Model-Based Programming in PAMELA

Dynamic Object Language Labs Inc.

paulr@dollabs.com, tmarble@dollabs.com
Paul Robertson and Tom Marble

15th April 2016

Outline

The Team

- Dynamic Object Language Labs:
  - Dr Paul Robertson
  - Dr Andreas Hofmann
  - Prakash Manghwani
  - Dan Cerys
- Consultants:
  - Tom Marble
- Open Source Collaborators:
  - TBD (your name could go here!)

What is PAMELA

- An open source modeling language and toolset.
- Support for machine learning algorithms (probabilistic variables).
- Philosophy: A modeling language to make integration of complex algorithms intuitive.
- Cyber Physical Systems
- Learning Algorithms
  - Hidden Markov Models (HMM)
  - Markov Decision Processes (MDP)
  - Partially Observable Markov Decision Processes (POMDP)
  - Bayesian Networks
  - Deep Learning
  - Linear and Nonlinear constraint solvers
  - Constraint solvers in general especially including uncertainty.
- A replacement for PDDL.

A brief History and Acknowledgement

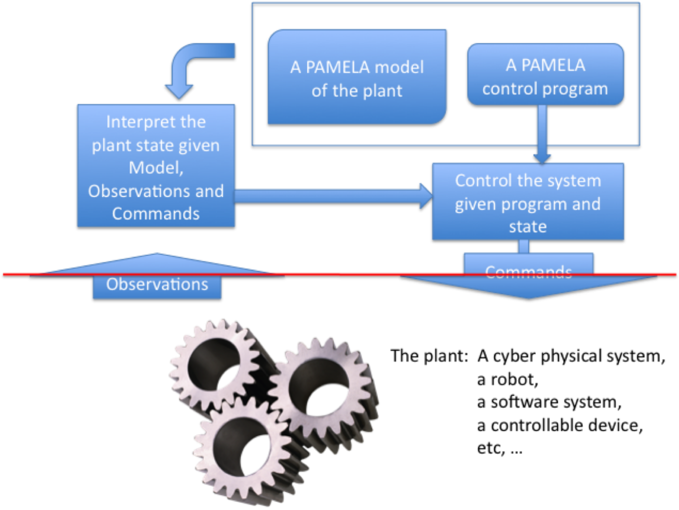
- Language antecedants: RMPL, MPL, Esterel, ESL, and PDDL.
- Lead innovators: Xerox Parc, NASA, MIT (Williams)
- Deployments: Deep Space 1 in 2001.
- Funding: DARPA:

This material is based upon work supported by the Army Contracting and DARPA under contract No. W911NF-15-C-0005. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Army Contracting Command and DARPA.
This work was supported by Contract FA8650-11-C-7191 with the US Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

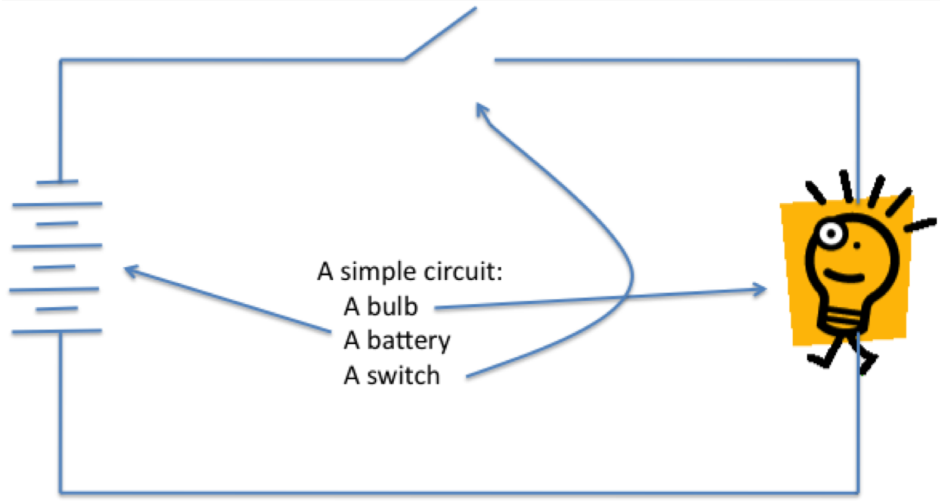
Tools and Architecture

- Model management: Database of models, search, reuse, ...
- Front-End: Manipulation of source, produce datastructures representing models.
- Back-End: Solvers, Planners, Learners, Monitors, runtime systems.
- Multimodal: Visualization tools.
- Architecture: Protocols for communication and Integration.

A model Based Program

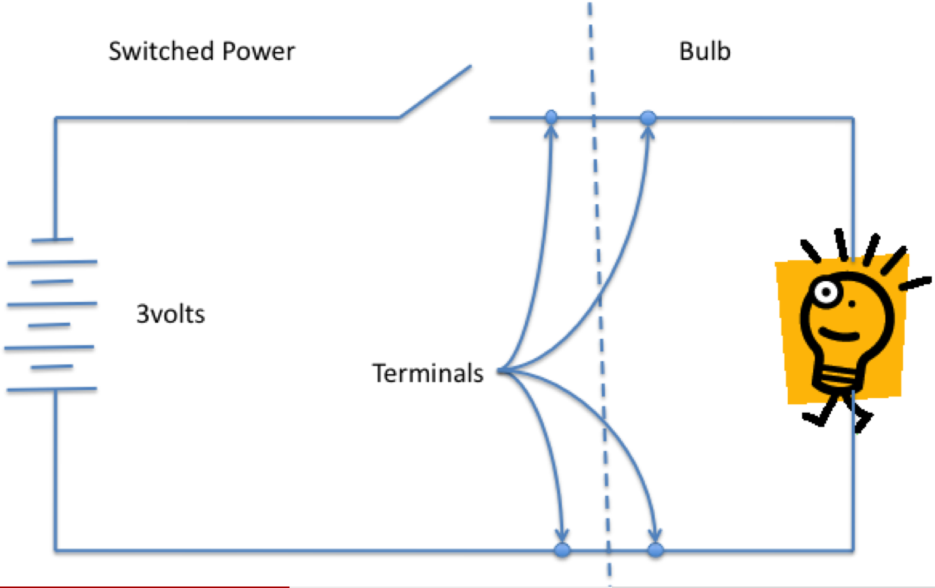

A PAMELA model of the plant
A PAMELA control program
Interpret the plant state given Model, Observations and Commands
Control the system given program and state
Observations
Commands
The plant: A cyber physical system,
a robot,
a software system,
a controllable device,
etc, ...

Simple Plant Example



Imagine a simple circuit consisting of a battery, a lamp, and a switch.

Divide the plant into individual connected components



Modeling Plants/Resources

```
;;; Define power values
(defpclass pwrvals []
  :meta {:version "0.2.0"
         :doc "Enum for power values"}
  :modes [:high :none])
;;; Define a switch
(defpclass psw [gnd pwr]
  :meta {:version "0.2.0"
         :depends [[pwrvals "0.2.0"]]
         :doc "Power Switch"}
  ...)
```

Here we define power values having values either :non or :high. We subsequently define a class for a switch that wcan be either on or off. The switch has two connections to other objects represented by its two paramerers. We will see later how we wire up objects at instantiation time.

Fields

```
(defpclass psw [gnd pwr]
  :meta {:version "0.2.0"
         :depends [[pwrvals "0.2.0"]]
         :doc "Power Switch"}
  ...)

  :fields {:TP1 gnd
           :TP2 pwr
           :pwr (pwrvals :initial :none) }
```

Here we add some fields to represent state of the object. :TP1 represents the connection to terminal TP1, similarly for :TP2, pwr defines the power state of the switch which is constrained to have values defined by pwrvals.

Modes

```
(defpclass psw [gnd pwr]
  :meta {:version "0.2.0"
         :depends [[pwrvals "0.2.0"]]
         :doc "Power Switch"}
  :fields {:TP1 gnd
           :TP2 pwr
           :pwr (pwrvals :initial :none) }

  :modes {:on (condition (field= :pwr :high))
          :off (condition (field= :pwr :none))
          :fail true }
  ...)
```

Our switch has two nominal modes, :on and :off. We also add a fail mode by tradition which is unconditional.

Transitions

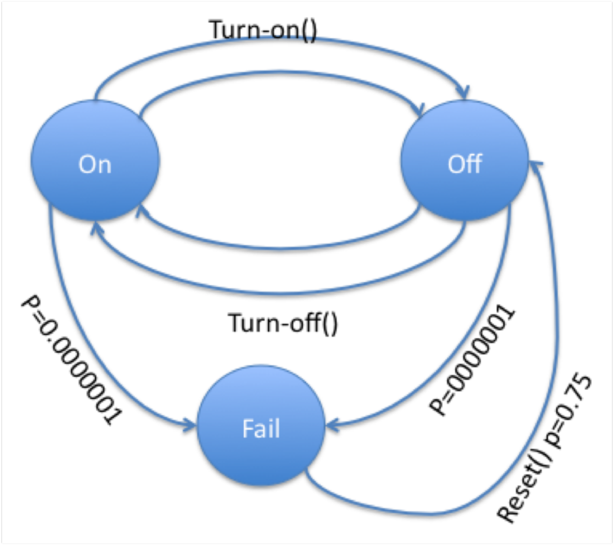
```
(defpclass psw [gnd pwr]
  ...
  :fields {:TP1 gnd
           :TP2 pwr
           :pwr (pwrvals :initial :none) }
  :modes {:on (condition (field= :pwr :high))
          :off (condition (field= :pwr :none))
          :fail true }
  :transitions {:off:on {:pre :off :post :on }
                :on:off {:pre :on :post :off }
                :*:fail {:post :fail :probability 0.0000001}}
...)
```

The switch can turn on, off, or fail. We can specify names transitions including how pre and post conditions including probabilities for the transition to occur.

Methods (Primitive)

```
(defpclass psw [gnd pwr]
  ...
  :modes {:on (condition (field= :pwr :high))
          :off (condition (field= :pwr :none))
          :fail true }
  :transitions {:off:on {:pre :off :post :on
                         :doc "turning on"}
                :on:off {:pre :on :post :off
                         :doc "turning off"}
                :*:fail {:post :fail :probability 0.0000001
                         :doc "spontaneous switch failure"}}
  :methods [(defpmethod turn-on
              {:pre :off :post :on :bounds [1 3]
               :doc "turns on the power supply"}
              []))
```

State Diagram of the Switched Power



```
(defpclass lightvals []
  :meta {:version "0.2.0"}
  :modes [:bright :dark])
(defpclass bulb [vcc vdd]
  :meta {:version "0.2.0"
         :depends [[lightvals "0.2.0"]]
         :doc "A light bulb"
         :icon "bulb.svg"}
  :fields {:anode vcc
           :cathode vdd
           :illumination (lightvals :initial :dark :access :pu
           :sensed-illumination (lightvals :observable true
                                           :access :public :initial :da
} ...)
```

The light bulb will not be commandable but will turn on and off depending upon its power inputs. We define modes and transitions for those in the next slide.

```
  :modes {:on (condition (and
                 (field= :illumination :bright)
                 (field= :sensed-illumination :bright)))
        :off (condition (and
                  (field= :illumination :dark)
                  (field= :sensed-illumination :dark)))
        :fail true }
  :transitions {:off:on {:pre (condition (and
                                  (mode= :off)
                                  (field= :anode :none)
                                  (field= :sensed-illumination
:dark)))
                          :post :on
                          :bounds [1 3]} ...
                 :*:fail {:probability (lvar "pfbulb")}})
```

Wiring up the components

```
(defpclass circuit1 []
  :meta {:version "0.2.0"
         :depends [[psw "0.2.0"]
                   [bulb "0.2.0"]]
         :doc "An example circuit: power switch and light bul
  :fields {:source (lvar "source")
           :drain (lvar "drain")
           :bulb1 (pclass bulb :source :drain)
           :switchedpower (pclass psw :source :drain)})
```

We create two names logic variables :drain and :source these are passed in to the instances of the pclass and switchedpower classes that are instantiated using 'pclass'. The instances are thus wired together. The runtime system will provide bindings for the lvars in response to interactions with the plant.

Control Programs

This is a detailed and complicated topic. I propose to do a second session to cover the details that include:

- State constraints
- Temporal constraints
- Maintaining
- Conditionals

A subset of these capabilities can result in a TPN which can be solved and dispatched efficiently. More interesting hybrids are possible however which combine TPN like mission modeling with reactive components.

Simple TPN example

```
(defpclass quadcopter1 [plant]
  :methods [(defpmethod verify-tires [A B C D]
              (sequential
               (plant$gotowaypoint A)
               (plant$take-high-res-images)
               (plant$process-and-move-on B)
               (plant$process-and-move-on C)
               (plant$process-and-move-on D)
               (plant$process)))])
```

CONTINUED ON NEXT SLIDE

```
(defpclass quadcopter1 [plant]
   ... ON PREVIOUS SLIDE
       (defpmethod process []
              (sequential
               (plant$extract-evidence-from-image)
               (parallel
                (choose
                  (choice :cost 3.0 :reward 10
                    (plant$interpret-1))
                  (choice :cost 2.0 :reward 4
                    (plant$interpret-2))
                  (choice :cost 2.0 :reward 4
                    (plant$interpret-3))))))
       (defpmethod process-and-move-on [X]
              (parallel
               (plant$gotowaypoint X)
               (plant$process)))])
```