

Declarative Programming in Python using PHIDIAS

Corrado Santoro

ARSLAB - Autonomous and Robotic Systems Laboratory

Dipartimento di Matematica e Informatica - Università di Catania, Italy

`santoro@dmi.unict.it`



Robotic Systems

- PHIDIAS (*PytHon Interactive Declarative Intelligent Agent System*) is a Python tool to program *autonomous systems* using the **declarative** paradigm
- The aim is to offer an all-in-one programming environment that allows a developer to write together both **imperative** and **declarative** code
- It is available on github:
`http://github.com/corradosantoro/phidias`

- PHIDIAS is based on the paradigm **belief-desire-intention (BDI)** and allows a developer to specify, within Python code, of behaviours based on:
- Knowledge (base and derived)
- “Reactive” rules (event-condition-action model)
- “Proactive” rules

Knwoledge in PHIDIAS

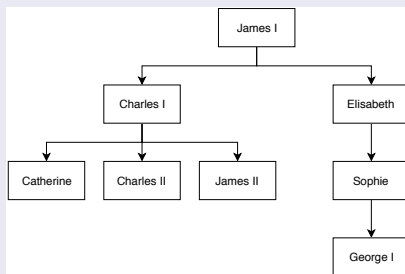
Knowledge in PHIDIAS—Beliefs

- The knowledge model in PHIDIAS is similar to that of Prolog
- The basic knowledge is specified through **Beliefs** (equivalent to Prolog “facts”)
- The Beliefs must be **declared** by subclassing the base class **Belief**
- Then they can be expressed by means of the classical syntax based on **atomic formulae** with ground terms:
 - `position(150, 60)`
 - `block("red")`
 - `stack("cube", "pyramid")`
- Beliefs can be **added** to or **removed** from the **Knowledge Base (KB)**
- The KB can be **queried** to check the presence of certain facts and behave accordingly

Knowledge in PHIDIAS—Goals

- The **derived** knowledge is represented by **Goals**
- Goals must be **declared** by subclassing the base class **Goal**
- Then they can be used by means of logic formulae with AND connectives, in first-order logic
- Formulae can contains variables (universally quantified) or ground terms
- But used variables must be declared (not the type but their “presence”)

Example: the Genealogic Tree



Beliefs

- **male**(X) \rightarrow X is a man
- **female**(X) \rightarrow X is a woman
- **parent**(X, Y) \rightarrow X is parent of Y

Example: the Genealogic Tree

Derived Knowledge: Goals

- **father**(**X**, **Y**) \rightarrow X is Y's father:

$$\forall x, y : \text{parent}(x, y) \wedge \text{male}(x) \Rightarrow \text{father}(x, y)$$

- **mother**(**X**, **Y**) \rightarrow X is Y's mother:

$$\forall x, y : \text{parent}(x, y) \wedge \text{female}(x) \Rightarrow \text{mother}(x, y)$$

Example: the Genealogic Tree (1)

```
# library imports
from phidias.Types import *
from phidias.Main import *
from phidias.Lib import *

# definizione dei beliefs
class parent(Belief): pass
class male(Belief): pass
class female(Belief): pass

# definizione dei goal
class father(Goal): pass
class mother(Goal): pass

# definizione delle variabili usate
# nel programma (occhio agli apici!)
def_vars('X','Y')

# definizione dei goal come predicati logici
father(X,Y) << ( parent(X,Y) & male(X) )
mother(X,Y) << ( parent(X,Y) & female(X) )

# ....
```

Example: the Genealogic Tree (2)

```
# ...  
# inserimento iniziale nella knowledge base  
PHIDIAS.assert_belief(male('james1')),  
PHIDIAS.assert_belief(male('charles1'))  
PHIDIAS.assert_belief(male('charles2'))  
PHIDIAS.assert_belief(male('james2'))  
PHIDIAS.assert_belief(male('george1'))  
PHIDIAS.assert_belief(female('catherine'))  
PHIDIAS.assert_belief(female('elizabeth'))  
PHIDIAS.assert_belief(female('sophia'))  
PHIDIAS.assert_belief(parent('james1', 'charles1'))  
PHIDIAS.assert_belief(parent('james1', 'elizabeth'))  
PHIDIAS.assert_belief(parent('charles1', 'charles2'))  
PHIDIAS.assert_belief(parent('charles1', 'catherine'))  
PHIDIAS.assert_belief(parent('charles1', 'james2'))  
PHIDIAS.assert_belief(parent('elizabeth', 'sophia'))  
PHIDIAS.assert_belief(parent('sophia', 'george1'))  
  
# start dell'engine di PHIDIAS  
PHIDIAS.run()  
  
# start della shell interattiva  
PHIDIAS.shell(globals())
```

Run Example: the Genealogic Tree

```
PHIDIAS Release 1.1.0  
Autonomous and Robotic Systems Laboratory  
Department of Mathematics and Informatics  
University of Catania, Italy (santoro@dmf.unict.it)
```

```
eShell:  main > kb  
male('james1')           male('charles1')  
male('charles2')         male('james2')  
male('georgel')          female('catherine')  
female('elizabeth')      female('sophia')  
parent('james1', 'charles1') parent('james1', 'elizabeth')  
parent('charles1', 'charles2') parent('charles1', 'catherine')  
parent('charles1', 'james2')   parent('elizabeth', 'sophia')  
parent('sophia', 'georgel')  
eShell:  main >
```

Run Example: the Genealogic Tree (2)

```
eShell:  main > father(X,Y)
father('james1', 'charles1')
father('james1', 'elizabeth')
father('charles1', 'charles2')
father('charles1', 'catherine')
father('charles1', 'james2')
eShell:  main > mother(X,Y)
mother('elizabeth', 'sophia')
mother('sophia', 'georgel')
eShell:  main >
```

Example: the Genealogic Tree (3)

Derived Knowledge

- **sibling(X, Y)** \rightarrow X is Y's sibling (X and Y have the same parent):
 $\exists p, \forall x, y : \text{parent}(p, x) \wedge \text{parent}(p, y) \wedge x \neq y \Rightarrow \text{sibling}(x, y)$
- **brother(X, Y)** \rightarrow X is Y's brother:
 $\forall x, y : \text{sibling}(x, y) \wedge \text{male}(x) \Rightarrow \text{brother}(x, y)$
- **sister(X, Y)** \rightarrow X is Y's sister:
 $\forall x, y : \text{sibling}(x, y) \wedge \text{female}(x) \Rightarrow \text{sister}(x, y)$

```
sibling(X,Y) << ( parent(P, X) & parent(P, Y) & neq(X,Y) )
```

```
brother(X,Y) << (sibling(X,Y) & male(X))
```

```
sister(X,Y) << (sibling(X,Y) & female(X))
```

neq(X,Y) (not-equal) is a *special belief* (ActiveBelief) that checks whether the arguments (X,Y) are different

Inside PHIDIAS

The expression:

```
sibling(X,Y) << ( parent(P, X) & parent(P, Y) & neq(X,Y) )
```

is interpreted by Python in the following way:

- Since **sibling**, **parent** and **neq** are Python classes, **parent(X,Y)** is the creation of an object instance of the relevant class
- Variables *X* and *Y* are defined by **def_vars(...)**, a function that creates, at runtime, objects of the type **Variable**, assigning the relative name so they can be used in the expressions
- By means of **operator overloading**, the use of **<< e &** runs the proper methods of base classes *Goal* and *Belief*, whose code creates, inside PHIDIAS engine, a proper data structure that represent the goal
- The PHIDIAS engine analyses this data structure and interpretes properly the expressions (like a Prolog predicate, in this case)

Behaviour Programming in PHIDIAS

A behaviour in PHIDIAS is implemented by means of **plans** that can be:

- **reactive** based on the paradigm **Event-Condition-Actions**
- **proactive** based on procedures constrained by a **pre-condition**

Reactive Programming with PHIDIAS

Reactive Plans

- **Reactive Plans** are specified by means of the following syntax:

event "/" condition ">>" "[" list of actions "]"

- The **event** can be

the assertion of a belief	+bel (...)
the retraction of a belief	-bel (...)

- The **condition** is a predicate specified on one or more beliefs present in the knowledge base and/or on the variables
- It can contain free variables (that are assigned according to the extracted belief) and can also use Goals previously defined
- The **list of actions** represents the “things to do” when the plan is executed; the list can contain one of the following statement (comma-separated):

assertion of a belief	+bel (...)
retraction of a belief	-bel (...)
procedure call	proc (...)
library/user-defined actions	go-to(X, Y)
Python expression	"X = X + 1"

A “rational” example: Students and Graduated

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

```
class student(Belief): pass
class graduated(Belief): pass

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

Let's analyse the example...

```
class student(Belief): pass
class graduated(Belief): pass

def_vars('X')
+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 the plans have the same **triggering event**: `+graduated(X)`
- These two plans represent a **group** because they have the same event
- But **only one plan** of a group can be executed
- The selection is made on the basis of the **writing order**:
the first plan that satisfy the condition is executed

A “rational” example: Students and Graduated

Let's add a further consistency rule to the knowledge:

- A degree cannot become a student again

```
class student(Belief): pass
class graduated(Belief): pass

def_vars('X')
+graduated(X) / student(X) >> [ -student(X),
    show_line("yeah ", X, " is now graduated!") ]
+graduated(X) >> [ show_line(X, " is not a student"),
    -graduated(X) ]
+student(X) / graduated(X) >> \
    [ show_line(X, " is graduated and cannot be a student again"),
    -student(X) ]
```

Reactive Programming: the Sieve of Erathostenes

Algorithm

- The algorithm of the “**Sieve of Erathostenes**” to find the prime numbers operates by considering all the numbers from **1** to **N**, and proceeds by “deleting” multiples
- The remaining numbers (not deleted) will be the prime numbers

Implementation in PHIDIAS

- Let's consider each number represented by the belief **number (X)**
- Let's use a rule that is activated on the basis of the event “assertion of a belief **number (X)**”
- The condition specifies the search for a number **Y** that is a sub-multiple of **X**: if it exists, the number **X** has to be removed from the knowledge base

Reactive Programming: the Sieve of Erathostenes

Implementation in PHIDIAS

- Let's consider each number represented by the belief **number (X)**
- Let's use a rule that is activated on the basis of the event "assertion of a belief **number (X)**"
- The condition specifies the search for a number *Y* that is a sub-multiple of *X*: if it exists, the number *X* has to be removed from the knowledge base

```
from phidias.Types import *
from phidias.Main import *
from phidias.Lib import *

class number(Belief): pass

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

# instantiate the engine
PHIDIAS.run()

# populate the KB (and run the rules)
for i in range(2,100):
    PHIDIAS.assert_belief(number(i))

# run the engine shell
PHIDIAS.shell(labels())
```


Reactive Programming: the Sieve of Erathostenes

Implementation in PHIDIAS and Execution Semantics

- The computational part of the algorithm is in the rule:

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

- When a number X is asserted, the condition implies to find **all the numbers Y sub-multiple of X**
- **For each number Y** found, an instance (**intention**) of possible execution of the plan is created
- However, only the **first instance** (intention) is executed

Reactive Programming: the Sieve of Erathostenes

Execution Example

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

- The belief **+number(6)** is asserted
- The system identifies the following intentions:

```
+number(6) / (number(2) & neq(6, 2) & (lambda: (6 % 2) == 0) )  
>> [ -number(6) ]
```

```
+number(6) / (number(3) & neq(6, 3) & (lambda: (6 % 3) == 0) )  
>> [ -number(6) ]
```

- The **first intention** is executed

```
+number(6) / (number(2) & neq(6, 2) & (lambda: (6 % 2) == 0) )  
>> [ -number(6) ]
```

Predicates and ActiveBeliefs

Lambda ad ActiveBeliefs

The algorithm of the Sieve of Erathostenes requires the presence of two conditions:

- the number extracted Y must be different than X
- The number extracted Y must be sub-multiple of X

These conditions can be expressed by means of **lambdas**:

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

Lambda and ActiveBeliefs

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

But there may cases in which:

- The condition to specify is more complex than a simple comparison and we would like to use an ad-hoc **boolean function**, or...
- (for example) we would like to bind a variable to data sampled by a sensor

In all of these cases, we can use an object of the type **ActiveBelief**:

- It **does not represent a knowledge** (it cannot be added to the KB)
- It implements a predicate/comparison in the method **evaluate** which returns a boolean

Lambda and ActiveBeliefs

```
class Multiple(ActiveBelief):  
    def evaluate(self, x, y):  
        return (x() % y()) == 0  
  
+number(X) / (number(Y) & (lambda : X != Y) & Multiple(X,Y) )  
>> [ -number(X) ]
```

- The example show an ActiveBelief **Multiple** which evaluates if the first argument is multiple of the second
- It is used in the condition of the plan
- In the method **evaluate**, the arguments are **special objects**, so the real variable values must be “extracted” by using **x()**

- The PHIDIAS library includes a set of ActiveBeliefs that implement classical comparisons:

<code>eq(X, Y)</code>	<code>X == Y</code>
<code>neq(X, Y)</code>	<code>X != Y</code>
<code>gt(X, Y)</code>	<code>X > Y</code>
<code>geq(X, Y)</code>	<code>X >= Y</code>
<code>lt(X, Y)</code>	<code>X < Y</code>
<code>leq(X, Y)</code>	<code>X <= Y</code>

- If we use only ActiveBeliefs, the Sieve of Erathostenes becomes:

```
class Multiple(ActiveBelief):
    def evaluate(self, x, y):
        return (x() % y()) == 0

+number(X) / (number(Y) & neq(X, Y) & Multiple(X,Y) )
>> [ -number(X) ]
```

Proactive Programming in PHIDIAS

Proactive Plans

- **Proactive plans** are specified with:

```
procedura "/" condizione ">>" "[" lista di azioni "]"
```

- The **procedure** specify the way in which the plan is invoked
- It is declared as a subclass of **Procedure** and can contain variables or ground terms
- The **condition** is a predicate on one or more beliefs present in the KB and/or on the variables
- It can contain free variables (that are assigned on the basis of the extracted belief) and also use Goals previously defined
- The **list of actions** represents the “things to do” when the plan is executed; the list can contain the following statements (comma-separated):

belief assertion	+bel (...)
belief retraction	-bel (...)
procedur call	proc (...)
library or user-defined action	go_to (X, Y)
Python expression	"X = X + 1"

“Hello World” in PHIDIAS

Implementation in PHIDIAS

```
from phidias.Types import *
from phidias.Main import *
from phidias.Lib import *

class say_hello(Procedure): pass

say_hello() >> [ show_line("Hello world from Phidias") ]

PHIDIAS.run()
PHIDIAS.shell(globals())
```

Conditional Constructs and Iteration

- PHIDIAS does not provide explicit conditional constructs (**if**), nor iteration (**for/while**)
- However, the use of conditions can overcome the problem

Example: factorial

```
from phidias.Types import *
from phidias.Main import *
from phidias.Lib import *

class fact(Procedure): pass

def_vars("Acc", "N")
fact(N) >> [ fact(N, 1) ]
fact(1, Acc) >> [ show_line("the resulting factorial is = ", Acc) ]
fact(N, Acc) >> \
    [
        "Acc = N * Acc",
        "N = N - 1",
        fact(N, Acc)
    ]

PHIDIAS.run()
PHIDIAS.shell(globals())
```

Conditional Constructs and Iteration

Example: computing the maximum

```
from phidias.Types import *
from phidias.Main import *
from phidias.Lib import *

import random

class number(Belief): pass

class compute_max(Procedure): pass

def_vars('X', 'TempMax')

compute_max() / number(X) >> [ compute_max(X) ]
compute_max(TempMax) / (number(X) & gt(X, TempMax)) >> [ compute_max(X) ]
compute_max(TempMax) >> [ show_line("The maximum is ", TempMax) ]

# populate the KB
for i in range(1,50):
    PHIDIAS.assert_belief(number(random.uniform(0,50)))

# instantiate the engine
PHIDIAS.run()

# run the engine shell
PHIDIAS.shell(globals())
```

Computing the Maximum

Knowledge Base

<code>number(5)</code>	<code>number(8)</code>	<code>number(2)</code>
<code>number(10)</code>	<code>number(3)</code>	<code>number(4)</code>

Execution example

```
compute_max() / number(X) >> [ compute_max(X) ]  
compute_max(TempMax) / (number(X) & gt(X, TempMax)) >> [ compute_max(X) ]  
compute_max(TempMax) >> [ show_line("The maximum is ", TempMax) ]
```

The procedure `compute_max()` is invoked, the runtime identifies the following intentions:

```
compute_max() / number(5) >> [ compute_max(5) ]  
compute_max() / number(8) >> [ compute_max(8) ]  
compute_max() / number(2) >> [ compute_max(2) ]  
compute_max() / number(10) >> [ compute_max(10) ]  
compute_max() / number(3) >> [ compute_max(3) ]  
compute_max() / number(4) >> [ compute_max(4) ]
```

Computing the Maximum

Knowledge Base

<code>number(5)</code>	<code>number(8)</code>	<code>number(2)</code>
<code>number(10)</code>	<code>number(3)</code>	<code>number(4)</code>

Execution example

```
compute_max() / number(X) >> [ compute_max(X) ]  
compute_max(TempMax) / (number(X) & gt(X, TempMax)) >> [ compute_max(X) ]  
compute_max(TempMax) >> [ show_line("The maximum is ", TempMax) ]
```

The **first intention** is executed:

```
compute_max() / number(5) >> [ compute_max(5) ]
```

Computing the Maximum

Knowledge Base

<code>number(5)</code>	<code>number(8)</code>	<code>number(2)</code>
<code>number(10)</code>	<code>number(3)</code>	<code>number(4)</code>

Execution example

```
compute_max() / number(X) >> [ compute_max(X) ]  
compute_max(TempMax) / (number(X) & gt(X, TempMax)) >> [ compute_max(X) ]  
compute_max(TempMax) >> [ show_line("The maximum is ", TempMax) ]
```

When procedure `compute_max(5)` is invoked the runtime identifies the following intentions:

```
compute_max(5) / (number(8) & gt(8, 5)) >> [ compute_max(8) ]  
compute_max(5) / (number(10) & gt(10, 5)) >> [ compute_max(10) ]
```

Computing the Maximum

Knowledge Base

<code>number(5)</code>	<code>number(8)</code>	<code>number(2)</code>
<code>number(10)</code>	<code>number(3)</code>	<code>number(4)</code>

Execution example

```
compute_max() / number(X) >> [ compute_max(X) ]  
compute_max(TempMax) / (number(X) & gt(X, TempMax)) >> [ compute_max(X) ]  
compute_max(TempMax) >> [ show_line("The maximum is ", TempMax) ]
```

The **first intention** is executed:

```
compute_max(5) / (number(8) & gt(8, 5)) >> [ compute_max(8) ]
```


Computing the Maximum

Knowledge Base

<code>number(5)</code>	<code>number(8)</code>	<code>number(2)</code>
<code>number(10)</code>	<code>number(3)</code>	<code>number(4)</code>

Execution example

```
compute_max() / number(X) >> [ compute_max(X) ]  
compute_max(TempMax) / (number(X) & gt(X, TempMax)) >> [ compute_max(X) ]  
compute_max(TempMax) >> [ show_line("The maximum is ", TempMax) ]
```

When procedure `compute_max(8)` is invoked the runtime identifies the following intention:

```
compute_max(8) / (number(10) & gt(10, 8)) >> [ compute_max(10) ]
```

Computing the Maximum

Knowledge Base

<code>number(5)</code>	<code>number(8)</code>	<code>number(2)</code>
<code>number(10)</code>	<code>number(3)</code>	<code>number(4)</code>

Execution example

```
compute_max() / number(X) >> [ compute_max(X) ]  
compute_max(TempMax) / (number(X) & gt(X, TempMax)) >> [ compute_max(X) ]  
compute_max(TempMax) >> [ show_line("The maximum is ", TempMax) ]
```

The intention is executed:

```
compute_max(8) / (number(10) & gt(10, 8)) >> [ compute_max(10) ]
```

Computing the Maximum

Knowledge Base

<code>number(5)</code>	<code>number(8)</code>	<code>number(2)</code>
<code>number(10)</code>	<code>number(3)</code>	<code>number(4)</code>

Execution Example

```
compute_max() / number(X) >> [ compute_max(X) ]  
compute_max(TempMax) / (number(X) & gt(X, TempMax)) >> [ compute_max(X) ]  
compute_max(TempMax) >> [ show_line("The maximum is ", TempMax) ]
```

When procedure `compute_max(10)` is invoked the runtime identifies the following intention and the program terminates:

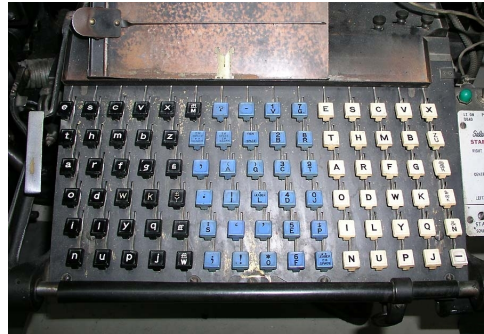
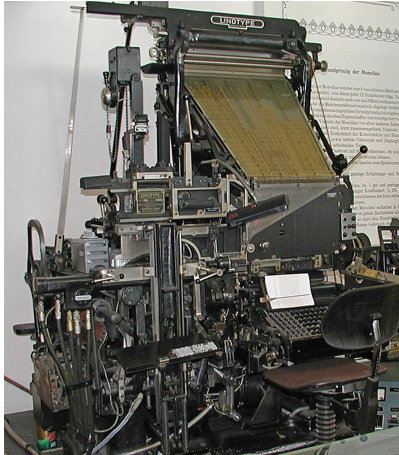
```
compute_max(10) >> [ show_line("The maximum is ", 10) ]
```

Case Study

SHRDLU: the Block World

- **SHRDLU** is one of the first AI programs (1968-1970), it supported the **reasoning/planning** and natural language processing
- SHRDLU “lives” in a virtual environment made of some “blocks” with different colors and shapes (cubes, pyramids, prisms, cylinders, ecc.)
- The program was able to understand commands such as “**Pick up a big red block**” or “**Grasp the pyramid**” and to behave accordingly
- The strange name **SHRDLU** comes from the sequence **ETAOIN SHRDLU** which is the key sequence of the “Linotype” (the sequenza was based on the frequency of the letters in the English language)

The Linotype and the sequence SHRDLU



SHRDLU and PHIDIAS

- Let's implement a small version of SHRDLU with PHIDIAS (file `"SHRDLU.py"`)
- Our world is composed of objects of the type: **cube**, **cylinder**, **prism**
- These objects are on a **table** and can be **captured** by a robot
- Objects can be **stacked**, one upon another, thus forming a **tower**
- An object can be captured only if it is **free**, i.e. it has no other object upon it

BELIEFS

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

PROCEDURES

- **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: capturing objects

- Object X is on the table, but another object Y is on X, then **X cannot be captured**

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

- If the previous condition is **false**, object is on the table or upon another one, in any case can be captured

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

SHRDLU plans: capturing objects

- If the previous condition is **false**, object is on the table and can be captured

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

- If the previous condition is **false** and object X is already owned by the robot, it cannot be captured again

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

- If all the previous conditions are **false**, the last plan will be executed:
this implies that object X does not exist

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

SHRDLU plans: releasing objects

Putting an object on the table

- If the object X is owned by the robot, then **we put X on the table**

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

- If the object X is on the table, then there is nothing to do

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

- If object X is not owned by robot nor placed on the table
then **the object does not exist**

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

SHRDLU plans: releasing objects

Putting an object X upon another object Y

- Object **X** is **owned** by the robot, and the object **Y** is on the **table**, but Y has another object Z upon it: **we cannot do the action**

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

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

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

Case Study “Change Coins”

“Change Coins”

- We want to automate the procedure of erogating the change by a food and beverage dispenser machine
- The machine has tanks for the coins:
 - 50 cent
 - 20 cent
 - 10 cent
 - 5 cent
- Given M the amount of money to change, the system will provide proper coins, always starting from those with the highest value and considering the possibiliy of empty tanks

“Change Coins”

- Let's use the following beliefs that indicate the number of coins in the relevant tanks:
 - **fifty(N)** number of 50 cent coins
 - **twenty(N)** number of 20 cent coins
 - **ten(N)** number of 10 cent coins
 - **five(N)** number of 5 cent coins
- Let's define a procedure **change(M)** that has the aim of providing the change coin by coin
- It will be a recursive procedure that identifies the coin with the highest value, provides the coin and updates the beliefs accordingly

“Change Coins”

The procedure `change (M)`

- Let's think in terms of “single coin value”
- Let C the number of coins of value t , then...
- ... if $M \geq t$ and $C > 0$, we can provide a coin of value t , decrement C , and recursively call procedure `change($M - t$)`:

```
Procedure Change(M);  
if ( $M \geq t$ )  $\wedge$  ( $C > 0$ ) then  
|    $C := C - 1$ ;  
|    $M := M - t$ ;  
|   Change(M);  
end
```


“Change Coins”

The procedure `change (M)`

```
# check 50 cent coins
change(M) / (fifty(C) & geq(M, 50) & gt(C, 0) ) >> \
[
    show_line("50 cent"),
    -fifty(C), "C = C - 1", +fifty(C),
    "M = M - 50", change(M)
]

# check 20 cent coins
change(M) / (twenty(C) & geq(M, 20) & gt(C, 0) ) >> \
[
    show_line("20 cent"),
    -twenty(C), "C = C - 1", +twenty(C),
    "M = M - 20", change(M)
]

...
```

"Change Coins"

The procedure `change (M)`

```
...  
# check 10 cent coins  
change(M) / (ten(C) & geq(M, 10) & gt(C, 0) ) >> \  
[  
    show_line("10 cent"),  
    -ten(C), "C = C - 1", +ten(C),  
    "M = M - 10", change(M)  
]  
  
# check 5 cent coins  
change(M) / (five(C) & geq(M, 5) & gt(C, 0) ) >> \  
[  
    show_line("5 cent"),  
    -five(C), "C = C - 1", +five(C),  
    "M = M - 5", change(M)  
]  
  
change(M) >> \  
[  
    show_line("End of change, remaning ", M, " cents")  
]
```

“Change Coins” and Singleton Beliefs

“Change Coins” and SingletonBeliefs

- The various plans tied to procedure “change” need to **update** the parameter of the belief that represents the value of the coin to provide
- To this aim, the piece of code used is:

```
...      -ten(C), "C = C - 1", +ten(C),  
...
```

- We **remove** belief, change the parameter and **assert again** the same belief
- This is needed because the parameter of a belief cannot be changed

“Change Coins” and SingletonBeliefs

- However, the beliefs:

- `fifty(N)`
- `twenty(N)`
- `ten(N)`
- `five(N)`

are characterised by the fact that they can exist in **single instance**

- Indeed from the application point of view, having two beliefs `five` with different parameters **does not make sense**
- In these cases, the beliefs that can exist in single instance can be declared as **SingletonBelief**
- Indeed, the operation “add” (+) on a belief of this type provokes the update of the single instance (if it is already present)

“Change Coins”

change (M) with SingletonBelief

```
class fifty(SingletonBelief): pass
class twenty(SingletonBelief): pass
class ten(SingletonBelief): pass
class five(SingletonBelief): pass

# check 50 cent coins
change(M) / (fifty(C) & geq(M, 50) & gt(C, 0) ) >> \
    [
        show_line("50 cent"),
        "C = C - 1", +fifty(C),
        "M = M - 50", change(M)
    ]

# check 20 cent coins
change(M) / (twenty(C) & geq(M, 20) & gt(C, 0) ) >> \
    [
        show_line("20 cent"),
        "C = C - 1", +twenty(C),
        "M = M - 20", change(M)
    ]

...
```

Declarative Programming in Python using PHIDIAS

Corrado Santoro

ARSLAB - Autonomous and Robotic Systems Laboratory

Dipartimento di Matematica e Informatica - Università di Catania, Italy

`santoro@dmi.unict.it`



Robotic Systems