Reactive Autonomous System Programming using the PROFETA tool

Corrado Santoro

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

Miniscuola WOA 2013 - Torino, 04 Dicembre 2013

Outline

Introducing PROFETA

Introducing PROFETA

PROFETA Basics

- PROFETA *(Python RObotic Framework for dEsigning sTrAtegies*) is Python tool for programming autonomous systems (agents or *robots*) using a **declarative approach**.
- PROFETA has been designed at ARSLAB @ UNICT in 2010.
- 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.

PROFETA Basics

- PROFETA provides
  - A set of classes to represent basic BDI entities, i.e.
    - Beliefs $\Rightarrow$ Knowledge
    - Goals $\Rightarrow$ States to be achieved
    - Actions $\Rightarrow$ Things to do to reach the Goals
  - A declarative language—dialect of AgentSpeak(L)—to express agent's behaviour
  - An engine executing that behaviour
  - A Knowledge base storing the beliefs

PROFETA Declarative Syntax

- A PROFETA behaviour is a set of **rules** in the form
  Event" /" Condition" >>" set_of Action
  where:
  - *Event* can be: belief assert or retract, goal achievement request, goal failure.
  - *Condition* refers to a certain state of the knowledge base.
  - *Action* can be: belief assert or retract, goal achievement request, user defined *atomic* action.

Example:

```
+object_at("X","Y") / object_got("no") >>
      [ move_to("X", "Y"), pick_object() ]
```

- Such expressions are managed thanks to **operator overloading**

Basic Parts of a PROFETA Program

- Download PROFETA from:
  `http://github.com/corradosantoro/profeta`

- Import the PROFETA libraries
- Define **beliefs** and **goals** as subclasses of `Belief` and `Goal`
- Define user **actions** by subclassing `Action` and overriding method `execute()`
- Instantiate the PROFETA Engine
- Define the **rules** by means of the declarative sytntax
- Run the Engine

“Hello World” in PROFETA

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

# instantiate the engine
PROFETA.start()

# define a rule
+start() >> [ show_line("hello world from PROFETA") ]

# assert the "start()" belief
PROFETA.assert_belief(start())

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

- `start()` is a library belief
- `show_line()` is a library action

The “Picking Object” Robot

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

class object_at(Belief):
    pass

class object_got(Belief):
    pass

class move_to(Action):
    def execute(self):
        x = self[0]
        y = self[1]
        # ... perform movement to x,y

class pick_object(Action):
    def execute(self):
        # drive the arm to pick object

PROFETA.start()

+object_at("X", "Y") / object_got("no") >>
    [ move_to("X", "Y"), pick_object(), -object_got("no"), +object_got("yes"),
      -object_at("X", "Y") ]

PROFETA.assert_belief(object_got("no"))
PROFETA.run()
```

Belief Syntax

- A **belief** is defined as a subclass of `Belief`
- In a rule, it is referred as an **"atomic formula"** with:
  - one or more ground terms, e.g. `object_at(123, 456)`
  - one or more variables, e.g. `object_at("X", "Y")`
  - A variable is expressed using a string starting with capitals
- Within the set of actions:
  - `+bel(...)` asserts a belief in the KB
  - `-bel(...)` retracts a belief from the KB
- As trigger event:
  - `+bel(...)` triggers the rule when the belief is asserted
  - `-bel(...)` triggers the rule when the belief is retracted

Action Syntax

- An **action** is defined as a subclass of `Action`
- The `execute()` method has to implement the computation which concretely performs the action; such a computation is executed **atomically**
- In a rule, it is referred as an **"atomic formula"** with:
  - one or more ground terms, e.g. `move_to(123, 456)`
  - one or more variables, e.g. `move_to("X", "Y")`
  - A variable is expressed using a string starting with capitals
- In method `execute()` terms can be accessed as `self[`*term_index*`]`

The “Picking Robot” Rule

```

+object_at("X", "Y") / object_got("no") >>
    [ move_to("X", "Y"), pick_object(), -object_got("no"), +object_got("yes"),
      -object_at("X", "Y") ]
```

That rule says:

- When an object is seen at a certain position and ...
- No object has been picked from the robot, then ...
- Move to that position, pick the object, and ...
- Update the beliefs in order to reflect the new state.

Sensors

- Belief `object_at()` is intended to be asserted as soon as an object is found (or detected) at a certain position.
- How is such a detection performed?
- A **Sensor** class is provided with the objective of allowing the programmer to write the proper code to “sense” the environment and generate the proper beliefs.
- The programmer has to:
  - subclass **Sensor**
  - override the `sense()` method
  - inform the Engine that a new sensor has been added in the program.

The “Picking Object” Robot with a Sensor

```

 ....

class ObjectSensor(Sensor):
    def sense(self):
        Obj = detect_an_object()
        if Obj is None:
            return None
        else:
            (x, y) = Obj
            return +object_at(x, y)

PROFETA.start()

+object_at("X", "Y") / object_got("no") >>
    [ move_to("X", "Y"), pick_object(), -object_got("no"), +object_got("yes"),
      -object_at("X", "Y") ]

PROFETA.assert_belief(object_got("no"))
PROFETA.add_sensor(ObjectSensor())
PROFETA.run()
```

Sensor Semantics

```
....

class ObjectSensor(Sensor):
    def sense(self):
        Obj = detect_an_object()
        if Obj is None:
            return None
        else:
            (x, y) = Obj
            return +object_at(x, y)

...
```

Method `sense()` may return

- `None`, nothing has been sensed
- `+bel(...)`, something has been sensed, a belief is added to the KB and the “add event” is generated.
- `bel(...)`, something has been sensed, a belief is added to the KB but the “add event” is NOT generated.

PROFETA Structures

Terminology (according to AgentSpeak):

- A **Plan** is a rule of the PROFETA program
- An **Intention** is a plan which is selected for execution (the instantiation of a plan)

PROFETA Structures:

- An **Event Queue** which stores add/retract belief events
- An **Intention Stack** which stores Intentions to be executed
- A **Sensor Set** which stores instantiated sensors

PROFETA (Informal) Execution Semantics

PROFETA Main Loop Execution:

1. If the Intention Stack is not empty, execute an action step and go to 1.
2. If the Event Queue is not empty, pick an event, find the relevant plan, verify the condition part and then put the plan at the top of Intention Stack, then go to 1.
3. If both the Intention Stack and Event Queue are empty, scan sensors in Sensor Set and, for each of them, call the `sense()` method and analyse the return value; if it is an event, put it in the Event Queue.
4. Go to 1.

Special features of PROFETA

Special features of PROFETA

Special Kind of Beliefs

```

+object_at("X", "Y") / object_got("no") >>
    [ move_to("X", "Y"), pick_object(), -object_got("no"), +object_got("yes"),
      -object_at("X", "Y") ]
```

Here ...

- Belief `object_at()` represents that "an object has been detected"; after plan execution, the belief is removed.
- Belief `object_got()` represents a state of the robot, which could have picked or not the object. Only one belief of this kind can be present in the KB.

Special Kind of Beliefs

To support such cases, the following “special beliefs” are provided:

- `Reactor`s, i.e. beliefs which are automatically removed from the KB when the associated plan is executed.
- `SingletonBelief`s, i.e. beliefs which are “single instance”.

Therefore ...

```
...

class object_at(Reactor):
    pass

class object_got(SingletonBelief):
    pass

...

+object_at("X", "Y") / object_got("no") >>
    [ move_to("X", "Y"), pick_object(), +object_got("yes") ]
...
```

Example: Factorial in PROFETA Using Reactors

```

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

class fact(Reactor):
    pass

class KB(Sensor):
    def sense(self):
        e = raw_input ("Enter Number: ")
        return +fact(int(e), 1)


PROFETA.start()
PROFETA.add_sensor(KB())

+fact(1, "X") >> [ show("the resuilting factorial is "), show_line("X") ]
+fact("N", "X") >> [ "Y = int(N) * int(X)", "N = int(N) - 1", +fact("N", "Y") ]

PROFETA.run()
```

Special Kind of Actions

```
...

+fact(1, "X") >> [ show("the resuilting factorial is "), show_line("X") ]
+fact("N", "X") >> [ "Y = int(N) * int(X)", "N = int(N) - 1", +fact("N", "Y") ]

...
```

- `show()` and `show_line()` are *library actions* which are used to print an output onto the screen.
- An action can also be any Python expression, given that it is expressed as a string (*string action*)
- A string action can manipulate and create any plan variable but... look out!! PROFETA is not typed!

Goals in PROFETA

Goals in PROFETA

Goals

- **Goals** are (according to AgentSpeak(L)) "states of the systems which an agent wants to achieve", but indeed ...
- ... a goal is something like a "procedure plan" which expresses the things to do to achieve the goal itself.
- In PROFETA, a goal:
  - is first defined as a subclass of **Goal**
  - it is expressed, in the plans, as an "atomic formula", with zero or more terms.

Example: Factorial with goals

```

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

class do_factorial(Reactor):
    pass

class fact(Goal):
    pass

class KB(Sensor):
    def sense(self):
        e = raw_input ("Enter Number: ")
        return +do_factorial(int(e))


PROFETA.start()
PROFETA.add_sensor(KB())

+do_factorial("N") >> [ show("computing factorial of "), show_line("N"),
                        fact("N", 1) ]
fact(1, "X") >> [ show("the resuilting factorial is "), show_line("X") ]
fact("N", "X") >> [ "Y = int(N) * int(X)", "N = int(N) - 1", fact("N", "Y") ]

PROFETA.run()
```

The “Object Picker” with Goals

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

class object_at(Reactor):
    pass

class object_got(SingletonBelief):
    pass

class pick_object_at(Goal):
    pass

class move_to(Action):
    def execute(self):
        # ... perform movement to x,y

class drive_arm(Action):
    def execute(self):
        # drive the arm to pick object

PROFETA.start()

+object_at("X", "Y") / object_got("no") >> [ pick_object_at("X", "Y"),
                                            +object_got("yes") ]
pick_object_at("X", "Y") >> [ move_to("X", "Y"), drive_arm() ]

PROFETA.assert_belief(object_got("no"))
PROFETA.run()
```

Why Goals?

- Goals are "procedures", thus they help to better organize your PROFETA program, e.g.

```
... >> [ goal1(), goal2(), goal3() ]
goal1() >> [ ... do something ... ]
goal2() >> [ ... do something ... ]
goal3() >> [ ... do something ... ]
```

- Goals can introduce **branches** since a condition is allowed in the head of the rule, e.g.

```
pick_object_at("X", "Y") / (lambda : X < 100) >> [ move_to("X", "Y"), ... ]
pick_object_at("X", "Y") / (lambda : X >= 100) >> [ # do other things ]
```

- Goals may **fail**!

Goal Failure

- Let us suppose that the robot cannot reach the object if it is located around position $(1500, 1000)$, so:

```
...
def is_reachable(x,y):
    return math.hypot(x - 1500, y - 1000) > 100

+object_at("X", "Y") / object_got("no") >> [ pick_object_at("X", "Y"),
                                             +object_got("yes") ]
pick_object_at("X", "Y") / (lambda : is_reachable(X,Y)) >>
                              [ move_to("X", "Y"), ... ]
pick_object_at("X", "Y")  >> [ fail() ] # the goal fails!
```

- Library action `fail()` causes the current goal to fail
- Calling goals/plans fail in sequence (is something like an exception)

Recovery Actions

- A goal failure can be triggered also within the code of an Action, by calling `fail().execute()`
- A failure event can be catched by a rule in order to try a recovery action
- To catch a failure of a goal **g**, the related event is `-g()`, e.g.

```
...
+object_at("X", "Y") / object_got("no") >> [ pick_object_at("X", "Y"),
                                             +object_got("yes") ]
pick_object_at("X", "Y") / (lambda : is_reachable(X,Y)) >>
                             [ move_to("X", "Y"), ... ]
pick_object_at("X", "Y")  >> [ fail() ] # the goal fails!
-pick_object_at("X", "Y") >> [ ... do smthg to perform recovery ... ]
```

Goal Failure Semantics

- If we have

```
1 ... >> [ goal1(), ...  ]
2 goal1() >> [ goal2() ]
3 goal2() >> [ goal3() ]
4 goal3() >> [ ..., fail() ]
```

- In plan in line 4, we have a stack of goal calls `goal1()` → `goal2()` → `goal3()`, and `goal3()` fails, thus
- If a plan related to `-goal3()` exists, it is first executed;
- otherwise, if a plan related to `-goal2()` exists, it is executed;
- otherwise, if a plan related to `-goal1()` exists, it is executed;
- otherwise, the calling plan of `goal1()` (line 1) is interrupted.

Contexts in PROFETA

Contexts in PROFETA

Scenario 1: Synchronous vs. Asynchronous Actions

- According to PROFETA execution semantics, Sensors are polled only if there are no intentions to be executed
- Therefore, during intention execution, an agent/robot is **blind**, w.r.t. events which could happen in the environment
- Action dynamics/duration matters!! E.g.
  - If action `move_to(X,Y)` is *synchronous*, i.e. it waits for the robot to reach the desired position, no events can be captured during the move!
  - In a robotic application this is quite **undesirable**.
- Solution:
  - Make action `move_to(X,Y)` **asynchronous** and add a Sensor checking that the target position has been reached.

The “Object Picker” with Asynchronous move_to()

```
...
class object_at(Reactor): pass

class object_got(SingletonBelief): pass

class target_got(Reactor): pass

class move_to(Action):
    def execute(self):
        # ... trigger movement to x,y and immediatelly return

class drive_arm(Action):
    def execute(self):
        # drive the arm to pick object

class TargetGot(Sensor):
    def sense(self):
        if motion_control.target_reached(): return +target_got()
        else: return None

PROFETA.start()
PROFETA.add_sensor(TargetGot())

+object_at("X", "Y") / object_got("no") >> [ move_to("X", "Y") ]
+target_got() >> [ drive_arm(), +object_got("yes") ]

PROFETA.assert_belief(object_got("no"))
PROFETA.run()
```

Uncontextualised beliefs

- Here, `target_got()` is asserted whatever movement has been started.
- If we want to check that *a certain movement* is completed, we should add a proper belief, e.g. to make the robot following a path:

```
... >> [ move_to(1000, 0), +movement("one") ]
+target_got() / movement("one") >>[ move_to(1000, 1000), +movement("two") ]
+target_got() / movement("two") >>[ move_to(0, 1000), +movement("three") ]
+target_got() / movement("three") >> [ show("yeahhh!!!") ]
...
```

- that is, we have to add a *contextual information* by means of additional beliefs.

Scenario 2: A robot making "many things"

- Let's consider a robot which has to:
- 1. Gather at most 10 objects placed in the environment
  2. Put the object into a specific container
  3. Go to step 1.
- Here we have two tasks or "macrostates" in which events sensed from the environment may have require different actions.
- Contextual beliefs could be used but ...
- ... it could be better to have language constructs which allows a **better identification of the plans related to each task**.

Contexts in PROFETA

- PROFETA let a programmer to define **contexts**:
  <u>macrostates in which only specified plans can be triggered</u>.
- A context is defined by placing the statement `context(ContextName)` in the plan list; after it, all plans will be referred to `ContextName`.
- At run-time, a context is "entered" by calling the library action `set_context(ContextName)`.
- This action asserts the library reactor `start()` which is used to trigger an "intialization plan" in the entered context.

Contexts in PROFETA: An Example

```
...
class go(Goal): pass

go() >> [ set_context("pick_objects") ]

context("pick_objects")
# put here plans for 'pick_object' task
+start() >> [ +objects_got(0) ]
+object_detected_at("X", "Y") >> [ move_to("X", "Y") ]
+target_got()  / objects_got("N") >> [ drive_arm(), "N␣=␣N␣+␣1",
                                       +objects_got("N"), after_pick() ]
after_pick() / objects_got(10) >> [ set_context("put_away_objects") ]


context("put_away_objects")
# put here plans for 'put_away_object' task
+start() >> [ move_to(1500, 1000) ]  # the position of the container
+target_got() >> [ release_all_objects(), set_context("pick_objects") ]


PROFETA.achieve(go())
PROFETA.run()
```

Other features

Other features

Handling specific Sensor Dynamics

- Sensors are polled using the latency derived from the execution of the main PROFETA loop
- If you need to poll a Sensor with a specific dynamics, the **AsyncSensorProxy** allows a Sensor to be decoupled from the main loop
- It can have its own peridicity (*NOT guaranteed!!*).

```
...
class MyAsyncSensor(Reactor):
    def sense(self):
        time.sleep(millisecs/1000.0)
        # perform sensing

PROFETA.add_sensor(AsyncSensorProxy(MyAsyncSensor()))
```

Handling specific Action Dynamics

- Actions can be easily made asynchronous by subclassing `AsyncAction` instead of `Action`.
- An AsyncAction is executed in a different thread w.r.t. that of PROFETA main loop

```
...
class MyAsyncAction(AsyncAction):
    def execute(self):
        # ....


... >> [ ... MyAsyncAction(...) ... ]
```

Conclusions

- PROFETA allows to define robot strategies in a flexible and intuitive way, by using the principles of the BDI model.
- PROFETA combines the advantages of both imperative and declarative approach, while offering an all-in-one environment
- PROFETA performs quite well also in systems with few memory, and has been successfully applied in various robot built at ARSLab

TO DO List

- **Concurrent Plans**. But under the **strong** control of the programmer.
- **MAS Support**. Currently PROFETA does not support muliple-agents: one robot/agent for virtual machine.
- **Better notion of Task (or "Goal")**. A Task is what the agent should do to achieve a specific goal. Why not **mulitple and different choices** for a goal/task? This could give a "deliberation ability" to agents.
- **Porting to other languages**. Only Python? Why not C++?

References

- `http://github.com/corradosantoro/profeta`
- 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.
- 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.

Reactive Autonomous System Programming using the PROFETA tool

Corrado Santoro

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

Miniscuola WOA 2013 - Torino, 04 Dicembre 2013