

# Framework for a Smart Environment

## For AI Project

Gaetano Dibenedetto<sup>a</sup>, Mauro Andretta<sup>a</sup>

<sup>a</sup>*Dipartimento di Informatica, Università degli Studi di Bari Aldo Moro, Via E. Orabona, 4 - Bari 70125*

September, 2022

---

### Abstract

Smart environments aim at improving our daily lives by automatically tuning ambient parameters such as: temperature, brightness and noise. In this way it is possible to accomplish the user preferences and achieving energy savings through self-managing physical systems. In this article, we propose a framework to represent smart environments, where the user sets his own preferences. A Prolog prototype of the framework is showed over a person's bedroom.

---

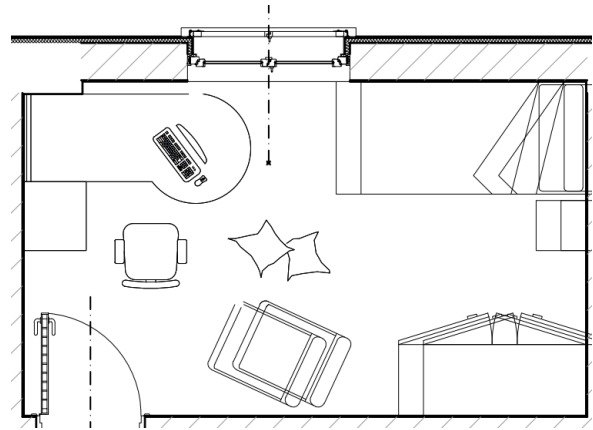
### 1. Introduction

Smart environments are getting increasing attention from the market and the research community, because they are able to simplify and improve the quality of a person's daily life routine. Indeed, nowadays several commercial solutions exist on the market, such as Amazon Alexa, that only allow to set simple goals that can just modify devices settings without considering additional important parameters that could be relevant. The improvement of the daily life routine isn't just the only advantage, in fact the smart environments are also able to improve the energy management and consumption in the whole house. With our project we initially focus on the management of a bedroom, thanks to sensors used to check some environmental factors, such as temperature and brightness, that could be useful for improving and making the daily life more conformable, and actuators that modify the value of the devices present in the bedroom to achieve the desired preferences specified by the user.

### 2. Method and Equipment

Before starting the project, we took a model of a bedroom to carry out our reasoning on how to implement a smart environment in it. We focused on what could be the relevant sensors and actuators to be inserted, in order to have complete control over all the activities carried out in a bedroom, to be able to optimize them based on the user's needs.

For the insertion of sensors and actuators, all the objects present in the room taken as an example have been considered (Fig. 1), for this room we've identified the following elements that could be relevant for us:



**Figure 1:** Bed room example

- light desk
- main light
- corner light
- ac
- window

In the room, sensors were considered to control specific environmental factors such as temperature, brightness and noise, the sensors can be internal, external or both. For each environmental factor considered, we have tried to subdivide the elements of the room based on what environmental factor they would have directly modified:

- light desk - brightness
- main light - brightness
- corner light - brightness

- ac - temperature
- window - temperature
- roller shutter - brightness

Considering that some elements such as the window could be used to manage more environmental factors such as temperature and brightness, we have tried to subdivide the item window in:

- *window* for the brightness factor
- *rollershutter* for the temperature factor

### 2.1. Model

To model the smart environment, we started with the creation of our knowledge base using the Prolog programming language, in specific Swi-Prolog. We first build up a list of all types of environmental parameters. We call property types the elements in such a dictionary, assuming they are declared as in:

```
propertyType(TypeId).
```

where TypeId is a value denoting the unique environmental factor identifier. Given a propertyType we can then define actuators and sensors that operate on that. Actuators are declared as in:

```
actuator(ActuatorId, TypeId).
```

where ActuatorId is the unique actuator identifier and TypeId the associated environmental factor. In our reasoning there isn't a specific actuator acting on the noise, so, we supposed that to work on the noise environmental factor, we should work on the temperature actuators. This idea is given to the fact that if we have a noise coming from outside, we should close the window (which is an actuator for the temperature), and this could cause a variation of the inside desired temperature, so we act on the air conditioning system in order to restore the expected temperature.

So the noise actuators are defined as following:

```
actuator(X, noise) :- actuator(X, temp).
```

We can change the value of the ActuatorId using as in:

```
actuatorValue(ActuatorId, Value).
```

Analogously, sensors are declared as in:

```
sensor(SensorId, TypeId).
```

where SensorId is the unique sensor identifier and TypeId is the associated environmental factor. Environmental values monitored by each sensor are denoted by:

```
sensorValue(SensorId, Value).
```

where SensorId identifies the sensor and Value is the last value read by it.

**Example:** Considering the above descriptions, we showed here a part of the knowledge base:

```
propertyType(light).
propertyType(temp).

sensor(brightness, light).
sensor(temperature, temp).

sensorValue(brightness, 20).
sensorValue(temperature, 22).

actuator(mainLight, light).
actuator(ac, temp).

actuatorValue(mainLight, 0).
actuatorValue(ac, 0).
```

where two environmental factors are specified: light and temp. These are monitored by two sensors respectively brightness and temperature. We can notice that the actual temperature in the bedroom is 22°C and the brightness is 20 over a max of 100 (hypothetical max value that the sensor can reach). And the two considered actuators, the air conditioning system (ac) and mainLight, are turned off.

As we said, it is possible to have sensors and actuators that operate considering the outside environmental factors and the inside ones. To distinguish them we create the following properties.

```
inside(Id).
outside(Id) :- \+ inside(Id).
```

where Id represents the sensor or the actuator identifier, if a sensor or an actuator is related to the inside, than it will be inserted as in the following way in the knowledge base:

```
inside(ac).
```

Considering this model, the user is allowed in the creation of its own preferences. They are declared as in:

```
preferencesInstances(PiiD, TypeId, ExpectedValueSensor, Actuators).
```

where PiiD represents the identifier of the preference, ExpectedValueSensor is the value wanted by the user for the specific environmental factor (TypeId), and Actuators represents a list of actuators that can act for achieving the target value.

**Example:**

```
preferencesInstance(study, light, 20,
[light_desk, mainLight, roller_shutter]).

preferencesInstance(study, temp, 24,
[ac, window]).

preferencesInstance(study, noise, 10,
[ac, window]).
```

Once the user creates a preference, it can be applied to the environment with the following declarations:

**Example:**

```
set(PiiD, _).
set(PiiD, TypeId).
```

For each type of TypeId there are rules described by the knowledge base, like for the temperature preference: to reach the goal the intelligent system try to understand as first if the temperature inside the room is less then the desired temperature, if this is the case the sensor for the detection of the temperature outside is activated, if the temperature outside is greater than the desired temperature the window will be opened to reach the goal, otherwise the system activates the ac. Vice versa, if the temperature inside is greater than the desired temperature, the system works in the opposite way: it checks the temperature outside, if it is greater than the goal temperature, we can't open the window and the ac will be activated, otherwise the system will open the window.

In the Prolog code below we show a partial reasoning, in case we have an inside temperature lower than the desired one, and at the same time this is lower than the outside temperature.

**Example:**

```
set(PiiD, temp) :-
    preferencesInstance(PiiD, temp, Y, Actuators),
    sensor(SensorId_outside, temp),
    outside(SensorId_outside),
    sensorValue(SensorId_outside, X_outside),
    sensor(SensorId_inside, temp),
    inside(SensorId_inside),
    sensorValue(SensorId_inside, X_inside),
    X_inside < Y,
    X_outside > Y,
    setOutsideActuators(Actuators, Y),
    setInsideActuators(Actuators, 0).
```

As we can see, we get the value of the desired temperature from the knowledge base from the user's preference as Y, we compared it with the value of the sensor related to the inside temperature ( $X_{inside}$ ) and the outside one ( $X_{outside}$ ), if this is at the same time higher than the inside temperature and lower than the outside one, we act on the actuators (*Actuators*) given from the user's preference, extracting the outside actuators, setting them at the desired temperature, and at the same time all the inside actuators are turned off.

It is possible to notice that in the previous example we have two rules that are not already showed. They are declared as in:

```
%setInsideActuators(Actuators, Value).
setInsideActuators([H|T], Y) :-
    extractInsideActuators([H|T], [], L),
    setActuators(L, Y).

%setOutsideActuators(Actuators, Value).
setOutsideActuators([H|T], Y) :-
    extractOutsideActuators([H|T], [], L),
    setActuators(L, Y).
```

where we use a list of Actuators and the desired value, and for each element in the list of actuators we check whether it is an inside or outside one, and based on this property the system acts changing the value of the actuators.

For the extracting phase we show the rules below:

```
%extractInsideActuators(List, NewList,
                        variable).

extractInsideActuators([H|T], L,X) :-
    T \== [],
    inside(H),
    !,
    extractInsideActuators(T, [H|L], X).

extractInsideActuators([H|T], L, X) :-
    T\== [],
    \+ inside(H),
    !,
    extractInsideActuators(T, L, X).

extractInsideActuators([H|_], L,X) :-
    \+ inside(H),
    !,
    X = L.

extractInsideActuators([H|_], L, X) :-
    inside(H),
    !,
    X = [H|L].

extractInsideActuators(_, L, X) :-
    X = L.
```

we show the reasoning only for the inside extraction, because the outside one is similar, we have just changed the property *inside(H)* with *outside(H)*. The first step that we check is the case in which the list contains more than one element, in this case we check if the first element of the list is an inside actuator and if this is true then we add this first element to a new list which represents all the inside actuators found. Then we call the function recursively. As last step of the recursion, we basically have the case in which the list of actuators to check has only one element, then we do the same check (*inside(H)*) and if this is true we add this element to the list which represents all the inside actuators funded. We stop the recursion returning the list of all the inside actuators. In case the list of actuators to be checked is empty, we return an empty list.

It is relevant to notice the fact that for the study preferences showed previously, the preference for the noise in the environment has been set to 10 (expected decibel). This value is strictly related to the desired value for the temperature in the study preference. Indeed, if the temperature

outside can be used to reach the desired temperature inside, the window will be opened, but if there is too much noise outside, the window is closed and the air conditioning system is turned on.

## 2.2. Inference Engine

An inference engine makes a decision from the facts and rules contained in the knowledge base. For our project the inference engine used is Prolog, even though we are aware that it can not be considered a complete inference engine, because it should use all kinds of inferences, for reaching a goal. Instead, without changing its formalism, Prolog only uses the backward strategy, useful when we know our goal, that is what happens in our case, so our inference engine:

- starts from the goal
- applies all the rules backward, until it reaches a fact and so the goal can be proven

The Prolog system could be completed as inference engine with the forward strategy, that could be applied when there isn't an explicit representation of the goal. In our case this is not useful because we always have an explicit representation of the goal.

## 2.3. Storing data

Considering that the inference engine, for each execution, consults the knowledge base, some changes made by the user could be lost at each restart. To solve this problem we provided a method to store the information in the working memory in a log file, that will be added to the knowledge base when the system is restarted.

## 2.4. Linguistic modules

It contains all the answers and questions in natural language used to make the interaction more user-friendly. For our project, the user is allowed to ask (using the button *Why?* in the main window of the GUI) to the system why a selected actuator has a particular value. The system reasons about the actions made by the user and it answers using natural language. To apply the reasoning, the system uses the logs (Section. 2.3) in this way: first we check if the actuator's value has been modified manually by the user or by a preference, if one of these two conditions is satisfied, an answer about how the actuator's value has been changed will be showed (Section. 2.5.8). If these two conditions are not satisfied, then it means that the actuator has never been modified from its initialization.

## 2.5. User Interface

To make the interaction simpler for a common user, we created an user interface, that communicates with our inference engine with the following python library: *"pyswip"*. Instead the user interface has been created using a fork of the *"tkinter"* library, called *"CustomTkinter"*. The GUI is composed by several windows, that allow the user to:

1. Visualize the smart-environment's info
2. Select a preference
3. Add a preference
4. Add an actuator
5. Modify an actuator value
6. Add a sensor
7. Remove an instance
8. Understand the actuator's value

### 2.5.1. Visualize the smart-environment's info

To acquire the data from the knowledge base, some functions have been defined, like `getAllSensor()` that allows to acquire all the sensors value with a simple query in Prolog and `getAllActuator()` that allows to acquire all the actuators value. The (Fig. 2) shows the main window in the user interface, with a *nullPreference* set.

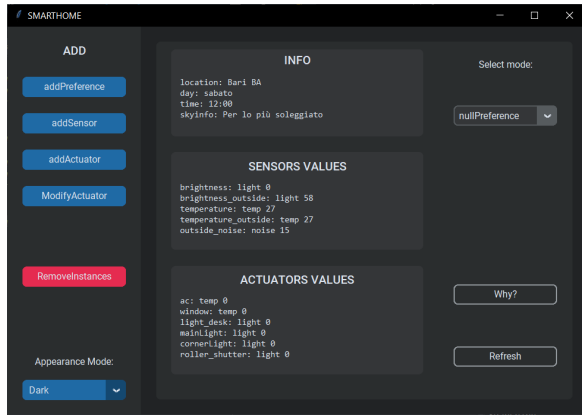


Figure 2: Gui main window

### 2.5.2. Select a preference

In the right side of the main window, we have a combo box that shows all the available preferences, giving the user the possibility to activate a determinate preference. The (Fig. 3) shows the main window in the user interface, with a *study* preference set, and as we can see, the values of the actuators have been changed to satisfy the user preferences.

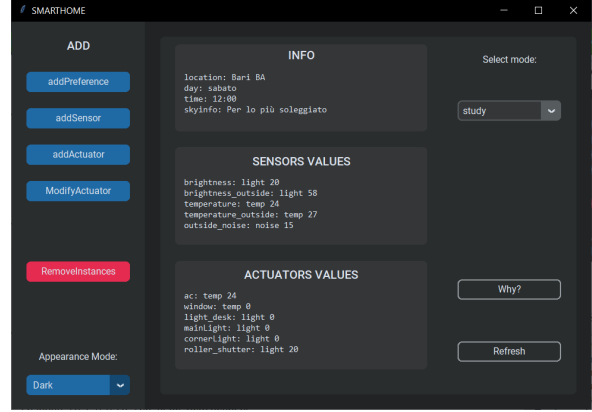


Figure 3: Gui main window with *study* preference

### 2.5.3. Add a preference

To add a new preference we must choose:

- name: it will be the identifier for the preference
- property type: it will unlock only the actuators related to the property
- actuators: we can select one or more actuators on which we apply the preference
- desired value: the desired value for the preference

In this window we only have one constraint, we can't use an actuator more than once on the same preference name, but we can use the same preference name that acts on different actuators.

The (Fig. 4) shows the window for the insertion of a new preference. .

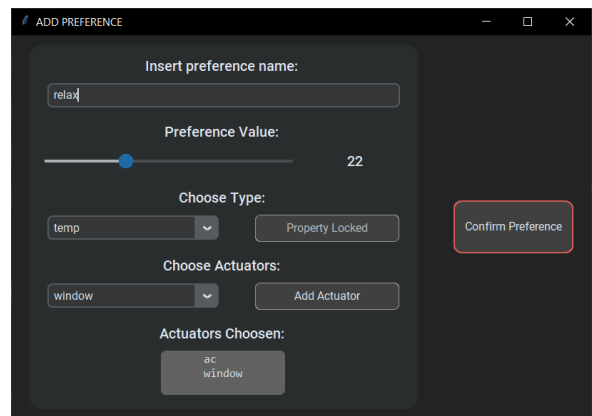


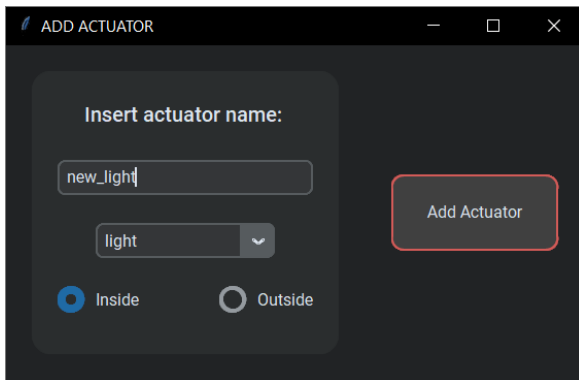
Figure 4: Window to add a new preference

#### 2.5.4. Add an actuator

To add a new actuator the user must choose:

- name: it will be the identifier for the actuator
- property type: based on the environmental factor that it will modify
- inside or outside actuator

Below it is showed how to add an actuator (Fig. 5), and the only constraint is that is not possible to create more than one actuator with the same name.



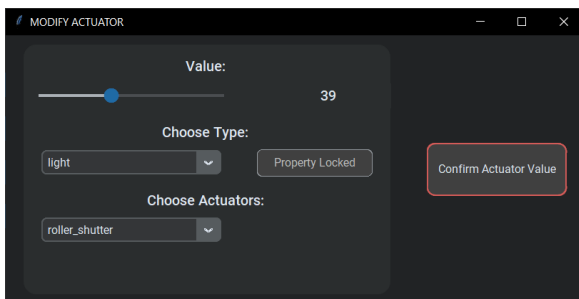
**Figure 5:** add actuator window

#### 2.5.5. Modify an actuator value

To modify an actuator value, the user must choose:

- value: the value to apply to the actuator
- property type: the property to filter the list of actuators
- actuator: the actuator on which the desired value will be applied

Below is showed how to modify an actuator's value (Fig. 6).



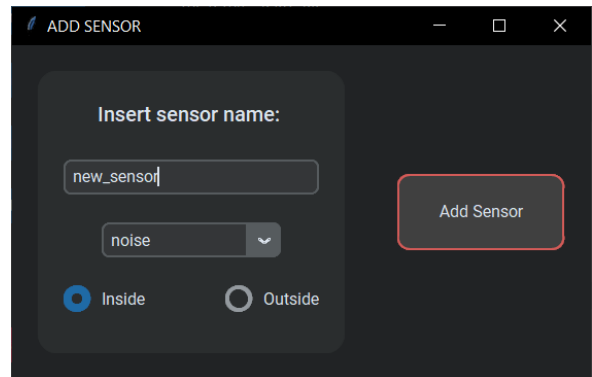
**Figure 6:** modify actuator window

#### 2.5.6. Add a sensor

To add a new sensor the user must choose:

- name: it will be the identifier for the sensor
- property type: based on the environmental factor related
- inside or outside sensor

Below it is showed how to add an sensor (Fig. 7), the only constraint is that is not possible to create more than one sensor with the same name.



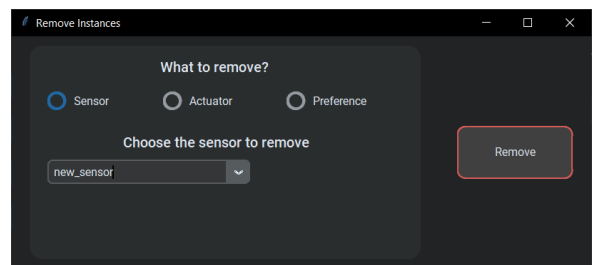
**Figure 7:** add sensor window

#### 2.5.7. Removing instances

To remove an instance, the user must choose:

- the kind of instance: sensor, actuator or preference
- instance's name: choose the name from the combo box

Below it is showed how to remove a sensor (Fig. 8).



**Figure 8:** remove instance window

### 2.5.8. Why actuator's value

To receive an answer about the actuator's value, the user must choose:

- property type: the property to filter the list of actuators
- actuator: the actuator on which the explanation will be given

Below are showed two examples: one where the user has modified manually the value of an actuator (Fig. 9) and the other where the actuator's value has been changed by a preference set by the user (Fig. 10).

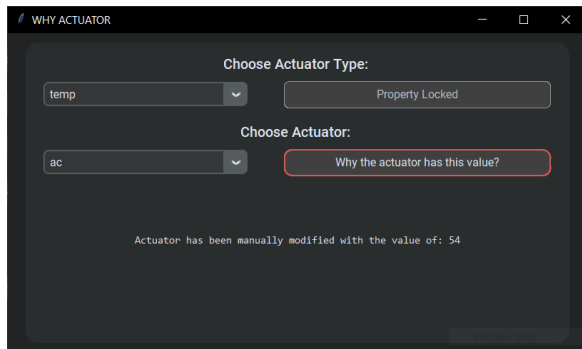


Figure 9: why of an actuator modified manually

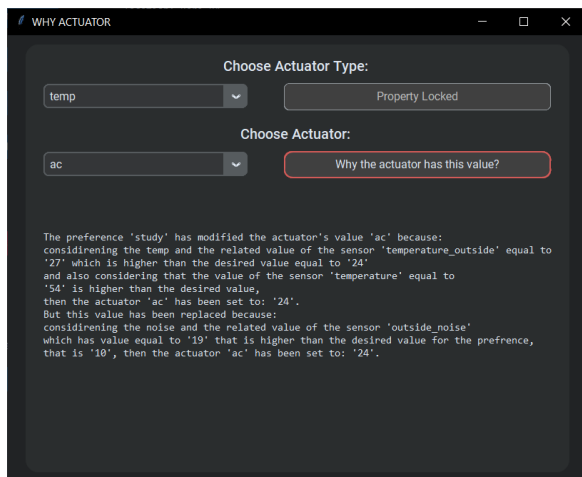


Figure 10: why of an actuator modified by a preference

- The value of the **brightness\_outside** is generated considering the **sky info** of the given city in input, and the **hour**.
- For the **noise** we consider the **wind force**, and we calculate with random values the decibel that it could generate.
- For the **temperature\_inside** we added a random value in the range of -10 and +10 to the value of the **temperature\_outside**.

To simulate the changing of the environmental factors due to the actuators modification, we've considered that an internal sensors could change its value by the activation/deactivation of one or more actuators, so we created a function to simulate the environmental changes. It sets the sensors value related to an environmental factor, based on the highest value of all the actuators related to the same environmental factor. In case of the temperature, if all is turned off, it will get a value of a temperature related to the temperature\_outside as explained previously.

## 2.6. Virtual Sensors Value

To start with a virtual environment to test the model, we have created some functions that considering a city given in input, they give us an initial value for the sensors of our smart environment. For example: