Logic Programming with PROFETA

Corrado Santoro

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

Programmazione Sistemi Robotici

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") ]

# run the engine shell
PROFETA.run_shell(globals())
```

PROFETA Reactive Plan Example

```
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")]

# run the engine shell
PROFETA.run_shell(globals())
```

- The plan is triggered when a belief `number(...)` is asserted
- In this case, variable **X** will be bound to the parameter of the belief and will be printed in the message
- Variables are **local** of the plan

PROFETA Reactive Plan Example

```
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")]

# run the engine shell
PROFETA.run_shell(globals())
```

- In PROFETA, strings starting with **uppercase** represents **variables**

Graduated Students: A more “rational” Example

- We want to represent a world in which we have some students that, sooner or later, become graduated
- When a student become graduated, s/he is no more a “student”
- We use two beliefs:
  - **`student(X)`** to represent the fact that **“X”** is a student
  - **`graduated(X)`** to represent the fact that **“X”** is graduated
- We have the following “knowledge rules”:
  - **“X”**, to be **`graduated(X)`**, must be (before) **`student(X)`**
  - When **“X”** becomes **`graduated(X)`** it is no more a **`student(X)`**

```
class student(Belief):
    pass

class graduated(Belief):
    pass

+graduated("X") / student("X") >> [ -student("X"),
     show_line("yeah ", "X", "is now graduated!") ]
+graduated("X") >> [ show_line("X", "is not a student"),
                     -graduated("X") ]
```

Graduated Students: A more “rational” Example

```
class student(Belief):
    pass

class graduated(Belief):
    pass

+graduated("X") / student("X") >> [ -student("X"),
      show_line("yeah ", "X", "is now graduated!") ]
+graduated("X") >> [ show_line("X", "is not a student"),
                     -graduated("X") ]
```

- Both plans have the same **triggering event**: `+graduated("X")`
- These plans represent a **group of plans** since they share the same triggering event
- **Only a plan** in a group can be executed
- Selection is made on the basis of **writing order**: the first plan that matches the condition is executed

Reactive Factorial

```
class factorial(Reactor):
    pass

+factorial(0, "Acc") >> [ show_line("factorial is ", "Acc") ]
+factorial("N", "Acc") >> [ "Acc = Acc * N", "N = N - 1",
                            +factorial("N", "Acc") ]

+factorial("N") >> [ +factorial("N", 1) ]
```

- Here `+factorial(...)` is used only to **generate an event** but not to **represent a knowledge**
- For this reason `factorial(...)` is defined as a **Reactor**, i.e. a belief that does not populate the knowledge base but can trigger plans
- To perform an iteration, implementation exploits “plan retriggering”, a technique which is similar to **recursion**
- The recursion exploits an additional belief variable **”Acc”** (stands for “accumulator”) which stores the partial result of the factorial

Factorial with Goal

```
class factorial(Goal):
    pass

factorial(0, "Acc") >> [ show_line("factorial is ", "Acc") ]
factorial("N", "Acc") >> [ "Acc = Acc * N", "N = N - 1",
                           factorial("N", "Acc") ]

factorial("N") >> [ factorial("N", 1) ]
```

- The same result can be achieved by using a **Goal** instead of a Reactor
- A Goal is simular to a “procedure call” so, in this case, is more appropriate than using the Reactor

Using Conditions: Summing *n* numbers

- Let's sum *n* numbers each represented by belief `number(X)`
- We use a goal **sum_all("N")** that and a plan that "retrieves" a number from the KB and sums it to *N*
- Each number processed is then removed from the knowledge base

```
class sum_all(Goal):
    pass

class number(Belief):
    pass

sum_all("N") / number("X") >> [ "N = N + X", -number("X"),
                                sum_all("N") ]
sum_all("N") >> [ show_line("the sum is", "N") ]
sum_all() >> [ sum_all(0) ]
```

Using Conditions: Summing *n* numbers

- Plan

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

  contains a **condition**

- It can be triggered if a belief `number` is present in the knowledge base
- Variable *X*, since it is not used before, is a **free variable** and can thus be bound to any value
- The result is a match to **any** belief `number`

Generative Reactive Programming: the Sieve of Eratosthene

- To implement the sieve of Eratosthene we consider numbers each one represented by belief `number(X)`
- A plan is used to find pairs (`number(X)`, `number(Y)`) and remove the former belief if **X** is a multiple of **Y**
- Here we use a condition made of an AND-composite predicate that also uses lambdas to implement boolean conditions

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

class number(Belief):
    pass

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

# populate the KB
for i in range(2,100):
    PROFETA.assert_belief(number(i))
```

Generative Reactive Programming: the Sieve of Eratosthene

- Plan

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

  is triggered when a `number(X)` is asserted

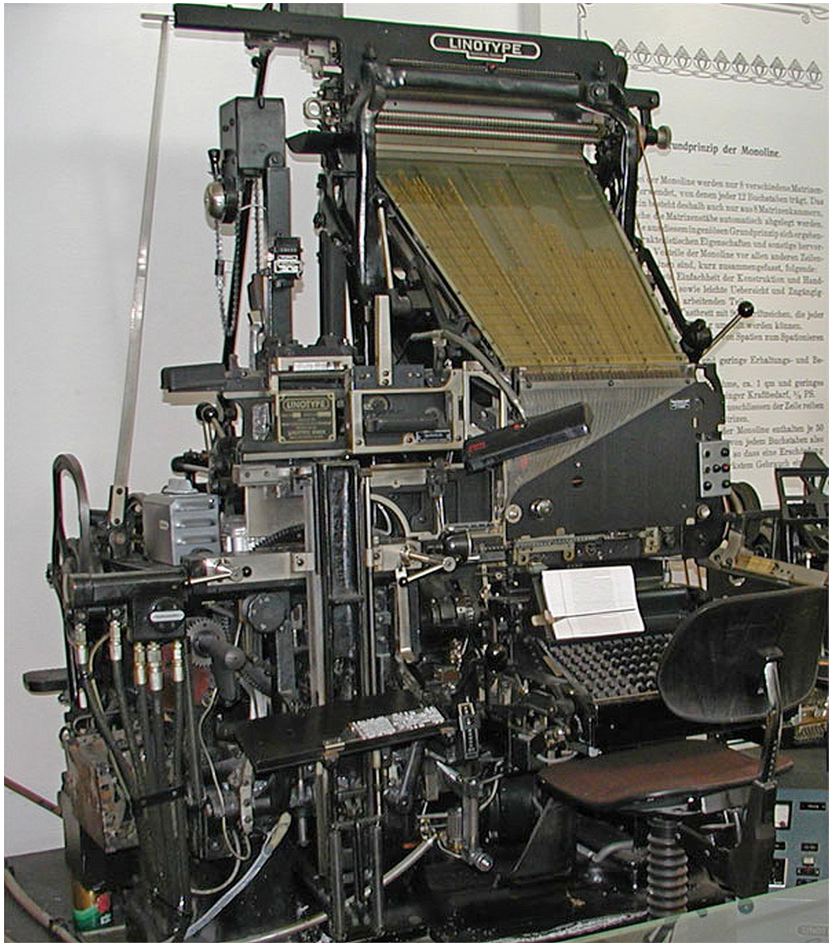
- The condition searches for any another `number(Y)`, but the other checks ensure that
  1. X and Y are different (i.e. the beliefs do not refer to the same number)
  2. X is a multiple of Y (their "module" operation results to zero)
- If all checks are true, `number(X)` (the multiple) is removed from the KB

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 $\Rightarrow$ 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 $\Rightarrow$ 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) $\Rightarrow$ 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 $\Rightarrow$ 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 $\Rightarrow$ 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

Logic Programming with PROFETA

Corrado Santoro

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

Programmazione Sistemi Robotici