University of Genoa

Software Architecture for Robotics project

# Browsing Robot Beliefs

*Bianchi Igor Pio, Comotti Michele*

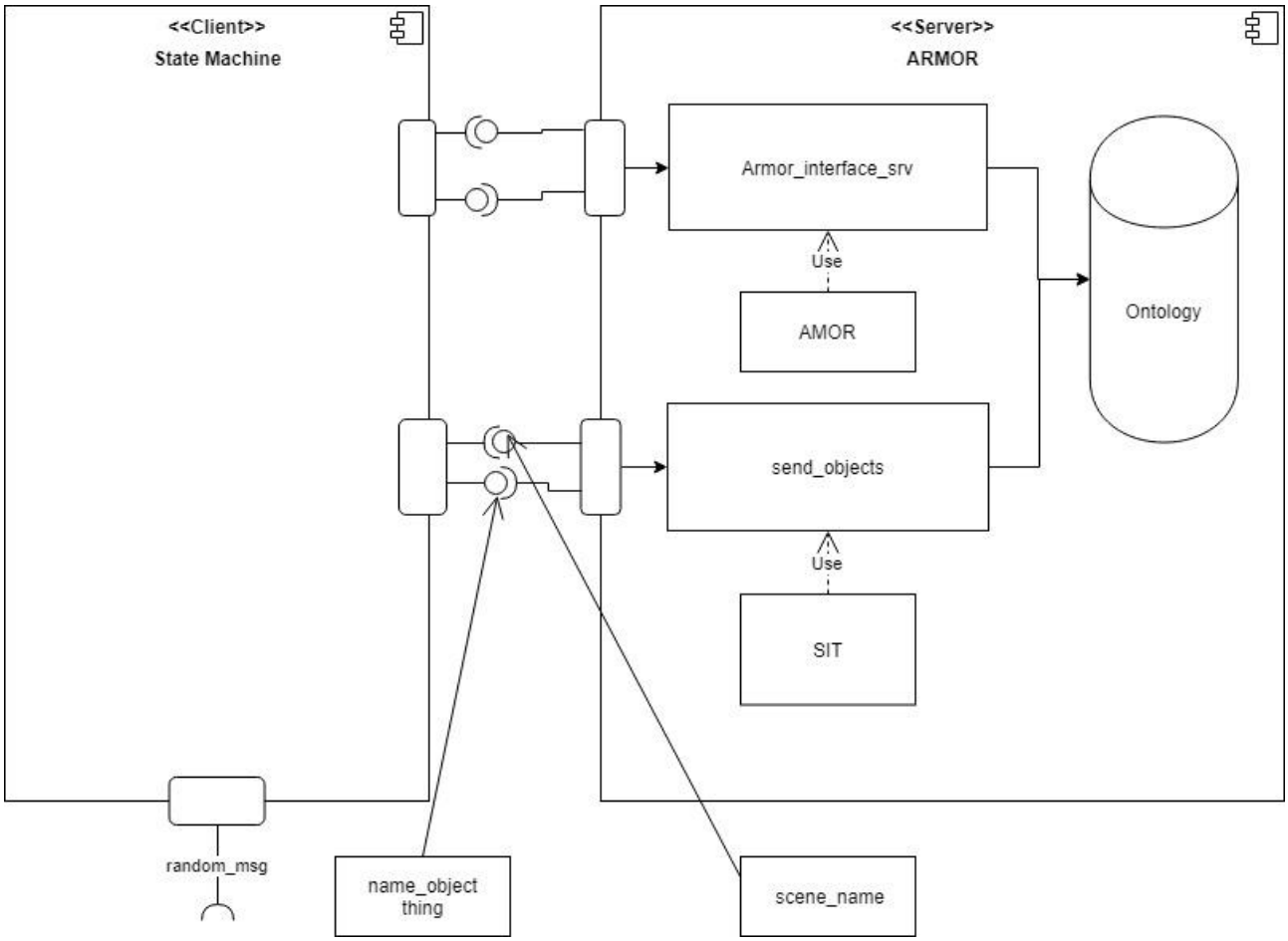
# Objective

The goal of this project is to simulate a dialogue between a robot and a human based on salient characteristics of a fixed environment. We simulate the acquisition of a table-top scenario involving some objects, then the robot should be able to learn the purposed scenario and share its knowledge through dialogues simulated on command prompt.

# Accomplishment

- Creation of ROS node to load ontology file into AMOR;
- Creation of ROS node to activate SIT using the ontology previous loaded;
- Creation of messages and services to simulate objects features and sending them to ARMOR;
- Extension of ARMOR package with a new JAVA class to connect it with SIT;
- Creation of ROS node to simulate the acquisition of geometric data objects from PIT;
- Creation of ROS Smach state machine to implement a simulation of dialogue between human and robot by integrating the previous points.

# Software architecture



Before dealing with software architecture, we want to specify that state machine is explained after the discussion about every single node that state machine is composite.

## ARMOR

ARMOR is a powerful and versatile management system for single and multi-ontology architectures under ROS. It allows to load, query and modify multiple ontologies and requires very little knowledge of OWL APIs and Java. Despite its ease of use, ARMOR provides a large share of OWL APIs functions and capabilities in a simple server-client architecture and offers increased flexibility compared to previous ontological systems running under ROS.
The ARMOR service is used by all clients implemented by us. ARMOR package has been modified to connect this module to SIT, so in this way an external ROS node (client) could inject information to SIT which is responsible to recognize compositions of objects based on their qualitative spatial relations.
SIT needs to have an OWL ontology file which contains the knowledges of robot so that it could work in the right way.

## Managing Ontology

### Load Ontology Amor Node

This node is created in ROS Python that uses the pre-existent ARMOR service called "armor_interface_srv" which has ROS messages as request and response parameter.

This node sends the ontology file to AMOR by given the predefined command "LOAD" and the path to the ontology file as parameter of the service request.

### Init Sit Node

This node uses the service mentioned before, with a different configuration in the request.
In this case, a new command "INIT" is defined and it becomes parameter in the request of the service.
This command is added to ARMOR command list.
This action calls a method, which is also defined by ARMOR, that is responsible for activating SIT by passing the ontology file, previously loaded in AMOR.

### Clean Ontology

A ROS python client, which is responsible to eliminate all individuals contained in the ontology, is created. This operation is required to learn a new scenario. This client took advantage of "armor_interface_srv" by defining a new command as well as some clients defined before.

## Simulating PIT

### New Messages and Services

New ROS messages and ROS services are defined to simulate the interaction between PIT and ARMOR. Several messages are defined to represent objects with their main geometrical features. For example, a message called "Sphere.msg" is created which contains its main features, i.e. radius and center coordinates. The center itself is a message containing its coordinates.
The same pattern is used to define other objects.
These messages are used by a new ROS service called "send_object" of type "ArmorObject.srv". This service contains a list of strings that represent the names of objects called "name_object" and a

list of objects called “thing” sent by client as parameters of the request.
A list of strings, that contains the names of all recognized scenes called “scene_name”, is returned as response.

**Client Send Object Node**

A new ROS node in python called “ClientSendObject“is implemented and it simulates the functionality of PIT by defining some fixed table-top scenario with two or three objects thanks to messages and ArmorObjects service mentioned before.

All these informations are sent to SIT for the recognition of scene, but a connection between ARMOR and SIT didn’t exist. A new java class called ArmorLinkSit is created to this purpose.

**Armor Link SIT**

This class works as bridge between ARMOR and SIT. It checks if information given by ClientSendObject node exist. In positive case, these informations are sent to SIT which starts the recognition and reasoning process.

**Sit**

SIT is a library that implements the Scene Identification and Tagging (SIT) algorithm which is a semantic algorithm to learn and recognize composition of objects based on their qualitative spatial relations**.**
Before our changes, this module was a sort of main class with a connection with ARMOR. In fact, there were defined objects features and the ontology to execute learning and recognition by robot The connection “strong” is build and the clients defined before can to load the ontology and send objects features, then SIT starts learning and recognizing the purposed scenario.

Learning and recognition were done through an ontology manipulation, for example when the robot learned a new scene, this module will add new individuals to the ontology.

All the information managed by the SIT will not be saved directly in the ontology, but they are saved in a sort of ram memory of the robot.

## DIALOGUE SIMULATION

After the robot learned the purposed scene, the dialogue part, which is the aim of this project, is implemented.
As said before this dialogue is a simulation, we couldn’t have a direct speaking dialogue with the robot, but the interaction is implemented through command prompt.

We supposed a scenario like these, in which there are three fixed scenes:

**Fig. Scene 1:** Sphere and Plane

**Fig. Scene 2:** Two Spheres

**Fig. Scene 3:** One Plane and Two Spheres

The aim is to interrogate the robot about what it has seen. Three kind of questions that the human supervisor could ask to the robot are defined:

Question 1
H: What about this **scene**? Scene 1 is considered

```
$$$  Plane is Along X With   Sphere
$$$   Sphere is Along X With   Plane
$$$   Sphere is Right Of  Plane
```

Question 2
H: Do you see a **sphere** (or plane)? Scene 1 is considered

```
$$$ I see a Sphere
```

Question 3
H: Which object is **right** (left,behind,above,front,along) of **sphere** (plane)? Scene 1 is considered

```
$$$  Sphere is Right Of  Plane
```

The word written in bold are the core of dialogue. These words are passed as keyword of a Query Sparql. ARMOR already contains the command to execute a Sparql Query on the ontology of robot. It exploits again the "armor_interface_srv" configuration by passing as parameter the required command and the sparql query written in the right way. For example, the Query Sparql of Question 2 is:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> PREFIX owl:
<http://www.w3.org/2002/07/owl#> PREFIX xsd: <http://www.w3.org/2001/XMLSchema#> PREFIX rdfs:
<http://www.w3.org/2000/01/rdf-schema#> PREFIX sit: <http://www.semanticweb.org/emaroLab/luca-
buoncompagni/sit#> SELECT ?p ?cls WHERE { sit:TestScene
(owl:equivalentClass|(owl:intersectionOf/rdf:rest*/rdf:first))* ?restriction .

  ?restriction owl:onProperty ?p . FILTER(regex(str(?p),"Sphere",\"i\"))} UNION {  ?restriction
  owl:onClass ?cls . FILTER (regex(str(?cls),"Sphere",\"i\"))}}  LIMIT 1']
```

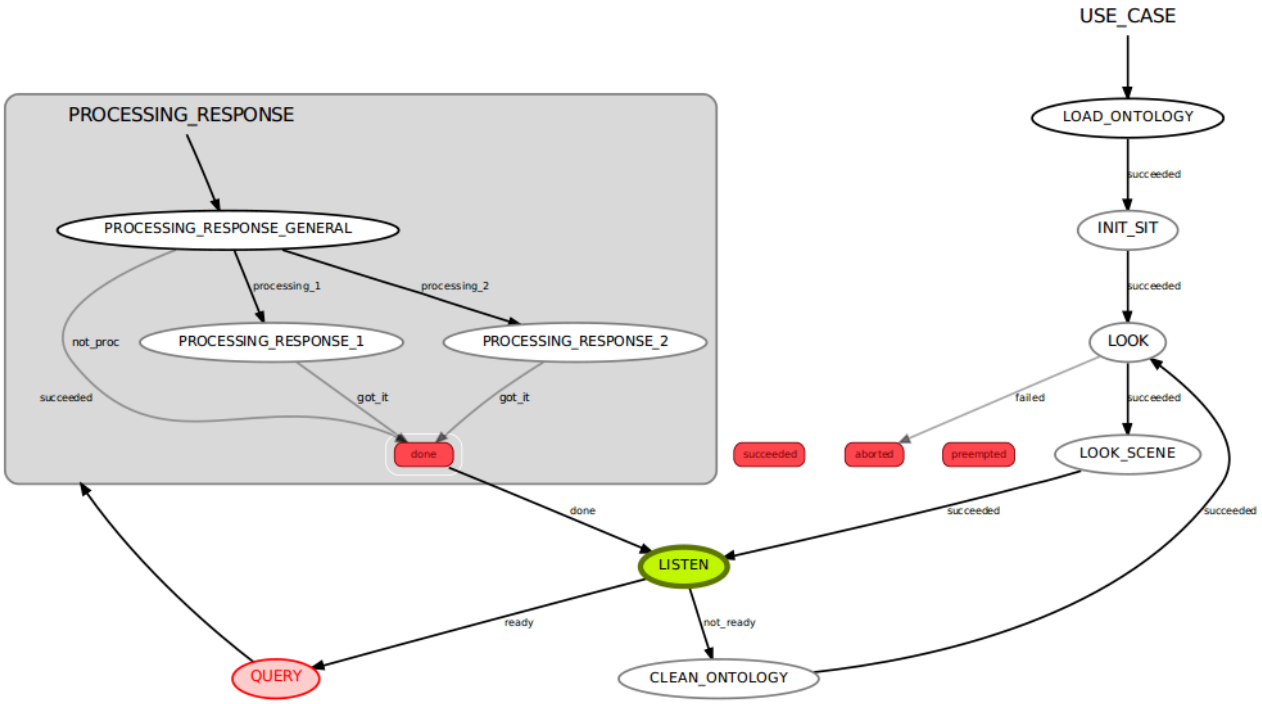
The response of this service is a list contained the information required or an empty list in case of the query fails, or it is asked something that the robot didn't see.

Then the response is processed to obtain a reasonable result.

The dialogue is implemented in the state machine.

## STATE MACHINE

A Ros Smack state machine is created to give a complete demonstration of all functionalities implemented by us. This state machine shows a complete simulation of the robot behaviour focusing on what we have described before.




- **LOAD ONTOLOGY:** this state recreates the same functionality defined in "Load Ontology Amor".

- **INIT SIT:** this state recreates the operations defined in "Init Sit", it sends the ontology and start the SIT

- **LOOK:** A publisher and subscribe pattern are used to simulate the vision of the robot and the PIT module. A ROS publisher node is created to send a random number from 1 to 3 to the state machine which take the role of a subscriber. Each number corresponds respectively to the scenes defined in the dialogue simulation (number 1 corresponds to scene 1). If the state doesn't receive a number within 30 seconds, the execution of the entire machine stops, and it is necessary to restart the state machine.

- **LOOK SCENE:** In this state a different scenario is obtained based on the random number received in the state before. Look and Look scene simulate the behaviour described by ClientSendObject node.

- **LISTEN:** The simulation of the dialogue is initialized between human and a robot. The robot asks human if he wants to ask something about what it has seen. This question is a print on the command prompt and human response is given by a keyboard input. If the reply is “yes”, the process advances on the Listen state otherwise if any other input is given, the process advances to Clean state.

- **CLEAN STATE:** it recreates the behaviour defined by Clean Ontology client. Then the execution returns to Look state.

- **QUERY:** It is defined what the robot needs to do to reply human questions. Human inserts one of the questions defined in the Dialogue paragraph on command prompt. The robot takes the keywords from the question and it builds a specific Sparql Query. If supervisor inserts a question that doesn’t contain any keyword, the robot replies to the human and repeat this state.

- **PROCESSING RESPONSE:** It is the initial state of a state submachine which processes the response obtained from sparql query

    - **PROCESSING RESPONSE GENERAL:** It is a sort of switch state in which depending on the keyword of question and response given by the query,the process could move to proceesing_respone_1 or processing_response_2

    - **PROCESSING RESPONSE 1:** It processes the response given by question of type 1 and type 3. This response is showed as a print on the command prompt. Then the execution returns to Listen state.

    - **PROCESSING RESPONSE 2:** it processes the response given by question of type 2. This response is showed as a print on the command prompt. The execution returns to Listen state.

## LIMITATION OF THE SYSTEM

In this paragraph we report the limitations of the system followed by possible suggestion to solve them (reported in corsive)

- When SIT manipulates the ontology, changes stayed in a ram memory of the robot and they weren’t saved in a file, so the robot doesn’t remember the scene seen before (è vero? O non vede solo gli individui?). *To solve this, it could be possible to call a command name “SAVE” with correct parameters defined in the ARMOR PACKAGE*

- The simulation of vision is limited. The client send_scene has only three fixed scenes with predefined object limited in shape and number (max 3 object of type sphere or plane). The vision can’t also distinguish same object, for example two sphere and this behavior affects the response given by robot. If response is like “Sphere is above of Sphere”, it can’t distinguish which sphere is above of the other. *To solve the problem, it could be possible to create new object by adding new messages with the same structure as Sphere.msg and*

*Plane.msg and extend them with an attribute color to distinguish same object. Then the client send_object should be modified to send a new scene. It is necessary to change the code in SIT which works on the objects features given by the vision.*

- In the current state machine, the robot looks a scene and it asks human if he wants to do some question about what it has seen. We thought it should be clearer if the human starts the dialogue by telling robot to start looking a scenario and then tell it to stop looking and start listening to him. *It's possible do that by changing the transition of the state machine or adding new states.*

- The dialogue is limited to 3 kind of questions and replies defined above. *New keyword and sparql queries have to be defined. This should be made in the state machine by extending states query and processing_response.*

- Queries are always created to the last configuration visualized by the robot, so a human could not select to query a scene seen before. We decided to query the scene with the highest cardinality but it's possible to do that in different ways. The robot can't give a reply which explains relation between scenes visualized. Robot has no memory of what it sees.

- User can't ask robot a question with opposite keyword respect to knowledge of robot. For example, if there is a configuration like this: "Plane is right of Sphere", robot doesn't recognize "Sphere is left of Plane". *Manipulation of ontology is necessary by adding inversion property.*

- Publisher, which simulates PIT, sends the same number of scene five times.

- When execution of state machine arrives in state "Listen", if human doesn't want to ask something to robot, human has to relaunch publisher node, otherwise after 30 seconds, state machine aborted.

- All new functionalities are commented but documentation is not generated. *Documentation needs to be generated.*

## FUTURE DEVELOPMENTS

- The dialogue implemented in the state machine is a prototype of a real dialogue. What we made could be used as example to implement a real dialogue with the robot using a Speech to text software.
- The process of the robot's replies works on strings given by the result of the sparql query. In a real case it should be useful to define a specific grammar that the robot could use to reply to human.

## HOW TO RUN

### Prerequisites

- Ros Kinetic (see http://wiki.ros.org/)

- Rosjava (see: http://wiki.ros.org/rosjava)
- Smach (see: http://wiki.ros.org/smach)
- RosPy (see: http://wiki.ros.org/rospy)
- Injected_armor_pkgs (see: https://github.com/TheGor/injected_armor_pkgs)
    - Armor_py_client_api (see: https://github.com/TheGor/injected_armor_pkgs/tree/developingMine/armor_py_client_api)
    - Injected_armor (see: https://github.com/TheGor/injected_armor_pkgs/tree/developingMine/injected_armor)
        - Armor (see: https://github.com/TheGor/injected_armor_pkgs/tree/developingMine/injected_armor/armor)
        - SIT (see: https://github.com/TheGor/injected_armor_pkgs/tree/developingMine/injected_armor/sit)
        - ClientApiJava (new) (see : https://github.com/TheGor/injected_armor_pkgs/tree/developingMine/injected_armor/client_api_java)
    - Injected_armor_msgs (see: https://github.com/TheGor/injected_armor_pkgs/tree/developingMine/injected_armor_msgs)

In order to perform the state machine, the following steps are required.

1. Roscore
2. Rosrun armor_py_client_api state_machine.py
3. Rosrun armor_py_client_api publisher_send_scene.py
4. Rosrun injected_armor armor it.emarolab.armor.ARMORMainService

In order to launch every single client, the following steps are required.

1. Roscore
2. Rosrun injected_armor armor it.emarolab.armor.ARMORMainService
3. Rosrun armor_py_client_api load_ontology_armor.py
4. Rosrun armor_py_client_api init_sit.py
5. Rosrun injected_armor client_api_java com.rosjava.github.injected_armor.client_api_java.ClientSendObject
6. Rosrun armor_py_client_api clean_ontology.py