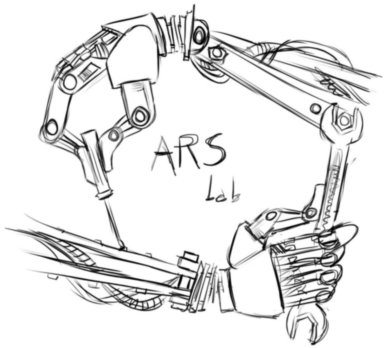
Programming Autonomous Robots using PROFETA and the BDI model

Corrado Santoro
http://www.dmi.unict.it/~santoro

ARSLAB - Autonomous and Robotic Systems Laboratory
Dipartimento di Matematica e Informatica
Università di Catania, Italy
santoro@dmi.unict.it



PROFETA Basics

PROFETA Basics

- PROFETA *(Python RObotic Framework for dEsigning sTrAtegies)* is Python tool for programming autonomous systems (agents or *robots*) using a **declarative approach**.
- The aim is to have a single environment to implement the software of an autonomous robot.
- It provides an "all-in-one" environment supporting both **imperative** (algorithmical part) and **declarative** (behavioural part) programming models.
- It can be downloaded from:
  `http://github.com/corradosantoro/profeta`

PROFETA Basics

- PROFETA is inspired by AgentSpeak(L), a kernel agent language based on the **belief-desire-intention (BDI)** paradigm
- Its syntax is logic/declarative with some similarities with Prolog
- Concepts:
  - **Beliefs**
  - **Goals**
  - **Actions**
  - **Plans**

PROFETA Beliefs

- **Beliefs** are used to represent **agent's knowledge** (agent status, data, environment status, etc.)
- Beliefs can be **added** to or **removed** from a **Knowledge Base (KB)** (provided by PROFETA)
- The KB can be **queried** to check the presence of certain beliefs and behave accordingly
- Beliefs are syntactically expressed by atomic forumlae with ground terms:
  - `position(150,60)`
  - `block("red")`
  - `stack("cube","pyramid")`

PROFETA Goals

- **Goals** are used to represent the **motivational state** of the agent, i.e. what the agent wants to do
- Actions needed to achieve a certain goal are specified in a **goal plan** of the behaviour (see below)
- A goal can be achieved by "calling it", in a way similar to a procedure call
- Goals are syntactically expressed by atomic forumlae with ground terms or free variables:
  - `pick_block("red")`
  - `pick_block("X")`

PROFETA Actions

- **Actions** are used to represent the **execution units**, driving an agent to achieve their goals
- Actions contain the specific code (written in Python) to let the robot to concretely perform "that thing" on the environment
- Actions may **fail** and may be **asynchronous**
- Actions to execute are specified in the behaviour **plans** (see below)
- Actions are syntactically expressed by atomic forumlae with ground terms or bound variables:
  - `go_to(1500, 120)`
  - `grip_open()`

PROFETA Plans

- A PROFETA behaviour is a set of **reactive plans** in the form **Event-Condition-Actions**:
  - The *Event* is predicate that triggers the plan:

belief assert	`+bel(...)`
belief retract	`-bel(...)`
goal achievement	`goal(...)`
goal failure	`-goal(...)`

  - *Condition* is a *guard* which needs to be true in order to trigger the plan
  - It is predicate on one or more beliefs present in the knowledge base
  - *Actions* is a list of "things to do" when the plan is triggered:

belief assert	`+bel(...)`
belief retract	`-bel(...)`
goal achievement	`goal(...)`
goal failure	`-goal(...)`
user-defined action	`go_to("X", "Y")`
python expression	`"X = X + 1"`

"Hello World" in PROFETA

```
# import libraries
from profeta.lib  import *
from profeta.main import *

class say_hello(Goal):
    pass

# instantiate the engine
PROFETA.start()

# define a goal plan
say_hello() >> [ show_line("hello world from PROFETA") ]

# achieve the "say_hello()" goal
PROFETA.achieve(say_hello())

# run the engine
PROFETA.run()
```

Belief Plans in PROFETA

```
# import libraries
from profeta.lib  import *
from profeta.main import *

class number(Belief):
    pass

# instantiate the engine
PROFETA.start()

# define an event plan
+number("X") >> [ show_line("yeah! Now I know the number", "X") ]

# add the following beliefs in the knowledge base
PROFETA.assert_belief(number("1"))
PROFETA.assert_belief(number("2"))
PROFETA.assert_belief(number("5"))

# run the engine
PROFETA.run()
```

PROFETA Syntax

Basic Parts of a PROFETA Program

1. Python imports, **PROFETA lib imports**
2. **Declaration** of application-dependent PROFETA entities:
   - **Our beliefs**
   - **Our goals**
   - **Our actions**
3. PROFETA engine **instantiation**: "`PROFETA.start()`"
4. Specification of **robot behaviour** by means of the PROFETA declartive syntax
5. Assertion of **initial beliefs** (if needed): "`PROFETA.assert_belief()`"
6. Start everything with or without a shell:
   - `PROFETA.run()`
   - `PROFETA.run_shell(globals())`

PROFETA Declarative Syntax

- The syntax of a **profeta plan** is based on the following expressions:

EVENT >> [*ACT1, ACT2, ..., ACTn*]
EVENT / *COND* >> [*ACT1, ACT2, ..., ACTn*]

- **Goals**, **Beliefs** and **Actions** are expressed as **atomic forumlae** with no parameters or Python strings as parameters
- Parameters/strings starting with a **lowercase letter/number/symbol** are treated as **constants** (*atoms*)
- Parameters/strings starting with a **uppercase letter** are treated as **variables** (Prolog-style)
- Underscore "_" is the special **"dont-care"** variable (Prolog-style)

```
pick("redbox") # redbox is a constant (atom)
before("redbox","X") # redbox is a constant (atom)
                     # X is a variable
```

PROFETA Declarative Syntax: Events

An **event** is able to trigger a plan. It can be:

- **Belief Assert**: `+bel(...)`
- The plan is triggered when a belief with the given pattern is asserted:

```
+block("cube") >> [ ... ]
# triggered when block("cube") is asserted
```

```
+block("X") >> [ ... ]
# triggered when a block(...) belief is asserted
# variable X is bound to the parameter
```

- **Bound variables** hold their values in the context of the **same plan**

PROFETA Declarative Syntax: Events

An **event** is able to trigger a plan. It can be:

- **Belief Retract**: `-bel(...)`
- The plan is triggered when a belief with the given pattern is retracted:

```
-block("cube") >> [ ... ]
# triggered when block("cube") is retracted
```

```
-block("X") >> [ ... ]
# triggered when a block(...) belief is retracted
# variable X is bound to the parameter
```

PROFETA Declarative Syntax: Events

An **event** is able to trigger a plan. It can be:

- **Goal Achievement**: `goal(...)`
- The plan is triggered when the achievement of a goal with the given pattern is requested:

```
go(1500,1000) >> [ ... ]
# triggered when go(1500,1000) is ''called''
```

```
go("X", 1000) >> [ ... ]
# triggered when go(.., 1000) is called
# variable X is bound to the first parameter of the call
```

PROFETA Declarative Syntax: Events

An **event** is able to trigger a plan. It can be:

- **Goal Failure**: `-goal(...)`
- The plan is triggered when that goal failed:

```
-go(1500,1000) >> [ ... ]
# triggered when the goal go(1500,1000) failed
```

```
-go("X", 1000) >> [ ... ]
# triggered when the goal  go(.., 1000) failed
# variable X is bound to the first parameter of the call
```

PROFETA Declarative Syntax: Conditions

A **condition** is a guard that expresses a predicate on:

- The presence, in the KB, of certain beliefs (with given patterns)
- The values of the variables bound in the plan

```
go("X","Y") / position("X", "Y") >> [ ... ]

# Triggered when go(..,..) goal is ``called''
#
# Variables X and Y are bound to the goal parameters
#
# The belief position(..,..) must be asserted, and its
# parameters must be equal to variables X and Y
```

PROFETA Declarative Syntax: Conditions

- Several belief patterns can be specified using the "&" operator
- Variables, when present, are unified

```
go("X","Y") / (position("X", "Y") & cube("X", "Y1")) >> [ ... ]

# Triggered when go(..,..) goal is ``called''
#
# Variables X and Y are bound to the goal parameters
#
# The beliefs position(..,..) and cube(..,..)must be
# asserted with the given variable patters
#
```

PROFETA Declarative Syntax: Conditions

- Boolean expressions on variables are possible using "lambdas"

```
go("X","Y") / (position("X", "Y") & (lambda : int(X) > 30) ) >>
  [ ... ]

# Triggered when go(..,..) goal is ''called''
#
# Variables X and Y are bound to the goal parameters
#
# The belief position(..,..) must be asserted, and its
# parameters must be equal to variables X and Y;
# variable X must also be greater than 30
#
```

- (Since variables are not typed, it's better to use typecasting in order to ensure correct evaluation of data)

PROFETA Declarative Syntax: Actions

- **Actions** are the computations to be executed when a plan is triggered
- Actions are syntactically expressed, in a plan, as a Python list
- Elements of the list can be:

belief assert	`+bel(...)`
belief retract	`-bel(...)`
goal achievement	`goal(...)`
goal failure	`-goal(...)`
library action	`show_line("variables are", "X", "Y")`
user-defined action	`go_to("X", "Y")`
python expression	`"X = X + 1"`

- Certain actions (KB manipulation, goal achievement/failure) are able to trigger further plans (if the proper event is present)

Example: Summing “n” numbers

- Python imports, **PROFETA lib imports**

```
# import libraries
from profeta.lib  import *
from profeta.main import *

...
```

Example: Summing “n” numbers

- **Declaration** of application-dependent PROFETA entities:
  - A **belief** `number(X)` which is used to represent a number to sum
  - A **goal** `sum_all()` which contains the actions to perform our sum
- Beliefs and goals are defined by simply subclassing `Belief` and `Goal`
- Then the **PROFETA** engine can be instantiated

```
...
class number(Belief):
    pass

class sum_all(Goal):
    pass

PROFETA.start()
...
```

Example: Summing “n” numbers

- Specification of the **behaviour** by means of a set of **plans**

```
sum_all() / number("X") >> [ -number("X"), sum_all("X") ]
sum_all("S") / number("X") >> [ -number("X"),
                                "S = S + X",
                                sum_all("S") ]
sum_all("S") >> [ show_line("the sum is", "S") ]
```

Example: Summing “n” numbers

1. The goal “sum_all()” is executed given that at least one belief “number(X)” is in the knowledge base; if this is the case, variable “X” is bound to the parameter; as a consequence, the belief is removed from the knowledge base, then the goal “sum_all()” is invoked with the number value as argument:

```
sum_all() / number("X") >> [ -number("X"), sum_all("X") ]
```

Example: Summing “n” numbers

2. The goal “`sum_all(S)`” is executed with the partial sum `S` as parameter and given that at least one belief “`number(Y)`” is in the knowledge base; the belief is removed from the knowledge base, the partial sum is updated and the goal “`sum_all(S)`” is recursively invoked:

```
sum_all("S") / number("Y") >> [ -number("Y"),
                                "S = S + Y",
                                sum_all("S") ]
```

Example: Summing “n” numbers

3. The last plan does not require the presence of the belief; it is executed given that the previous plan is not triggered; it means that no more numbers need to be processed, so the final value can be printed on the screen:

```
sum_all("S")  >> [ show_line("the sum is", "S") ]
```

Example: Summing “n” numbers

4. After the plans, we assert our initial beliefs:

```
PROFETA.assert_belief(number("4"))
PROFETA.assert_belief(number("8"))
PROFETA.assert_belief(number("10"))
PROFETA.assert_belief(number("20"))
PROFETA.assert_belief(number("25"))
PROFETA.assert_belief(number("1"))
PROFETA.assert_belief(number("21"))
PROFETA.assert_belief(number("12"))
```

Example: Summing “n” numbers

5. Finally, we specify the goal to be executed and we run the PROFETA engine:

```
PROFETA.achieve(sum_all())
PROFETA.run()
```

Example: Summing “n” numbers

Here is the complete listing:

```
from profeta.lib  import *
from profeta.main import *

class number(Belief):
    pass

class sum_all(Goal):
    pass

PROFETA.start()

sum_all() / number("X") >> [ -number("X"), sum_all("X") ]
sum_all("S") / number("X") >> [ -number("X"), "S = int(S) + int(X)", sum_all("S") ]
sum_all("S") >> [ show_line("the sum is", "S") ]

PROFETA.assert_belief(number("4"))
PROFETA.assert_belief(number("8"))
...

PROFETA.achieve(sum_all())
PROFETA.run()
```

Example: the Sieve of Eratosthenes

1. Let us fill our KB with all numbers from 1 to 100, using a belief `number(X)`
2. Let us write a plan that searches for two beliefs representing numbers e.g. `X` and `Y`
3. If `X` can be divided by `Y`, let us remove the belief `number(X)`
4. Let us re-trigger the plan at step 2

```
-number("_") / (number("X") &
                number("Y") &
                (lambda : X != Y) &
                (lambda : (X % Y) == 0) ) >> [ -number("X") ]
```

Example: the Sieve of Eratosthenes

```
from profeta.lib  import *
from profeta.main import *

class number(Belief):
    pass

PROFETA.start()

-number("_") / (number("X") & \
                number("Y") & \
                (lambda : X != Y) & \
                (lambda : (X % Y) == 0) ) >> [ -number("X") ]

for i in range(1,100):
    PROFETA.assert_belief(number(i))

PROFETA.run_shell(globals())
```

PROFETA Semantics

PROFETA Plan Groups

- The **same event** can be used in **different plans**, given that the **condition part is different** as well

1. *EVENT* / *COND1* >> [...]
2. *EVENT* / *COND2* >> [...]
3. *EVENT* / *COND3* >> [...]

- A set of plans with the same event is denoted as **plan group**
- **Only one plan in a group can be selected for execution**

Execution in Plan Groups

1. *EVENT* / *COND1* >> [...]
2. *EVENT* / *COND2* >> [...]
3. *EVENT* / *COND3* >> [...]

- When **EVENT** occurs, the PROFETA engine analyses the relevant plan group, **one plan at time, in strict declaration sequence**
- For each plan, the **condition** is evaluated, if it is **false**, the next plan is analysed
- If the condition is **true**, the plan is **executed**, all the other plans of the group are skipped, for this stage

The Sieve of Eratosthenes - version II

- Let's use plan groups and let's generate all numbers using plans

```
1  class number(Belief):
2      pass
3
4  +number("N") / (lambda : N < 100) >> [ "N = N + 1",
5                                         +number("N") ]
6  +number("N") >> [ show_line("end of sequence generation"),
7                    -number(1) ]
8
9  ...
```

- Generation is triggered by asserting "`number(1)`"
- Plans in lines 4 and 6 could be triggered, but the condition in line 4 is **true** so that plan is executed

The Sieve of Eratosthenes - version II

- Let's use plan groups and let's generate all numbers using plans

```
1  class number(Belief):
2      pass
3
4  +number("N") / (lambda : N < 100) >> [ "N = N + 1",
5                                         +number("N") ]
6  +number("N") >> [ show_line("end of sequence generation"),
7                    -number(1) ]
8
9  ...
```

- In line 4, N is incremented and a new "`number()`" belief is asserted
- This cause plans in the group to be considered again

The Sieve of Eratosthenes - version II

- Let's use plan groups and let's generate all numbers using plans

```
1  class number(Belief):
2      pass
3
4  +number("N") / (lambda : N < 100) >> [ "N = N + 1",
5                                         +number("N") ]
6  +number("N") >> [ show_line("end of sequence generation"),
7                    -number(1) ]
8
9  ...
```

- When N reaches 100, condition in line 4 is false
- This cause execution of plan in line 6 which includes the removal of belief "`number(1)`" in order to start prime numbers detection

The Sieve of Eratosthenes - version II

- Let's detect prime numbers

```
1 ...
2
3 -number("_") / (number("X") & \
4                  number("Y") & \
5                  (lambda : X != Y) & \
6                  (lambda : (X % Y) == 0) ) >> [ -number("X") ]
7 -number("_") >> [ show_line("prime numbers detected") ]
```

- Here plan in line 3 is used to remove non-prime numbers
- And plan in line 7 is executed when there are no more non-prime numbers to be removed

Semantics of KB Modifications

- A plan is executed "atomically" until its completion (unless a failure is detected)
- Therefore, the events generated from actions modifying the KB, i.e. `+bel()` or `-bel()`, are **not immediately handled** but only **after plan execution completion**

```
... >> [ +number(1), show_line("one"),
         -number(2), show_line("two") ]
+number("1") >> [ show_line("adding 1") ]
-number("2") >> [ show_line("removing 2") ]
```

- The output will be:

```
one
two
adding 1
removing 2
```

Semantics of Goal Execution

- Goal achievement is interpreted as a classical "procedure call"
- The current plan is suspended and the goal plan is called

```
... >> [ show_line("one"), g1(), show_line("two") ]
g1() >> [ show_line("this is goal 1"), g2() ]
g2() >> [ show_line("this is goal 2") ]
```

- The output will be:

```
one
this is goal 1
this is goal 2
two
```

Goal Execution and KB modifications

- When goal achievement and KB modifications are mixed, KB events are processed when the **complete goal calling line is over**

```
... >> [ show_line("one"), +number(1),
         g1(), show_line("two") ]
g1() >> [ show_line("this is goal 1"), g2() ]
g2() >> [ show_line("this is goal 2") ]
+number(1) >> [ show_line("number 1") ]
```

- The output will be:

```
one
this is goal 1
this is goal 2
two
number 1
```

Semantics of Special Beliefs

Together with "classical" beliefs, PROFETA provides two kind of **special** beliefs:

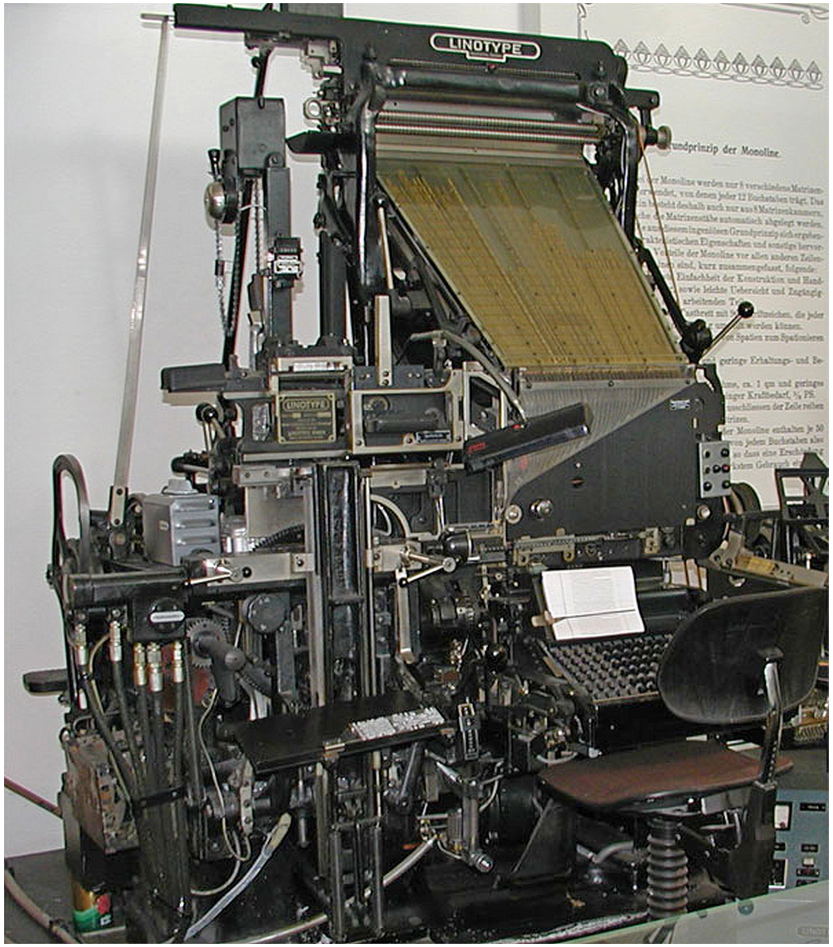
- **Singleton Beliefs**: these are beliefs that can exist in **single instance**
- Executing `+singleton_bel(...)` on an *existing belief* cause that belief to be modified
- A Singleton Belief is declared by subclassing `SingletonBelief`
- **Reactors**: these beliefs do not affect the KB but are only used to generate events
- They are mainly used in "sensors" (see below)
- A Reactor is declared by subclassing `Reactor`

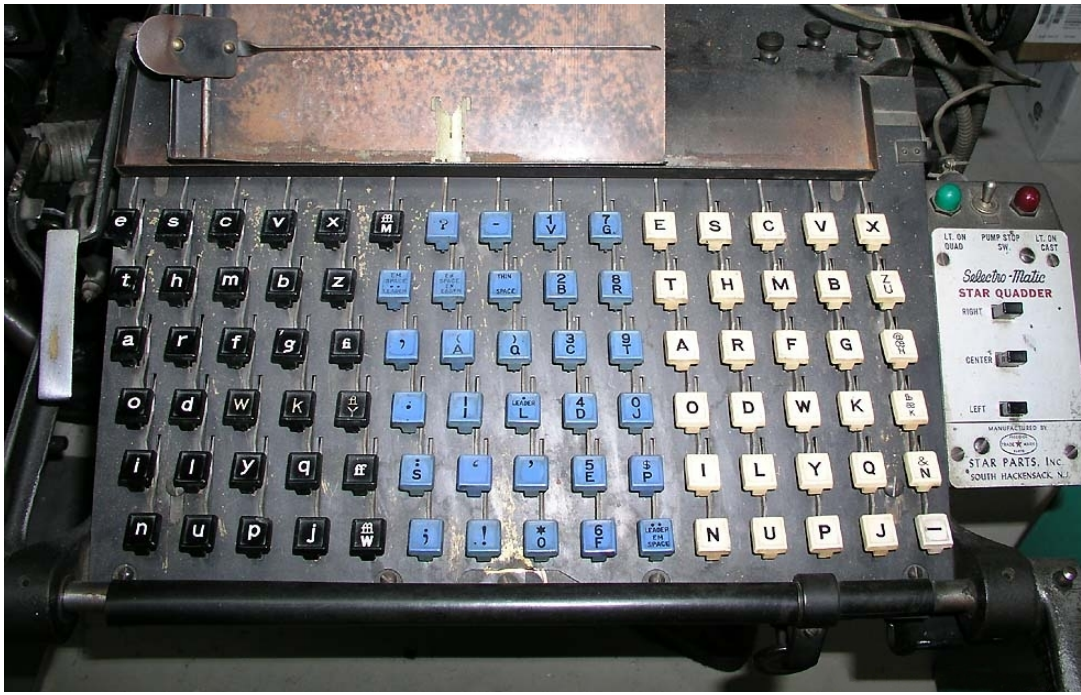
Case Study: SHRDLU

The world of SHRDLU

- **SHRDLU** is one of first artificial intelligence programs (1968-1970) supporting **reasoning/planning** and natural **language processing**
- It is based on a (virtual) environment composed of some "blocks" with different shapes and colors (cubes, pyramids, prisms, cylinders, etc.)
- The program can understand commands like "Pick up a big red block" or "Grasp the pyramid" and behave accordingly
- The name **SHRDLU** comes from the sequence ETAOIN SHRDLU (why?) which is the same sequence of the keys of the Linotype (based on the frequency of letters in the English language)

The Linotype and the SHRDLU sequence





SHRDLU and PROFETA

- Let's implement a small version of SHRDLU with PROFETA (see the code in file "`SHRDLU.py`")
- Our world is based on 3D objects: **cube**, **cylinder**, **prism**
- These objects are placed on a **table** and they can be **picked up** by a robot
- On the table, objects may be **stacked**, one on the top of another, in order to form a **tower**
- An object can be picked only if it is **free** (it has no other objects on the top)

SHRDLU and PROFETA

BELIEFS:

- `obj(X)`, represents the presence of block `X` on the table
- `owned(X)`, represents the fact that the block `X` is **owned** (picked) by the robot
- `upon(X,Y)`, block `X` is placed upon block `Y`

GOALS:

- `pick(X)`, request to pick block `X`, if possible
- `put(X)`, request to put block `X` on the table
- `put(X,Y)`, request to put block `X` upon block `Y`

SHRDLU plans: object picking

- Object X is on the table, but another object Y is upon it ⇒ X cannot be picked

```
pick("X") / (obj("X") & upon("Y", "X")) >> \
 [ show_line("cannot pick", "X", "since it is under the","Y") ]
```

- If previous condition is **false** and object X is on the table and it is on the top of object Y ⇒ X can be picked (update beliefs accordingly)

```
pick("X") / (obj("X") & upon("X", "Y")) >> \
  [ show_line("X", "picked"),
    -obj("X"), -upon("X", "Y"), +owned("X") ]
```

SHRDLU plans: object picking

- If previous condition is **false** and object X is on the table (no other objects on the top of it) ⇒ X can be picked (update beliefs)

```
pick("X") / obj("X") >> [ show_line("X", "picked"),
                          -obj("X"), +owned("X") ]
```

- If previous condition is **false** and object X is owned by the robot ⇒ X cannot be picked

```
pick("X") / owned("X") >> [ show("you've still got", "X") ]
```

- If all previous conditions are **false** the last plan is triggered ⇒ it means that object X does not exist

```
pick("X") >> [ show_line("cannot pick", "X",
                         "since it is not present") ]
```

SHRDLU plans: object picking

- Summary of the plans for object picking:

```
pick("X") / (obj("X") & upon("Y", "X")) >> \
 [ show_line("cannot pick", "X", "since it is under the","Y") ]

pick("X") / (obj("X") & upon("X", "Y")) >> \
  [ show_line("X", "picked"),
    -obj("X"), -upon("X", "Y"), +owned("X") ]

pick("X") / obj("X") >> [ show_line("X", "picked"),
                          -obj("X"), +owned("X") ]

pick("X") / owned("X") >> [ show("you've still got", "X") ]

pick("X") >> [ show_line("cannot pick", "X",
                         "since it is not present") ]
```

SHRDLU plans: object release

- Object X is owned ⇒ put X on the table

```
put("X") / owned("X") >> \
        [ show_line("X", "is now on the table"),
          -owned("X"), +obj("X") ]
```

- Object X is already on the table ⇒ nothing to do

```
put("X") / obj("X") >> \
        [ show_line("X", "is already on the table") ]
```

- Object X is neither held nor on the table ⇒ the object does not exist, nothing to do

```
put("X") >> [ show_line("X", "does not exist") ]
```

SHRDLU plans: object release

We want to put object X on the **top** of object Y:

- Object X is owned, object Y is on the table but Y has another object Z on the top of it ⇒ we cannot do the job

```
put("X", "Y") / (owned("X") & obj("Y") & upon("Z", "Y") ) \
    >> [ show_line("Y", "has", "Z", "on its top") ]
```

- Object X is owned, object Y is on the table but, since here the previous condition is **false**, Y is free ⇒ we can put X upon Y

```
put("X", "Y") / (owned("X") & obj("Y")) >> \
    [ -owned("X"), +obj("X"), +upon("X", "Y"),
      show_line("done") ]
```

SHRDLU plans: object release

- Summary of the plans for object release:

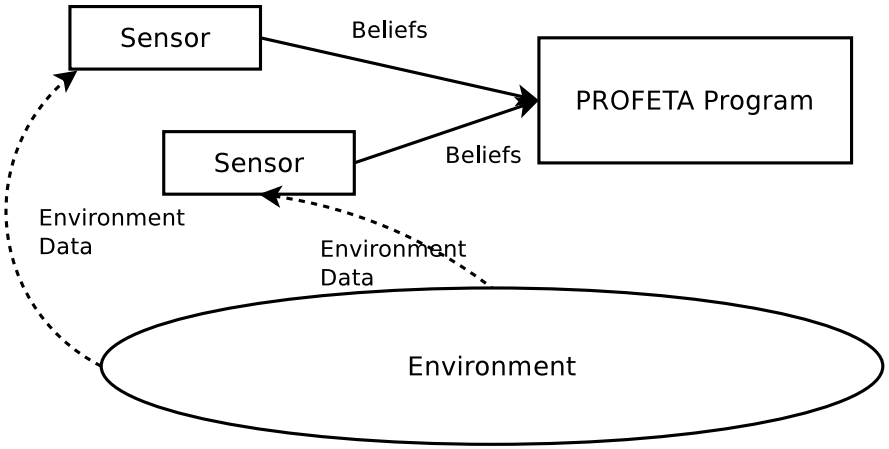
```
pick("X") / (obj("X") & upon("Y", "X")) >> \
 [ show_line("cannot pick", "X", "since it is under the","Y") ]

pick("X") / (obj("X") & upon("X", "Y")) >> \
  [ show_line("X", "picked"),
    -obj("X"), -upon("X", "Y"), +owned("X") ]

pick("X") / obj("X") >> [ show_line("X", "picked"),
                          -obj("X"), +owned("X") ]

pick("X") / owned("X") >> [ show("you've still got", "X") ]

pick("X") >> [ show_line("cannot pick", "X",
                         "since it is not present") ]
```

Let's see a run of SHRDLU

Interaction with the Environment

Sensors

- We used the SHRDLU program from the PROFETA shell, but this is not an usual way to run a program in a production system
- We need a **standalone execution** with proper abstractions to perform interaction with the environment
- In order to **provide data** to a PROFETA program **from the environment**, PROFETA offers the `Sensor` class
- A `Sensor` includes the code to **get data from the environment** and convert it into a **belief** for a PROFETA program



Sensors

- A **sensor** is defined by subclassing `Sensor` and overriding the method `sense`
- Method `sense` must include the code to retrieve data and return:
  - `None`: no interesting data has been sampled
  - `+bel(...)`: a belief is asserted and the relevant **"add" event** is generated
  - `bel(...)`: a belief is asserted but **no event** is generated (only KB is updated)
- Then an instance of that sensor must be added to the PROFETA engine through the `PROFETA.add_sensor()` API function

```
class my_s(Sensor):
    def sense(self):
        # do data poll
        return ...

...
PROFETA.add_sensor(my_s())
```

Sensor Semantics

- All sensors are managed in a list, which is scanned when there are **no pending events**
- Scanning the list means to call method `sense` of all defined sensors
- This implies that the **dynamics** (i.e. **time period**) of environment polling cannot be determined
- If a specific dynamics is needed, class `AsyncSensorProxy` provides a usefull way to perform asynchronous handling

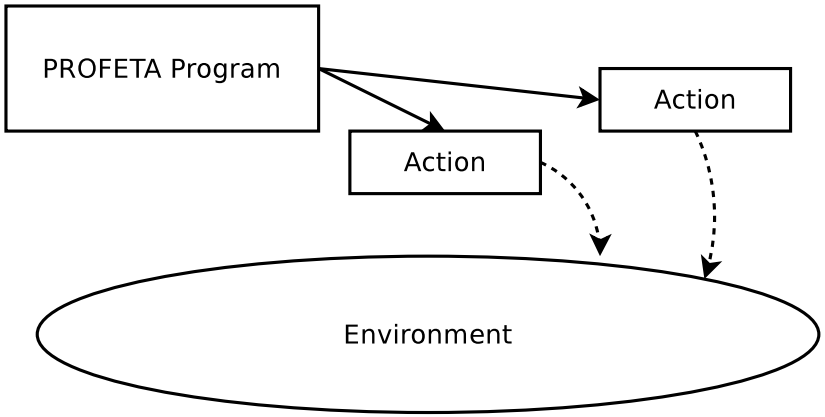
Managing Sensors

- Sensors can be turned **on** and **off** according to system requirements
- When a sensor is **off**, its method `sense` is not called at all
- By default, a sensor is **on**
- To manage sensors, the library actions `sensor_on(...)` and `sensor_off(...)` are provided, which can be used in plans

```
>> [ ..., sensor_on(my_s), ... ]
>> [ ..., sensor_off(my_s), ... ]
```

User-defined Actions

- Acting on the environment implies to write proper code that drives **actuators**
- To this aim, the class `Action` is provided to write a **user-defined action** (UDA)



User-Defined Action

- A UDA is implemented by subclassing `Action` and overriding the method `execute`
- A UDA is called inside a plan by referring it as an atomic formula with one or more parameters
- Parameters are referred, in method `execute`, as `self[0]`, `self[1]`, ...

```
class move_to(Action):
    def execute(self):
        motion_system.perform_move_to(self[0], self[1])


pick_object_at("X", "Y") >> [ move_to("X", "Y") ]
```

Sensors and Actions in SHRDLU

- A real implementation of SHRDLU should consider a "user-friendly" user-interface (not the PROFETA shell)
- A microphone and a speech-to-text API would be ideal, but (in the absence of it) ...
- ... let us implement a **prompt** in which a user can type a command in natural language
- And let us also to write a (very simple!) **natural language parser** in PROFETA

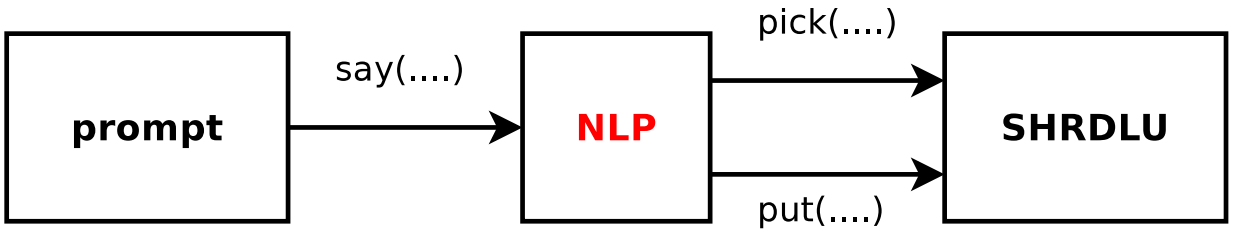
A "Prompt" Sensors

```
class prompt(Sensor):
    def sense(self):
        e = raw_input("COMMAND: ")
        tokens = e.lower().split()
        # use '*' to expand tokens into a sequence of arguments
        return +say(*tokens)
```

Here the `prompt` sensor:

- Waits for a phrase typed by the user
- Splits the phrase into words
- Asserts a "`say(...)`" reactor using the list of words as arguments
- e.g., if the user types "pick the cube", the generated reactor is "`say("pick","the","cube")`"

SHRDLU Data-Flow



- The **prompt** sensor generates the "`say(...)`" reactor
- The **NLP** interprets the "say(...)" reactor and calls one of the goals to pick and put objects
- The NLP is implemented by means a set of PROFETA plans
- The called goals are executed by the **SHRDLU** plans

NLP in SHRDLU

```
+say("V", "X")         / (lambda : V in ['pick', 'get', 'catch' ]) >> [ pick("X") ]
+say("V", "the", "X") / (lambda : V in ['pick', 'get', 'catch' ]) >> [ pick("X") ]
+say("V", "a", "X")    / (lambda : V in ['pick', 'get', 'catch' ]) >> [ pick("X") ]

+say("V", "X")         / (lambda : V in ['put', 'release' ]) >> [ put("X") ]
+say("V", "the", "X") / (lambda : V in ['put', 'release' ]) >> [ put("X") ]

+say("V", "the", "X", "upon", "the", "Y") / (lambda : V in ['put',  'release' ]) >> \
                                                                  [ put("X", "Y") ]
+say("V", "X", "upon", "the", "Y") / (lambda : V in ['put', 'release' ]) >> \
                                                                  [ put("X", "Y") ]
+say("V", "the", "X", "upon", "Y") / (lambda : V in ['put', 'release' ]) >> \
                                                                  [ put("X", "Y") ]
+say("V", "X", "upon", "Y") / (lambda : V in ['put', 'release' ]) >> [ put("X", "Y") ]

# additional plans for other usefull commands
+say("show", "me", "the", "table") >> [ show_table() ]
+say("show", "the", "table") >> [ show_table() ]
+say("show", "table") >> [ show_table() ]

+say("show", "me", "the", "tower", "of", "X") >> [ tower("X") ]
+say("show", "me", "tower", "of", "X") >> [ tower("X") ]
+say("show", "the", "tower", "of", "X") >> [ tower("X") ]
+say("show", "tower", "of", "X") >> [ tower("X") ]

+say("show", "my", "objects") >> [ show_my() ]

# last plan triggered when any set of paramters is given
+say("*") >> [ show_line("what?") ]
```

References

- `http://github.com/corradosantoro/profeta`
- A. Rao, M. Georgeff, *BDI agents: From theory to practice*, in: Proceedings of the first international conference on multi-agent systems (ICMAS-95), San Francisco, CA, 1995, pp. 312319.
- M. E. Bratman, *Intentions, Plans and Practical Reason*, Harvard University Press, 1987.
- A. Rao, *AgentSpeak (L): BDI agents speak out in a logical computable language*, Lecture Notes in Computer Science 1038 (1996) 4255.
- A. Pokahr, L. Braubach, W. Lamersdorf, *Jadex: A BDI reasoning engine*, Multiagent Systems Artificial Societies and Simulated Organizations 15 (2005) 149.

References

- Jason Home Page, `http://www.jason.sourceforge.net/`.
- Siegwart, I. R. Nourbakhsh, D. Scaramuzza, *Introduction to autonomous mobile robots*, MIT press, 2011.
- O. Boissier, R. H. Bordini, J. F. Hbner, A. Ricci, A. Santi, *Multi-agent oriented programming with jacamo*, Science of Computer Programming 78 (6) (2013) 747761.
- L. Fichera, D. Marletta, V. Nicosia, C. Santoro. *Flexible Robot Strategy Design using Belief-Desire-Intention Model*. In Proc. of *International Conference on Research and Education in Robotics (EUROBOT 2010)*, Spriger CCIS Series, Rappreswille, Swizerland), May 27-30, 2010.
- L. Fichera, D. Marletta, V. Nicosia, C. Santoro. *A Methodology to Extend Imperative Languages with AgentSpeak Declarative Constructs*. In Proc. of *Workshop on Objects and Agents (WOA2010)*, CEUR-WS Publisher, ISSN 1613-0073, Rimini, Italy, Sept. 5-7, 2010.

References

- G. Fortino, W. Russo, C. Santoro. *Translating Statecharts-based into BDI Agents: The DSC/PROFETA case*. MAS&S Workshop @ MATES 2013.
- F. Messina, G. Pappalardo, C. Santoro. *Integrating Cloud Services in Behaviour Programming for Autonomous Robots*. CSmart-CPS Workshop, LNCS, 2013.
- F. Messina, G. Pappalardo, C. Santoro. *A Goal-centric Framework for Behaviour Programming in Autonomous Robotic Systems*. IEEE/ASME International Conference on Mechatronics and Embedded Systems, 2014.
- F. Messina, G. Pappalardo, C. Santoro. *Designing Autonomous Robots using GOLEM*. In Proc. of *Workshop on Objects and Agents (WOA2014)*, CEUR-WS Publisher, Catania, Italy, Sept. 2014.

Programming Autonomous Robots using PROFETA and the BDI model

Corrado Santoro
http://www.dmi.unict.it/~santoro

ARSLAB - Autonomous and Robotic Systems Laboratory
Dipartimento di Matematica e Informatica
Università di Catania, Italy
santoro@dmi.unict.it