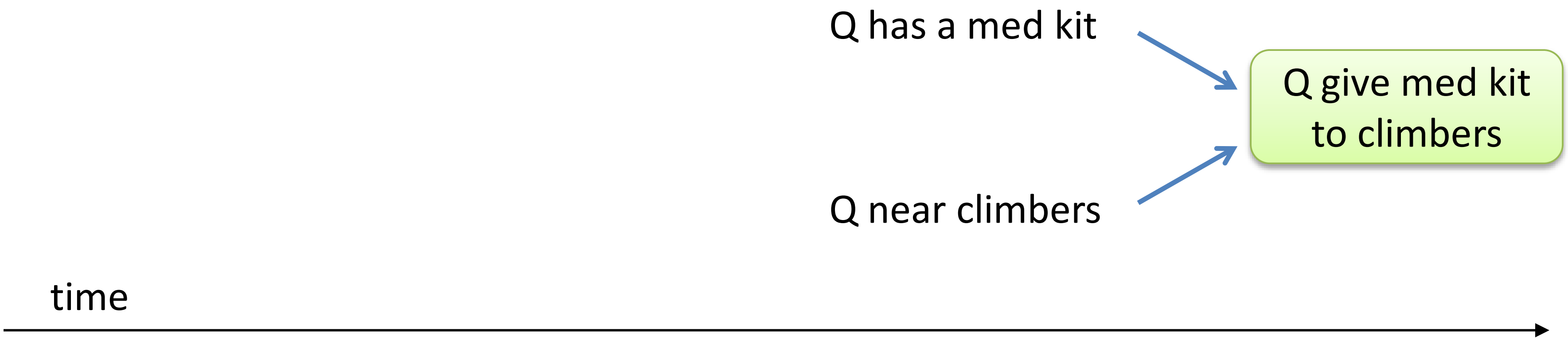
# Preconditions & effects of actions

# Preconditions & effects of actions

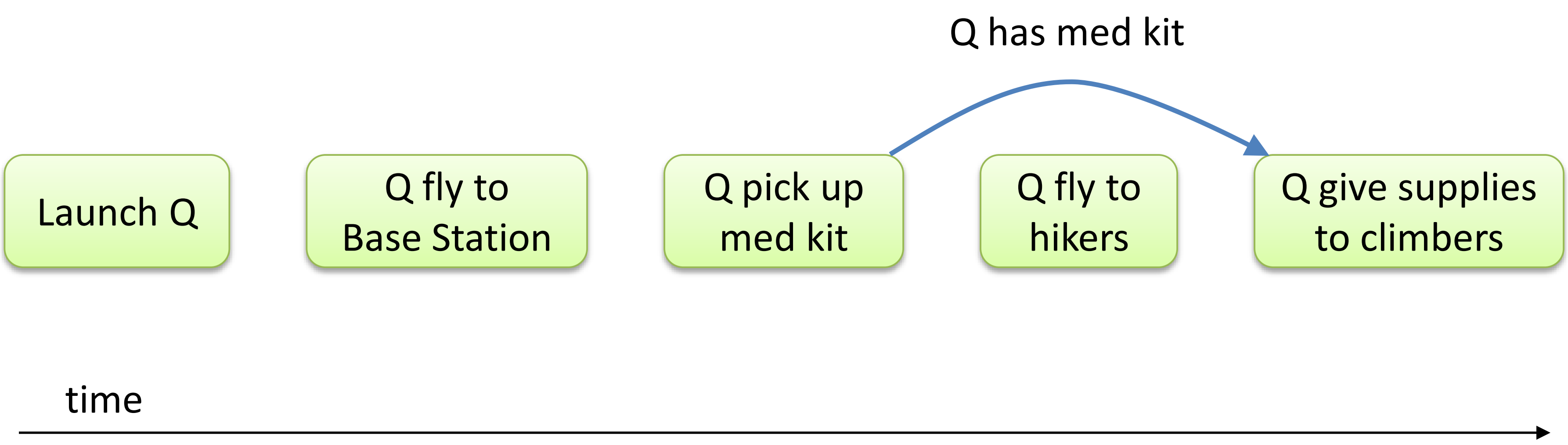
# Preconditions & effects of actions



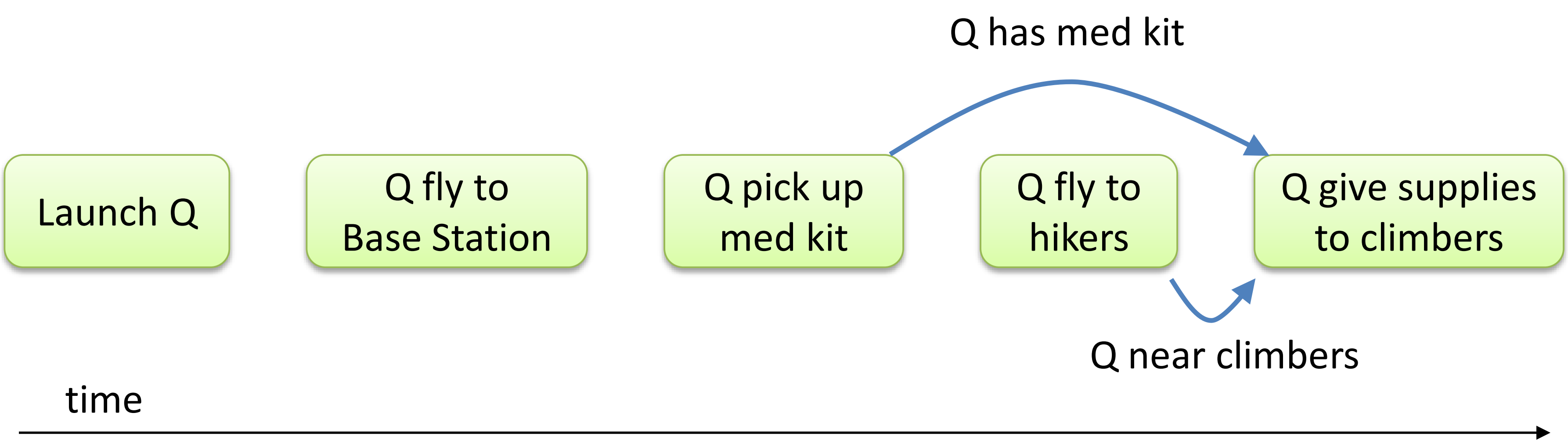
# Preconditions & effects of actions



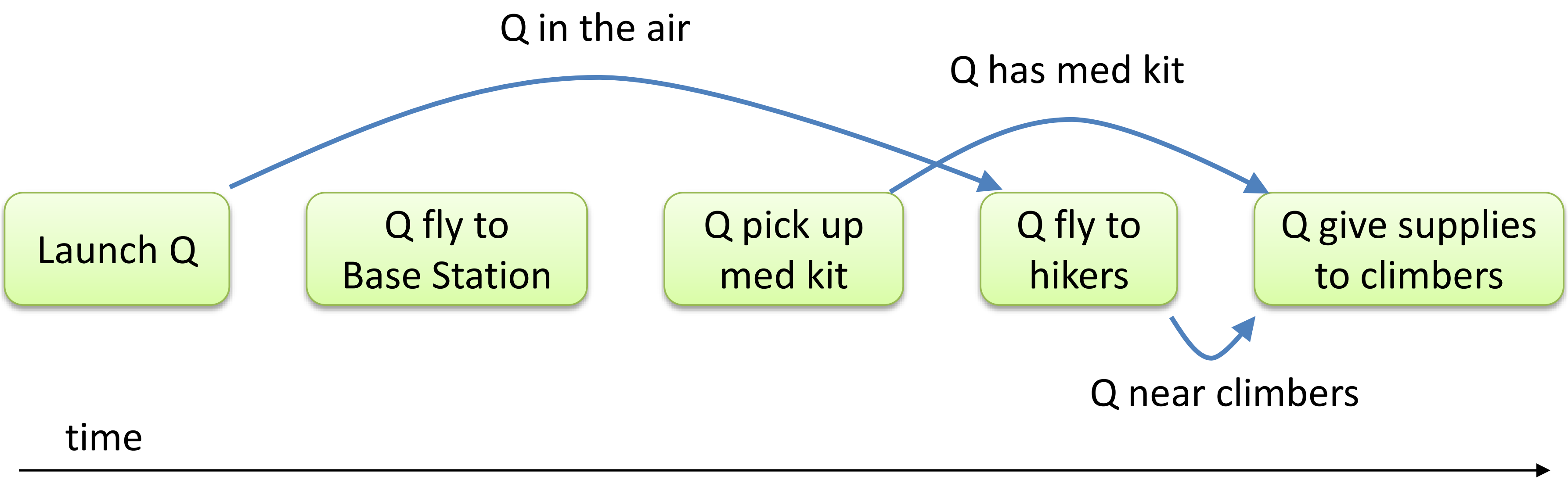
# Preconditions & effects of actions



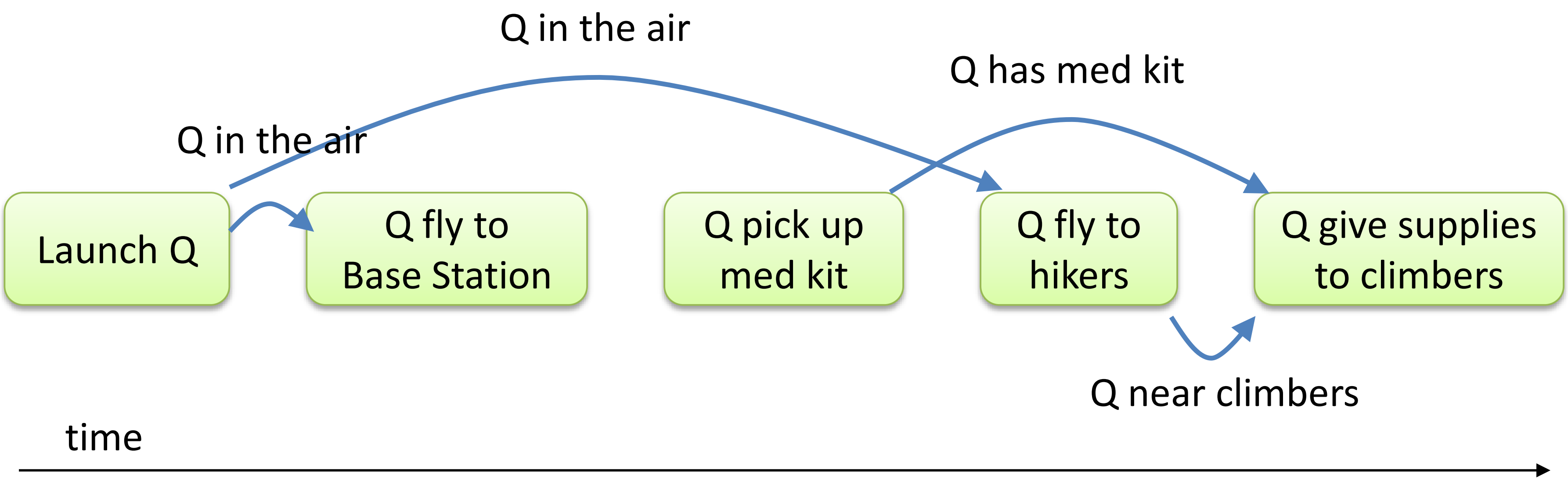
# Where do preconditions come from?



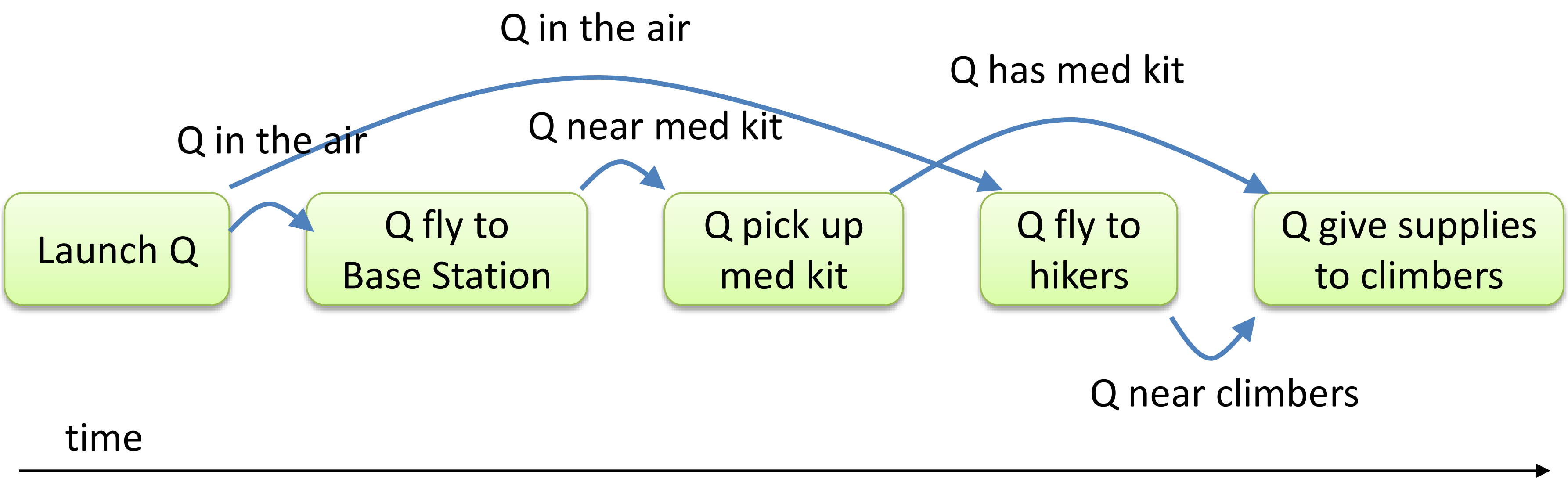
# Where do preconditions come from?



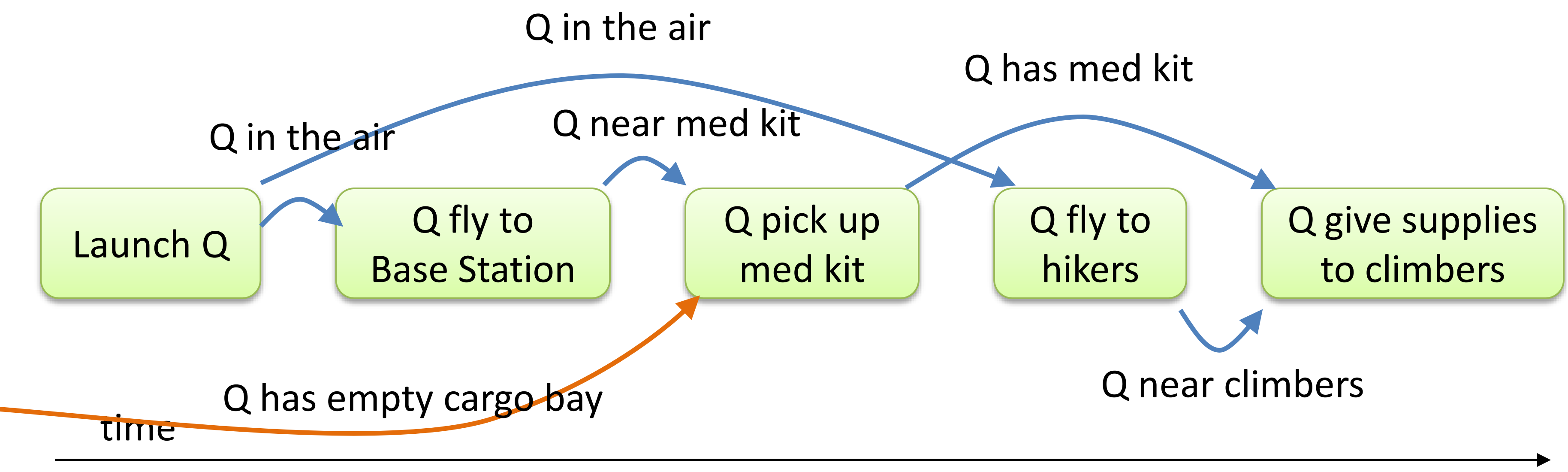
# Where do preconditions come from?



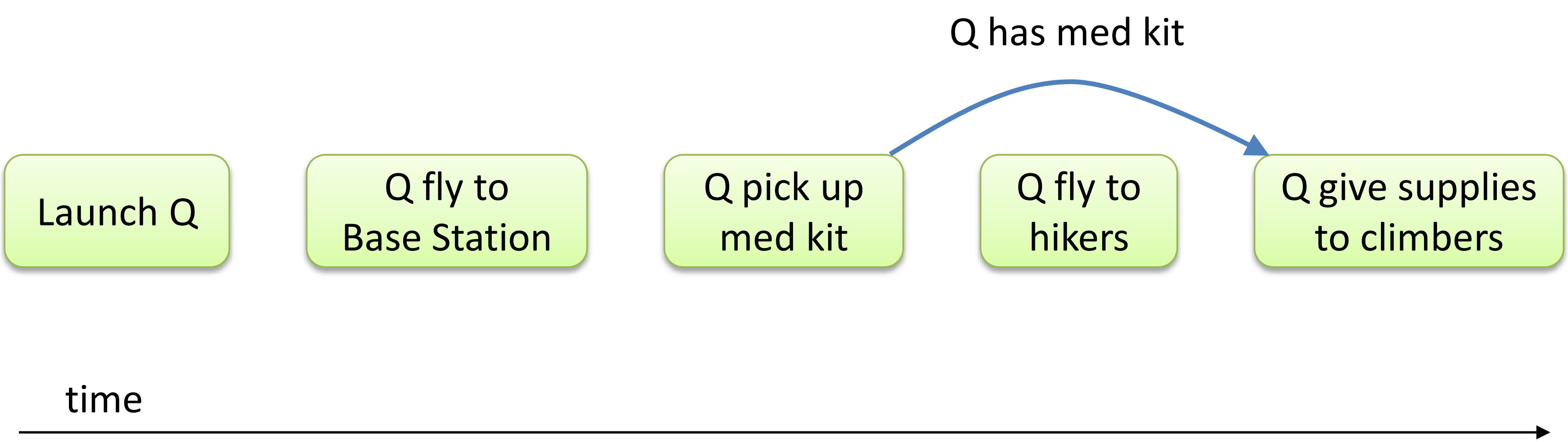
# Where do preconditions come from?



# Where do preconditions come from?

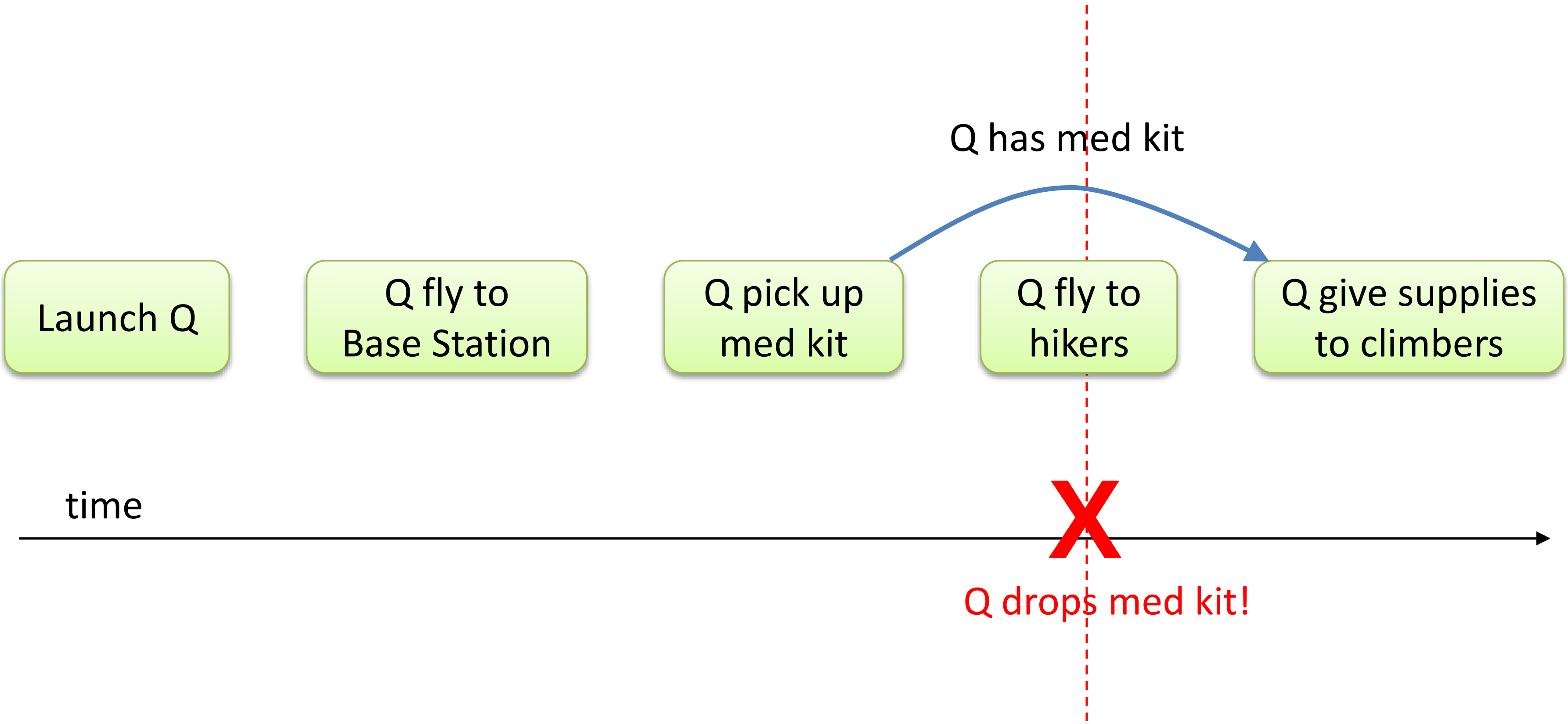


# These are called “causal links”

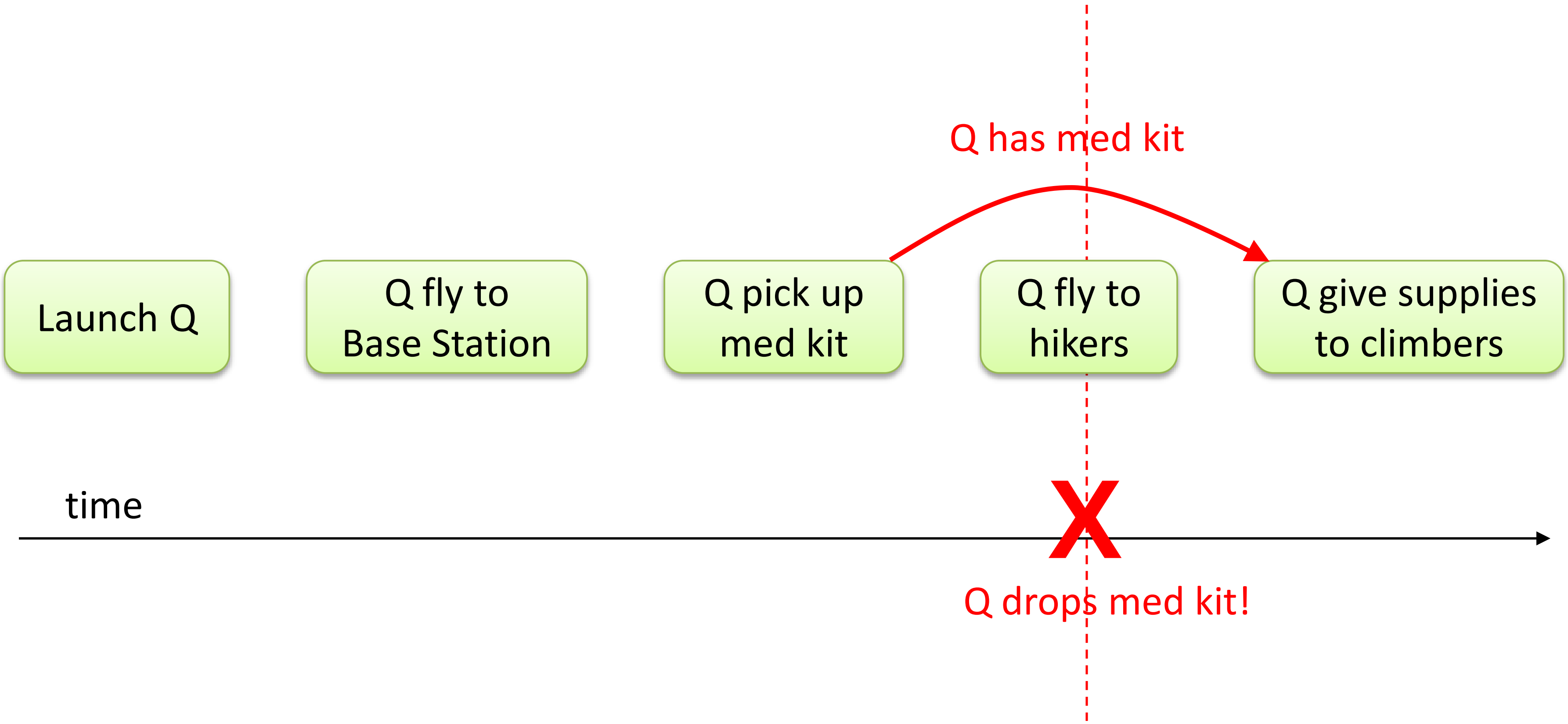



**Causal link**: one action produces something needed by a later action

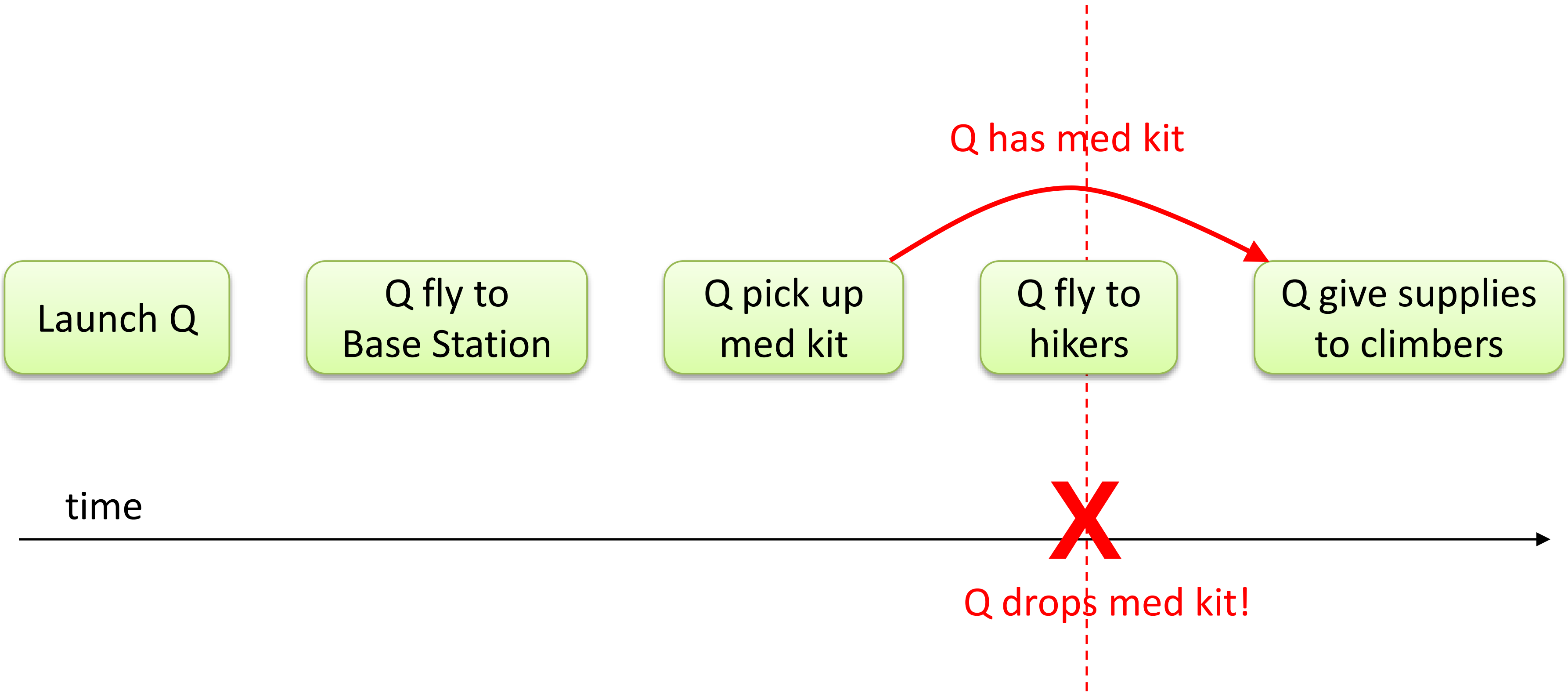
# What happens if something goes wrong?



# What happens if something goes wrong?



# Causal links allow execution monitoring




If condition is violated during causal link, signal failure.

# Causal links for execution monitoring

- Causal links tell you what needs to hold, and when
- Tells you what's *relevant* to the plan
- Can be used offline, for error checking

# How do we do execution monitoring?

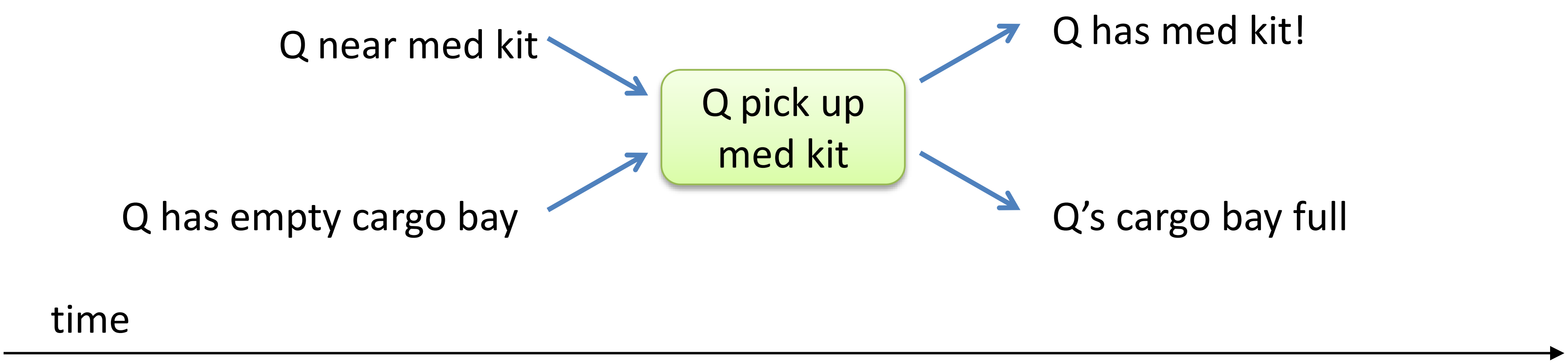
**We need:**

1. Sensing.
2. Action model.
3. Plan.

# 1. Sensing


"Q has medical kit"

# 2. Action model



# 3. Plan


Launch Q
Q fly to Base Station
Q pick up med kit
Q fly to hikers
Q give med kit to climbers
time

# How do we do execution monitoring?

**Offline:**

- Use **plan** & **action** model to extract causal links

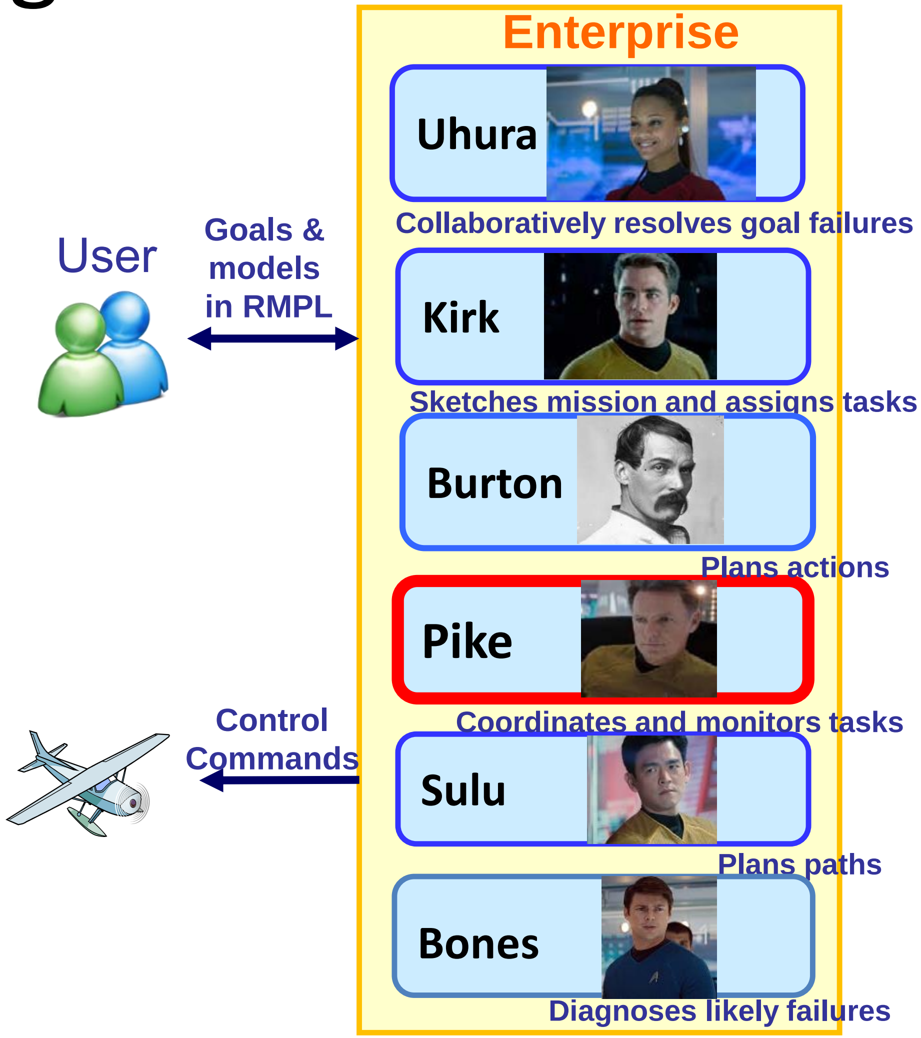
**Online**:

- Continuously **sense** monitor those causal links

# Key takeaways

- Need to monitor the plan when executing
- Causal links:
  - What needs to be true, and when
  - Offline (checking) and online (monitoring)

# A single “cognitive system” language and executive.



# Causal link extraction algorithm

**Extraction (non-temporal, totally-ordered plan - offline):**

```
for each action in plan:
    for each precondition p of action:
        a_p = latest action in plan with effect p
        Add causal link: a_p to action over p
```

**Monitoring (online):**

```
while True:
    cls = currently active causal links (based on actions)
    state = measure state with sensors
    for each causal link cl in cls:
        p = predicate associated with cl
        if not(p in state):
            Trigger execution monitoring exception!
```

16.412J / 6.834J Cognitive Robotics
Spring 2016