TOASTER, Situation Assessment Framework for Human Robot Interaction

July 12, 2016

Abstract

This document aims to explain TOASTER framework from architecture and conceptual level to implementation and computation details.

Contents

1 Introduction **3**
- 1.1 Brief history . . . 3
- 1.2 Why using TOASTER? . . . 3

2 Installation Instructions **4**

3 toaster-lib **4**

4 Global Architecture **6**

5 tools **6**

6 toaster_msgs **7**
- 6.1 Implementation details . . . 7
- 6.2 Symbolic representation and fact . . . 8

7 toaster_simu **10**
- 7.1 Implementation details . . . 10
- 7.2 Inputs . . . 10
- 7.3 Outputs . . . 10
- 7.4 Services . . . 10
- 7.5 Limitations and improvements . . . 12

8 Perceived Data Gathering **12**
- 8.1 Implementation details . . . 12
- 8.2 Facts computation . . . 13
- 8.3 Inputs . . . 14
- 8.4 Outputs . . . 14
- 8.5 Services . . . 14
- 8.6 Limitations and possible improvement . . . 15

9 area_manager **15**
9.1 Implementation details 15
9.2 Facts computation 15
9.3 Inputs 17
9.4 Outputs 17
9.5 Services 17
9.6 Examples 18
9.7 Future work and possible improvement 18

10 agent_monitor **19**
10.1 Implementation details 19
10.2 Facts computation 19
10.3 Inputs 22
10.4 Outputs 22
10.5 Services 22
10.6 Examples 23
10.7 Future work and possible improvement 23

11 move3d_facts **23**
11.1 Facts computation 23
11.2 Inputs 24
11.3 Outputs 24
11.4 Services 24
11.5 Examples 24
11.6 Future work and possible improvement 24

12 belief_manager **24**
12.1 Implementation details 24
12.2 Examples 25

13 database_manager **25**
13.1 Implementation details 25
13.2 Inputs 25
13.3 Outputs 26
13.4 Services 26
13.5 Examples 28
13.6 Future work and possible improvement 28

14 toaster_visualizer **28**
14.1 Implementation details 28
14.2 Inputs 28
14.3 Outputs 29
14.4 Services 29
14.5 Future work and possible improvement 29

15 Facts computation recapitulation **30**

1 Introduction

To understand the environment, the robot needs not only to perceive it but also to reason on the relationships between the different elements and the evolution of the properties describing the world state. This understanding of its surrounding is key in order for the robot to take appropriate decisions according to the situation. TOASTER (Tracking Of Agents and Spatio-TEmporal Reasoning), is a **generic multi-sensor situation assessment framework for human-robot interaction** able to maintain a **world state**, generate **symbolic facts** from geometric computations, track **human motion** and maintain a belief state for the robot(s) as well as an estimate of the **belief state** of its human partners. The framework consists in several components, each with its specific spatial and temporal reasoning.

1.1 Brief history

TOASTER development started in September 2015 in LAAS-CNRS[1]. A previous software named SPARK[2] [1] was used in LAAS for situation assessment. TOASTER was made to replace SPARK with a modular framework. It currently is at the version 0.4 and the framework consists in several modules.

1.2 Why using TOASTER?

TOASTER aims to be a generic situation assessment framework, using versatile sensor inputs. It also aims to build several levels of situation assessment: from world representation to human mental state representation. TOASTER is easily extensible both on the inputs (extending supported sensors) and on the framework itself by using a modular approach. We created modules to estimate the situation of the world and of the agents. In addition, it represents and keeps track of human mental state.

TOASTER was successfully used in several applications of human robot interaction. One of the first application was to use it as a situation assessment component for an adaptive and proactive human aware robot guide [2]. For this application, the robot needs to guide a group of users, through a crowded area, to bring them to a chosen location. One of the challenges in this work was to guide people in a socially acceptable and comfortable manner. To do so, the robot had to adapt its speed to the group in order to lead everyone to the destination with a pace that satisfies most of the users and prevent user disengagement. Another project aimed to build a robot to work with a human in an assembly line. In this scenario, the robot shares the working space with a human. Therefore, understanding the human spatial situation and current activity is key. TOASTER, being able to provide both low and higher level (belief state awareness) situation assessment, was used in a human robot interaction task to build the world state, monitor human actions and track human beliefs. As described in [3], the system uses agent belief awareness provided by TOASTER along with action recognition and context to enhance the robot with intention recognition capacities, which permits to give proactive behavior from the robotic system. The semantic world representation computed

[1] https://www.laas.fr

[2] https://www.openrobots.org/wiki/spark

by TOASTER enable to build a common ground to understand human requests and to generate appropriate speech utterances, such as "bring me the object on the kitchen's table" or even understanding when a human ask a request using multimodal language (such as pointing). In [4], we use TOASTER along with the MORSE simulator for dialog strategy learning.

Some of our latest works aimed to model the human mental state concerning the shared plan to have an adaptive robot partner. These works use the situation assessment frameworks like TOASTER to understand the world state but also to assess the human knowledge on the plan and update it when needed.

2 Installation Instructions

To install TOASTER you will need boost, ROS and toaster-lib. The ROS package Rviz is also used to visualize the data from TOASTER.

To get full instructions to install TOASTER, please refer to the wiki at `https://github.com/Greg8978/toaster/wiki/installationr`.

3 toaster-lib

toaster-lib is a C++ library which defines data structures used in TOASTER. The structures are:

- **Entity**: this class defines a physical element of the world. It can be an agent or a joint or an object.
  - **Agent**: this class defines an entity which has a set of joints. This agent can be a Human or a Robot.
    - **Human**: this class represents a human agent.
    - **Robot**: this class represents a robotic agent.
  - **Joint**: this class defines an entity which belongs to an agent. Despite its name, this can be any part of the agent's body: head, wrist, hand, gripper...
  - **Object**: this class defines an object. This object can be a MovableObjet or a StaticObject.
    - **MovableObject**: this class defines objects that may move during the interaction.
    - **StaticObject**: this class defines objects that can't move.
- **Area**: this class defines an area. An area is a defined location of the environment.
  - **PolygonArea**: this class defines a polygonal area.
  - **CircleArea**: this class defines a circular area.

The attributes and links between these class are shown in the UML diagram at the figure 1. These key concepts (data structures) are used by TOASTER to represent the world state and to perform geometrical computations in order to obtain situation awareness for human-robot interaction. Apart from this, toaster-lib defines frequently used mathematical functions and aims to work with object-oriented approach.

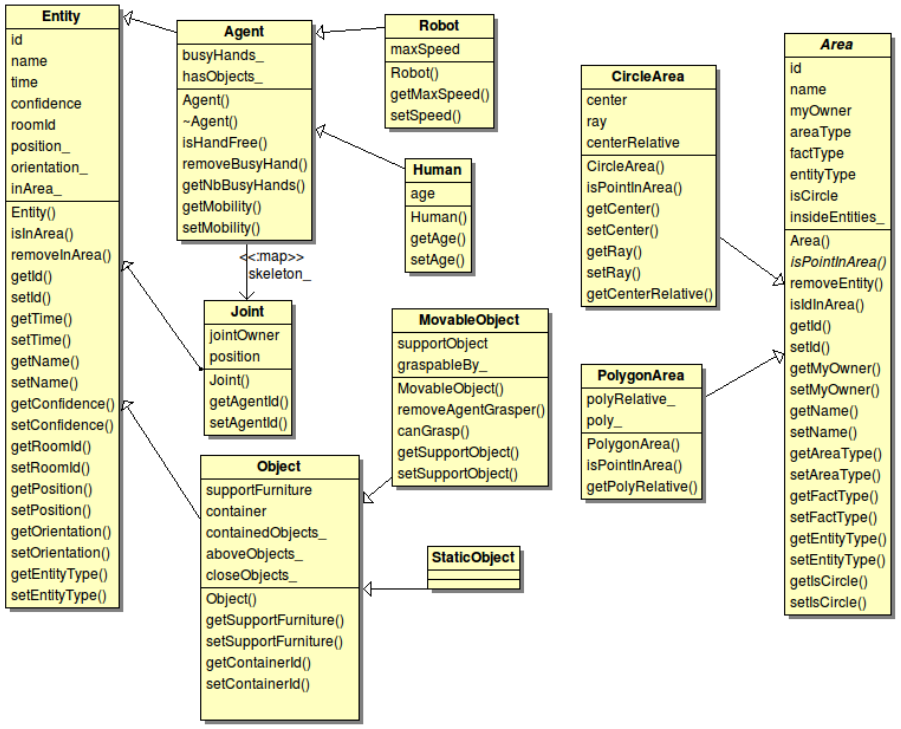

Figure 1: UML diagram from toaster-lib representing the different class and their attributes.

4 Global Architecture

TOASTER is a modular framework. Therefore, each component can be used independently and the framework configuration can be adapted to the requirement of the chosen application. As an example, it is possible to abstract the lower level and directly use the database by adding facts to the database tables through requests from scripts or external components.

However, some components can use data provided by other components. Fig. 2 presents how the modules can be linked together and shows a global representation of the situation assessment architecture.

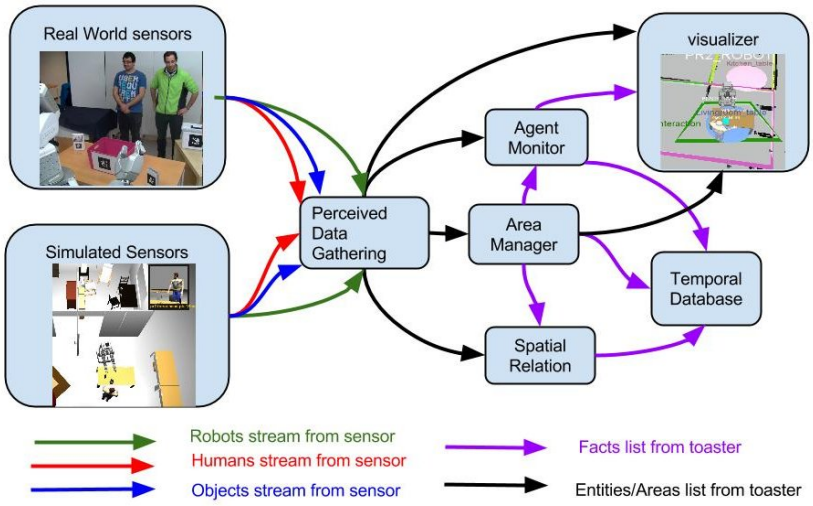

Figure 2: TOASTER architecture and interactions of the situation assessment components

5 tools

This repository is made to share tools linked to toaster. One of the repositories is *toaster-scripts*. It contains launch files and scripts.

It contains roslaunch files to launch several nodes of the toaster architecture with some parameters. As an example, the roslaunch file at *roslaunch/toaster_simu.launch* is running *toaster_simu*, *pdg*, *agent_monitor*, *area_manager*, *toaster_visualizer* and *rviz*. For *pdg* is uses parameters defined in *pdg/params/toaster_simu.yaml* so that pdg uses topics from *toaster_simu* as input.

This repository also provides shell scripts. As an example the script */shell/spencer/area_spencer2.sh* will send requests to *area_manager* to configure areas around a guiding robot as shown in 4.

In the *tools* repository, we also provide python script. As an example, the python script in *python/simu_set.py* is calling services in *toaster_simu* and *agent_monitor* to configure the environment as in 4 and to set an agent *Greg* as a keyboard controlled entity and a monitored agent.

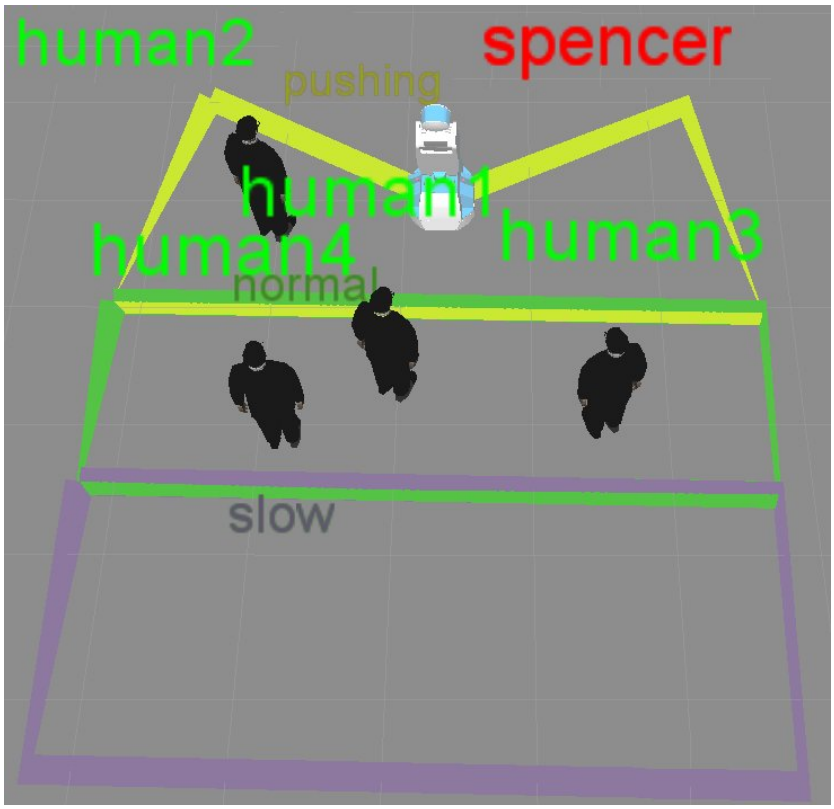

Figure 3: Areas are created and attached to the robot to assess the situation of the following group.

6 toaster_msgs

This package defines all ROS messages and services used in TOASTER. All the messages type like Area, Agent, Entity, Fact, etc, have been defined in msg folder while service like AddAgent, AddArea, etc, in srv folder. Descriptions in .msg and .srv files makes it easy for ROS tools to automatically generate source code for the message type in several target languages.

6.1 Implementation details

toaster_msgs includes uses one message type in defining another higher level of message. This establishes relationship of hierarchy and aggregation among various messages, can be seen in Fig. 5. Services used in various modules are described in subsequent sections. To see the detailed structure of each messages refer to `https://github.com/Greg8978/toaster/tree/master/toaster_msgs/msg` and for services: `https://github.com/Greg8978/toaster/tree/master/toaster_msgs/srv`.

This package also provides classes to help other modules to read the messages and to convert then into *toaster-lib* data structure. These class are *ToasterFactReader, ToasterObjectReader, ToasterHumanReader, ToasterRobotReader*.

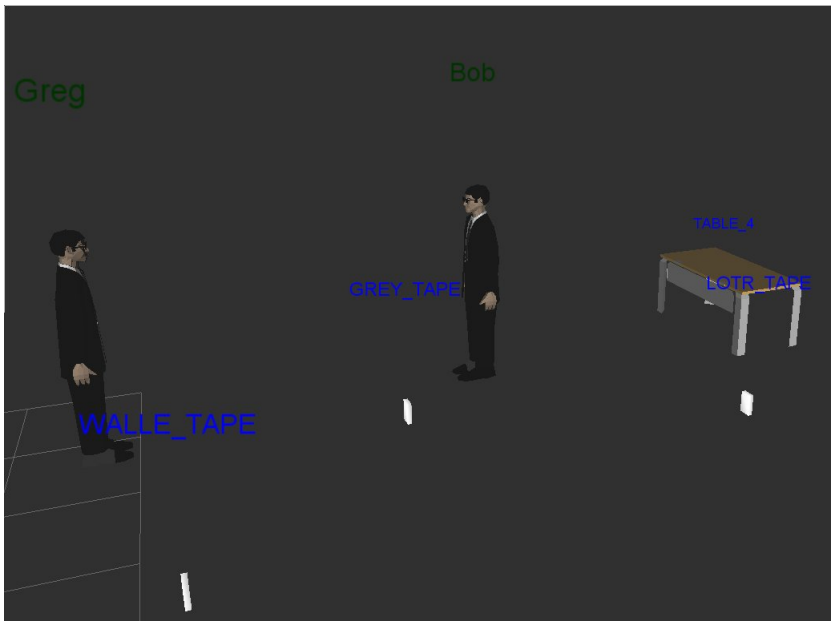

Figure 4: Environment setup from python/simu_set.py script.

6.2 Symbolic representation and fact

Our situation assessment modules perform geometric computations to get a symbolic representation of the world. This symbolic representation is made based on a list of a data structure named a *fact*. In our system, a *fact* is a vector, with several fields, used to represent a property of the environment. We detail below the fields of this vector.

- *Subject*: the entity on which the property applies (e.g.: *Red_Mug*, *Human1*, *Pr2*, *Human_Right_Hand*).
- *Property*: the property linked to the *Subject* (e.g.: *isOn*, *isFull*, *isMoving*, *isPointing*, *canSee*).
- *Target*: the property may link the entity-subject with an entity-target. As an example if an entity *BOOK* is on an entity *TABLE*, *BOOK* will be the subject while *TABLE* will be the target.
- *PropertyType*: this parameter defines the category in which the property falls (e.g.: *position*, *state*, *motion*, *posture*, *affordance*). Using this parameter, if an external module adds a fact, it is still possible to know which kind of property it is.
- *Value*: a property may have a value linked to it. As an example, the property *isFull* or *isMoving* may have the value *TRUE* or *FALSE* (if we want to have a close world representation). As another example, if we represent the distance between joints, such as robot's hand to human head, the *Value* parameter could be set to *DANGER*, *CLOSE* or *FAR*, or even holds a numerical value. In some situations, to represent a lack of

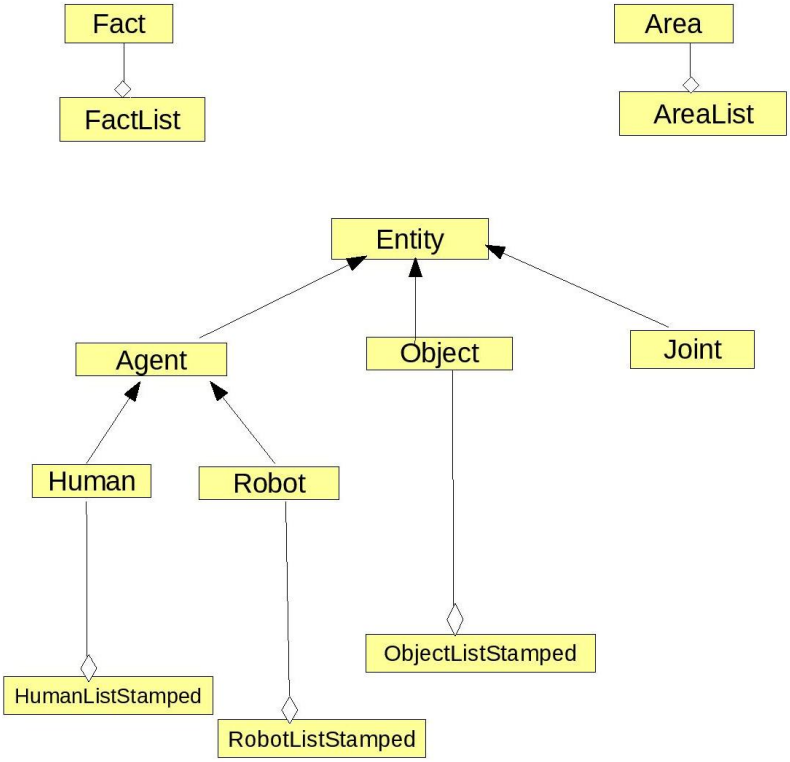

Figure 5: Relationship between .msg structures in toaster_msgs

knowledge on a property and the awareness of this lack, the fact's value can also be set to *unknown*.

- *Confidence*: this figure between 0 and 1 represents the reliability of the fact. This can be linked to sensor reliability or to the property computation itself.
- *Time*: the time at which the fact was computed.
- *FactObservability*: the probability that a human would acquire the awareness of the fact if he sees the Subject of the fact.

As an example, the vector $< Subject = Bob\_Right\_Hand,\ Property = isMovingToward, Target = Red\_Book, PropertyType = motion, Confidence = 0.8,\ time = 145571646570,\ FactObservability = 0.7 >$ represents the fact that, at the given time, the hand (*Bob_Right_Hand*) of Bob is going toward a book (*Red_Book*) with a confidence of *0.8*. The following sections describe spatial and temporal reasoning components that generate these kind of facts.

7 toaster_simu

toaster_simu is an inbuilt-simulation component of TOASTER which allows to add entities (objects, robots, humans and joints) and provides the desired environment to TOASTER for situation assessment. This makes possible to quickly set up an interaction environment in order to test new features or new modules without requiring any robotic device or any real environment.

7.1 Implementation details

Fig. 6 gives an idea of the implementation of services and topics in toaster_simu.

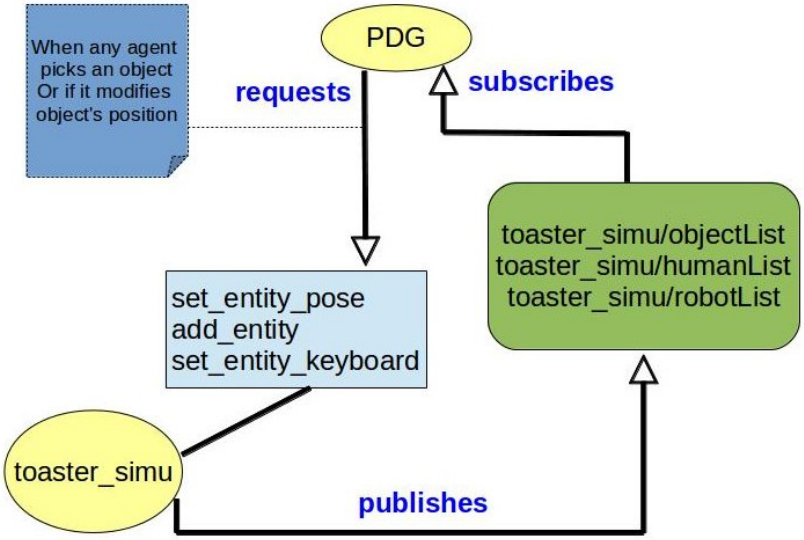

Figure 6: toaster_simu services and messages. Ellipses represent nodes, rectangles represent services and messages.

7.2 Inputs

Inputs for this module are id, type and position of entities to be added to the environment being assessed. They are provided through services as described below. Another set of inputs will be from keyboard if any entity is teleoperated (see fig. 7 for keyboard teleoperation keys).

7.3 Outputs

Outputs for toaster_simu component are published on topics */toaster_simu/humanList*, */toaster_simu/robotList* and */toaster_simu/objectList* which are subscribed by higher level component PDG.

7.4 Services

This component provides the following services :

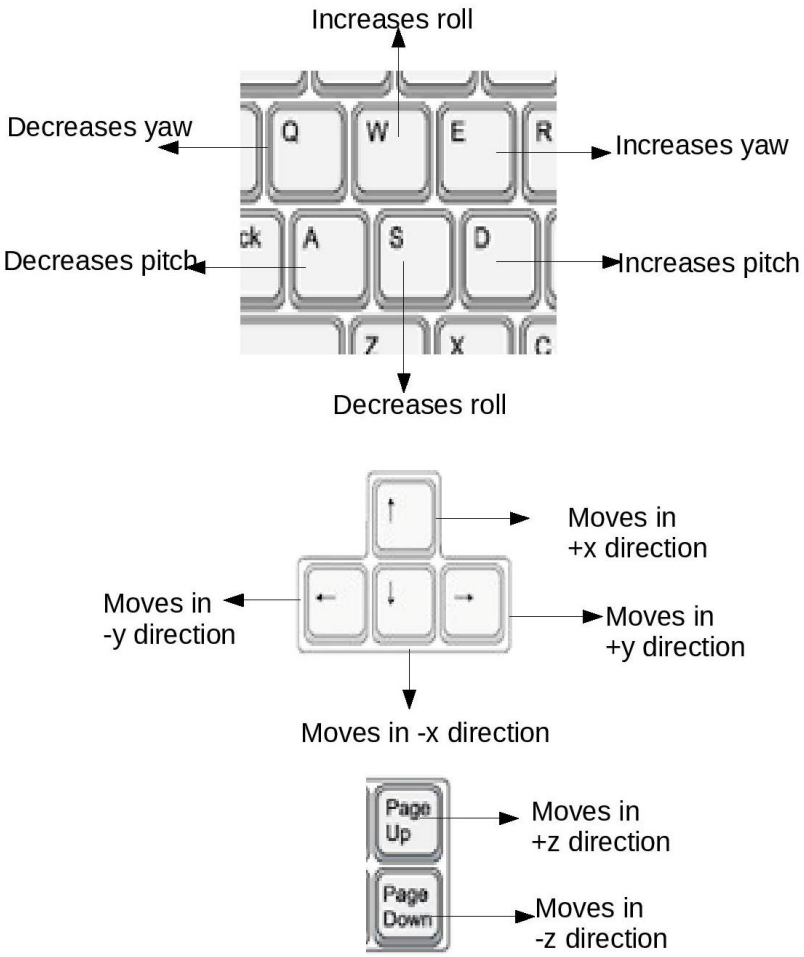

Figure 7: keys to control the entity

- **add_entity** - enables you to add an entity to the environment. It takes a string *id* to give to the entity, a string *name* which will be also use to find the corresponding 3D model, a string *type* which should be set to the entity type (human, robot, object or joint). The last parameter is the string *ownerId*. It is used for joint entities to give the id of the agent it belongs to.

  Shell command to call the service: *rosservice call /toaster_simu/add_entity "id: " name: " type: " ownerId: ""*

- **remove_entity** - enables you to remove an entity from the environment. It takes strings to identify the entity.

  Shell command: *rosservice call /toaster_simu/remove_entity "id: " type: " ownerId: ""*

- **set_entity_keyboard** - enables you to set any entity to keyboard mode

so that it can be controlled by keyboard. It takes a sting with the id of the entity you want to control.

Shell command: *rosservice call /toaster_simu/set_entity_keyboard "id: ""*

- **set_entity_pose:** - enables you to set position of any existing entity. The three first parameters *id, ownerId, type* are strings meant to identify the entity whose position is to be changed. The last parameter is the pose in which we want to put the entity with the type *geometry_msgs/Pose* from ros.

  Shell command:

  rosservice call /toaster_simu/set_entity_pose "id: ", ownerId: ", type: ", pose: position: x: 0.0 y: 0.0 z: 0.0 orientation: x: 0.0 y: 0.0 z: 0.0 w: 0.0""

7.5 Limitations and improvements

The current version of toaster_simu allows to quickly add or remove any kind of entity from the environment. However, as it is meant to be a tool for quick testing, it doesn't provide a simulation of perception as a simulator would do but directly add a "perceived" world state. Nevertheless, TOASTER is also compatible with inputs from the robotic open-source software MORSE.

An other limitation is that an agent can be added with his joints, however for now, the joint position is not updated with the agent's one[3].

8 Perceived Data Gathering

In a scenario of human-robot interaction, data concerning three entities (humans, objects and robots) may come from various sensors with heterogeneous data-types. As we wish to keep the geometrical reasoning generic while having a flexible data-source system, we use a separate component PDG (Perceived Data Gathering) to collect the data from all the required sensors and publish them in a unique format usable by any other TOASTER component.

8.1 Implementation details

The current version already supports several inputs. For human tracking, TOASTER can get data from kinect-like and motion capture devices. Concerning objects, an object recognizer based on tags and stereo vision is supported. Finally, two different models of robots are directly functional in TOASTER. However the set of inputs is easily extensible by adding a new sensor data reader to the *PDG* module. Additionally, TOASTER can use HRI simulators. The robotic simulator MORSE[4] [5] and a built-in simulation component (toaster_simu) are supported as inputs for any type of entities (objects, humans or robots). These two simulators allow to quickly test TOASTER components and allow to have a partially simulated interaction. As an example, we can have a real human interacting with a simulated robot by using data obtained from a

[3] see `https://github.com/Greg8978/toaster/issues/19`

motion capture for the human in real world, and data from a simulator output for the robot pose. The figure 8 illustrates the messages and services provided by *PDG*

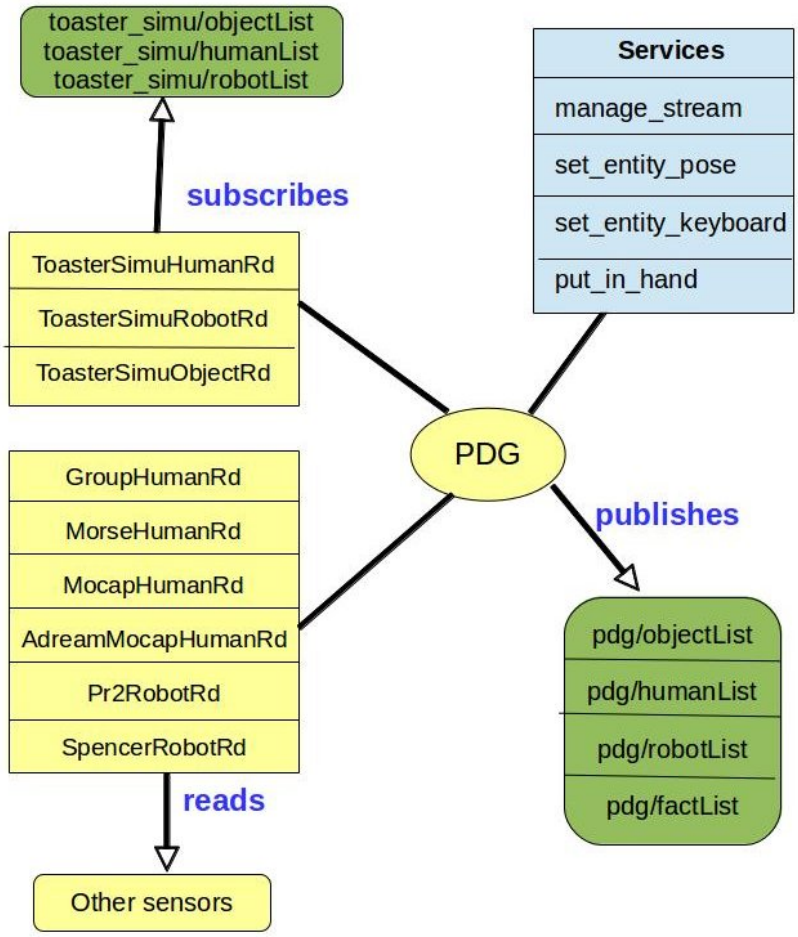

Figure 8: Implementation details of Perceived Data Gathering module

8.2 Facts computation

The facts generated by this module are:

- **IsInHand**: this fact is produced when an object has been set in an agent's hand. The *property* is *IsInHand*, the *propertyType* is *position*, the *subProperty* is *object*, the *subjectId* is the id of the object, the *targetId* is the id of the joint (hand), the *targetOwnerId* is the id of the agent, the *time* is set with the perception time of the joint, the *valueType* is set to zero and the *stringValue* is set to *true*.
- **IsSeen**: this fact is produced to tell that an object is currently seen (detected) by the robot. The fact *property* is *IsSeen*, the *propertyType* is

affordance, the *subjectId* is the id of the object perceived. The *time* of the fact is set using the time of the object perception. The *valueType* is set to zero and the *stringValue* is set to *true*.

8.3 Inputs

The data streams about position, orientation and other properties of humans, robots, objects and joints from various sensors like kinect act as an input for this module.

8.4 Outputs

Output is the sensor data published on various topics with well-defined data structure.

8.5 Services

On running this node, one can access following services:

- **manage_stream** - It enables to specify, at any moment, which sensor's data to use as raw input by putting the corresponding flag at *true*. This makes the PDG component highly adaptable to the data needed for the current task and to the set of available sensors.

  Shell command: *rosservice call /pdg/manage_stream "morseHuman: false, niutHuman: false, groupHuman: false, mocapHuman: false, adream-MocapHuman: false, toasterSimuHuman: false, pr2Robot: false, spencer-Robot: false, toasterSimuRobot: false, toasterSimuObject: false*

- **set_entity_pose** - enables you to set position of any existing entity. The three first parameters *id, ownerId, type* are strings meant to identify the entity whose position is to be changed. The last parameter is the pose in which we want to put the entity with the type *geometry_msgs/Pose* from ros.

  Shell command:

  rosservice call /toaster_simu/set_entity_pose "id: ", ownerId: ", type: ", pose: position: x: 0.0 y: 0.0 z: 0.0 orientation: x: 0.0 y: 0.0 z: 0.0 w: 0.0""

- **put_in_hand** - This service has provision of adding an object in any agent's hand. It actually links the object position to a joint. All three inputs are of string type to define the object and the joint.

  Shell command: *rosservice call /pdg/put_in_hand "objectId: " agentId: " jointName: ""*

- **remove_from_hand** - It enables to remove object from agent's hand. It takes a string *objectId* of the object that is to be removed.

  Shell command: *rosservice call /pdg/remove_from_hand "objectId: ""*

8.6 Limitations and possible improvement

PDG is meant to aggregate data from various sensors. It is possible to use several modalities to get the same entity's position. For now, there is no way to manage a conflict if several sensors detect the same object with a different position.

An other way to improve *PDG* is to add supported inputs. This can be easily done by using heritage from one of the class *ObjectReader, RobotReader* or *HumanReader* according to what kind of entity is localized by the sensor.

9 area_manager

We define an area as a bounded region in the environment with semantic meaning. This module associates area with the entity to calculate their interaction with other entities. These areas are useful to get a first discrimination of the situation and conditional computation.

9.1 Implementation details

In the current implementation, an area is bi-dimensional (however the concept is still valid for tri-dimensional areas). Areas can be a polygon or a circle, static or dynamic. In TOASTER, areas have several parameters: *areaType, factType, entityType, owner*. The *areaType* defines the category of the area (such as room, escalator, television_area). As areas could be used for different situation, we use the *factType* to define what type of calculation needs to be done for entities inside the current area. This parameter is very linked to the semantic of the area. The *entityType* defines what kinds of entities are concerned by the area. Entities which are not in the category will be ignored. Finally, *owner* defines which entity "owns" this area. If set, the area is updated with the owner's position and orientation. For a clear relationship between pdg node with its topics, services and other nodes, refer Fig. 9.

9.2 Facts computation

The facts generated by this module are:

- **IsInRoom**: this fact is produced for area with *areaType* equal to *room*. When a concerned entity (an entity which belongs to the class *entityType* of the room), is inside the area, this fact is generated. The *property* is *IsInRoom*, the *propertyType* is *position*, the *subProperty* is *room*, the *subjectId* is the id of the entity inside the room, the *targetId* is the name of the area, the *time* is set with the perception time of the entity, the *valueType* is set to zero and the *stringValue* is set to *true*. **example:** *Bob IsInRoom Livingroom ...*

- **IsAt**: this fact is produced for area with *areaType* equal to *support*. When a concerned entity (an entity which belongs to the class *entityType* of the area), is inside the area, this fact is generated. The *property* is *IsAt*, the *propertyType* is *position*, the *subProperty* is *location*, the *subjectId* is the id of the entity inside the room, the *targetId* is the name of the area, the *time*

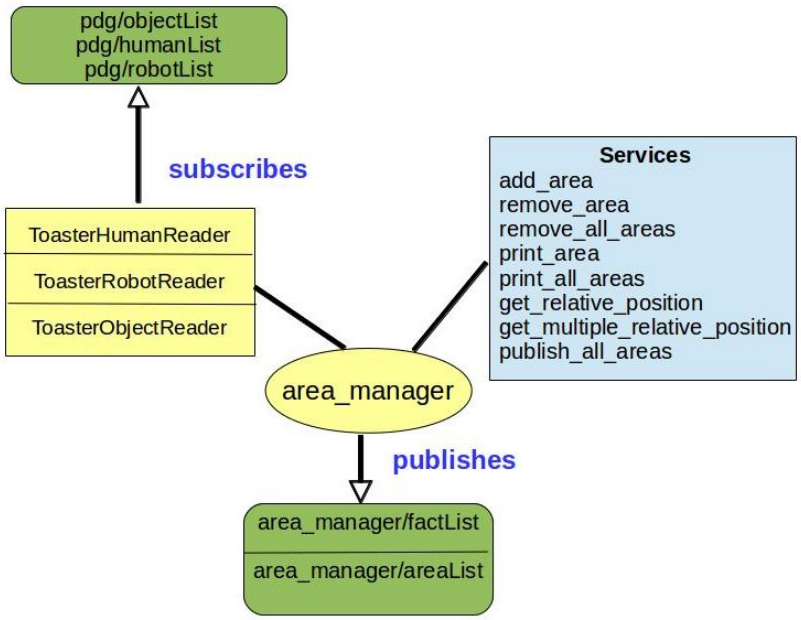

Figure 9: Implementation of area_manager module

is set with the perception time of the entity, the *valueType* is set to zero and the *stringValue* is set to *true*. **example:** *Bob IsAt Livingroom_table ...*

- **IsInArea**: this fact is produced for any other kind of area with *areaType* different to *support* and *room*. When a concerned entity (an entity which belongs to the class *entityType* of the area), is inside the area, this fact is generated. The *property* is *IsInArea*, the *propertyType* is *position*, the *subProperty* is set with *areaType*, the *subjectId* is the id of the entity inside the room, the *targetId* is the name of the area. If the area has an *owner*, we set *targetOwnerId* with its id. The *time* is set with the perception time of the entity, the *valueType* is set to zero and the *stringValue* is set to *true*. **example:** *Bob IsInArea corridor ...*

- **AreaDensity**: this fact is produced to get the density of element in an area if area has factType set to *density*. The fact *property* is *AreaDensity*, the *propertyType* is *density*, the *subProperty* is set with *ratio*. The *subjectId* is the name of the area. The *time* of the fact is set using the time of the entity in the area. The *valueType* is set to one and the *doubleValue* is the ratio *nbEntInArea*/`nbEnt`, with nbEntInArea the number of concerned entities (from the *entityType* of the area) and nbEnt the total number of these entities.

- **IsFacing**: this fact is computed for entity inside an area with *interaction* as factType. We compute the orientation of the entity and we compare it with the position of the area owner. If this orientation is close enough, we consider that the entity is facing the area owner.

The fact *property* is *IsFacing*, the *propertyType* is *posture*, the *subProperty* is set with *angle*. The *subjectId* is the id of the entity. The *targetId* is set with the id of the area's owner. The *time* of the fact is set using the time of the entity perception. The *valueType* is set to zero, the *stringValue* is *true* and the *doubleValue* is the angle of deviation between the orientation toward the area's owner and the actual entity orientation.

To compute the presence of an element in an area, we use the distance from the center for circular area which we compare to the ray of the area. For polygonal areas, we use the boost library and the *within* function of the polygon (see `http://www.boost.org/doc/libs/1_61_0/libs/geometry/doc/html/geometry/reference/algorithms/within/within_2.html`) .

9.3 Inputs

area_manager takes outputs of PDG module i.e. orientation, positions and other properties of entities.

9.4 Outputs

It publishes facts like isInArea, isAt, AreaDensity on topic named */area_manager/factList* and areas on topic */area_manager/areaList*.

9.5 Services

Services provided by area_manager are :

- **add_area**- In the current implementation, an area is bi-dimensional (however the concept is still valid for tri-dimensional areas). Areas can be a polygon or a circle, static or dynamic. In TOASTER, areas have several parameters: *areaType, factType, entityType, owner*. The *areaType* defines the category of the area (such as room, escalator, television_area). As areas could be used for different situation, we use the *factType* to define what type of calculation needs to be done for entities inside the current area. This parameter is very linked to the semantic of the area. The *entityType* defines what kinds of entities are concerned by the area. Entities which are not in the category will be ignored. Finally, *owner* defines which entity "owns" this area. If set, the area is updated with the owner's position and orientation. Concerning the id of the area, if an id is given, the area will have this id, replacing the previous area with the same id if any. When no *id* is specified (the default value 0 is given) the module will assign the area with the first available unsigned integer. The message format of the service looks like this:

  Shell command:

  rosservice call /area_manager/add_area "myArea: id: 0 name: " myOwner: " areaType: " factType: " entityType: " isCircle: false center: x: 0.0, y: 0.0, z: 0.0 ray: 0.0 poly: points: - x: 0.0, y: 0.0, z: 0.0 insideEntities: [0]"

- **remove_area** - Areas can be added and removed as per the requirements. This service is used to remove the area where input is the area's numeric id. This id is positive.

  Shell command:

  rosservice call /area_manager/remove_area "id: 0"

- **remove_all_area** - as the name suggests, service remove_all_areas removes all areas created for the situation assessment.

- **print_area** - This service take id of area and prints its details including its position, orientation, owner and list of entities inside this area. Similarly, the service print_all_areas prints details of all areas.

- **get_relative_position** - This service computes relative position of target with respect to subject and shows it in higher level terms like right, left, ahead, back etc, which are simple for human understanding. It takes string id of subject and target entity.

  Shell command:

  rosservice call /area_manager/get_relative_position "subjectId: " targetId: ""

- **get_multiple_relative_position** - this service is used to get relative position of an object compared to an other from an agent point of view. It takes a string *agentSubjectId* of the agent from which we want the point of view, the string *objectSubjectId* giving the id of the object which we will use to describe the position of the object identified by the string *objectTargetId*.

  Shell command:

  rosservice call /area_manager/get_multiple_relative_position "agentSubjectId: " objectSubjectId: " targetId: ""

- **publish_all_areas** - This service controls the publishing of areas on /area_manager/areaList topic. If this service is called, a parameter name publishingArea_ is negated. The node only publishes on areaList topic if this parameter is positive. By default, it is set to true.

9.6 Examples

As an example, it is possible to define a polygon in front of the robot (*myRobot*) to know if a human is in an interaction configuration. This area triggers some computation such as the human body orientation. It would have as set of parameters ¡*interaction, orientation, humans, myRobot*¿ and is updated with the position and orientation of *myRobot*. This example is illustrated by the pink area of Fig. 10.

9.7 Future work and possible improvement

To improve this module, we should add 3d areas. Also currently the number of areaType is quite limited. A way to improve would be to have a configuration file where we could define a new type of area and tell which facts computation

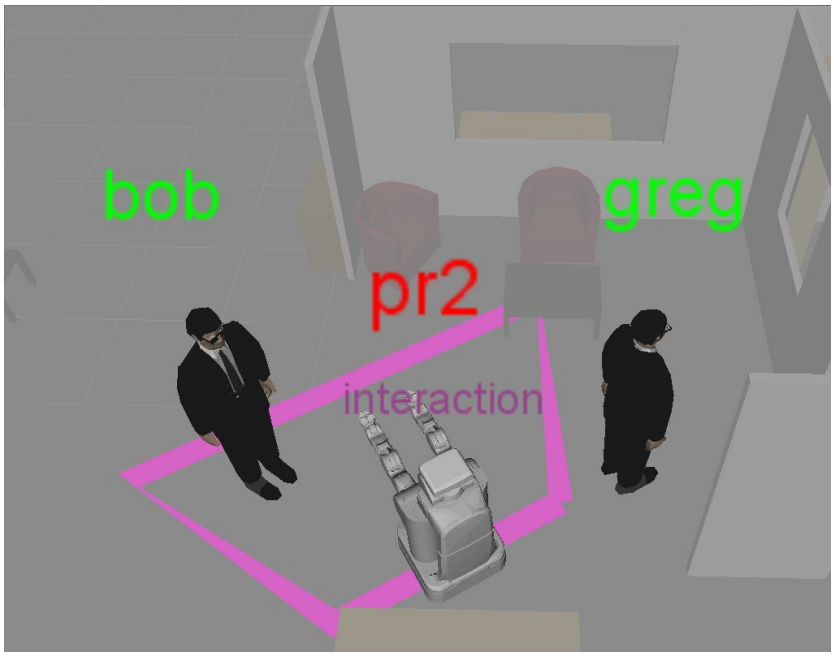

Figure 10: Illustration of an area to detect a user in "interaction" configuration (inside robot's interacting area and facing the robot).

should be triggered for this kind of area. As an example, the computation of facts in *move3d_facts* or *agent_monitor* should be triggered according to the configuration with areas. As an example, it could be useless to compute if an agent isMovingToward an object when the object is not in the same room.

10 agent_monitor

When the robot has to perform a task collaboratively with a human partner, it is key for the robotic system to understand the human's activity. To do so, one component of the TOASTER framework, called Agent Monitoring, is computing facts concerning the agent's motion, posture and distance regarding points of interest.

10.1 Implementation details

To implement the desired functionality for this module, circular buffer data structure has been used. At all time, the module records the position of any entity, for a short period of time, in a time stamped circular buffer. This allow to access the entity position at a given time (supposed not too far in the past).

10.2 Facts computation

The facts generated by this module concerning the monitored agent's body are:

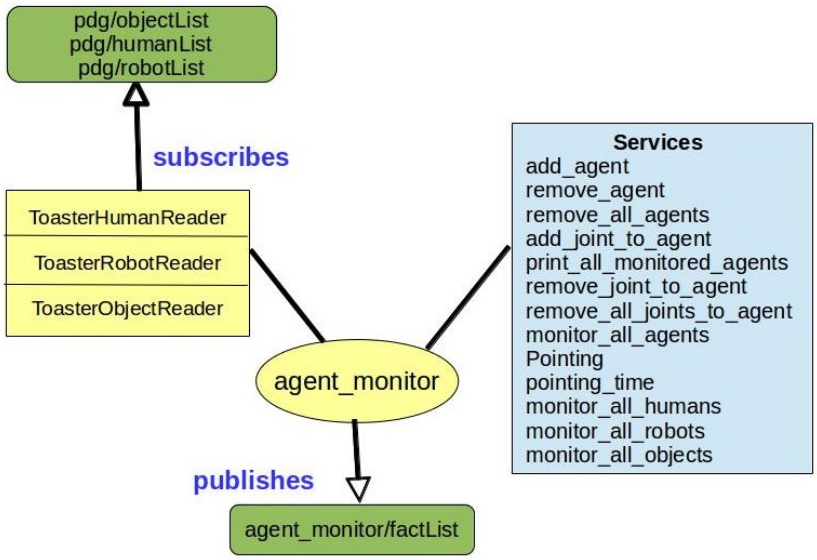

Figure 11: Implementation of agent_monitor module

- **IsLookingToward**: this fact is computed for any agent monitored. It uses the head joint of the agent and an angular aperture value to compute a cone from the agent's head. If an object is in the cone, or if the "head" of an agent is in the cone, it's considered as looked.

  The fact will have *IsLookingToward* as *property*, the *propertyType* is *attention*, the *subProperty* is *agent*, the *subjectId* is the id of the monitored agent, the *targetId* is the id of the entity looked upon, the *time* is set with the perception time of the monitored agent, the *valueType* is set to one and the *doubleValue* is set with the angular distance from the center axis of the cone to the taget entity. The *confidence* is set with a normalization from this angle *angleEnt* to the global cone angle *angleCone* : $angleCone - anlgeEnt/angleCone$.

  example: *Bob IsLookingToward LOTR_BOOK ...*

- **IsMoving**: this fact is produced if the monitored agent global body is in motion. To compute the motion, we take advantage of the time stamped circular buffer of toaster-lib. In *agent_monitor*, entities positions are recorder in the time stamped circular buffer. To compute the fact, we compute the distance from the given time to the last data (which gives the speed). Using ros dynamic reconfigure, it is possible to set the duration and the minimal speed required to consider the agent as moving. The default computation is made for $250ms$ and the speed threshold is $0.12m/s$. It means that, above this speed the agent is considered in motion and the fact will be generated.

  The *property* is *IsMoving*, the *propertyType* is *motion*, the *subProperty* is set with *agent*, the *subjectId* is the id of the monitored agent, *time* is set with the perception time of the monitored agent, the *valueType* is set to zero, the *stringValue* is set to *true* and the *doubleValue* is set with the

agent's speed in m/s. The *confidence* is the speed devided per 5 km/h (so it will reach 1 if it moves at 5 km/h or above).

example: *Bob IsMoving true doubleValue=1.0 confidence =0.72 ...*

- **IsMovingToward** (direction): this fact is computed only if the agent is moving. To compute this fact, we get the direction of the monitored agent's body from the trajectory. To do so, we use the last data of the agent's position and a previous data defined by the time difference with the last one. Once we get the global direction of the human for the defined time lapse, we compare this direction with the direction of the agent toward the entities of the environment. Using an angular threshold, we are able to tell which entities it may be going toward and give a confidence according to the angle between the trajectory direction and the entity direction. The time lapse and the angular threshold can be changed with ros dynamic reconfigure. The default values are $500ms$ and $1.0rad$.

  The *property* is *IsMovingToward*, the *propertyType* is *motion*, the *subProperty* is set with *direction*, the *subjectId* is the id of the monitored agent, *targetId* is the id of the entity it is moving toward. The *time* is set with the perception time of the monitored agent, the *valueType* is set to zero, the *stringValue* is set to *true*. The *confidence* is set with a normalization from the deviation angle *angleDevi* (angle between the direction of the trajectory and the direction toward the object) with the threshold angle *angleTh*: $angleTh - anlgeDevi/angleTh$.

  example: *Bob IsMovingToward BLUE_BOOK true ...*

- **IsMovingToward** (distance): this fact is computed only if the agent is moving. To compute this fact, we get the position of the monitored agent at the current time and at a previous time given in parameter. We also get the position of other entities at the same times. We compute the distance between the entities at both times. If the distance is decreasing above a given threshold for the time lapse, we will generate the fact IsMoving-Toward. The time lapse and the distance used can be changed using ros dynamic by giving a time and a speed (the speed the agent is moving toward the object). The default values are $250ms$ for the time lapse and $0.12m/s$ for the speed threshold.

  The *property* is *IsMovingToward*, the *propertyType* is *motion*, the *subProperty* is set with *distance*, the *subjectId* is the id of the monitored agent, *targetId* is the id of the entity it is moving toward. The *time* is set with the perception time of the monitored agent, the *valueType* is set to zero, the *stringValue* is set to *true*. The *confidence* is set with a normalization from the reative speed between the two elements.

  example: *Bob IsMovingToward BLUE_BOOK true ...*

Examples of IsMovingToward fact representation are shown in 12

When the monitored agent is not moving, we compute some facts concerning its monitored joints. We compute in a similar manner the facts *IsMoving* and *IsMovingToward* so we won't detail it again. We also compute facts concerning distances between the monitored joint and other entities:

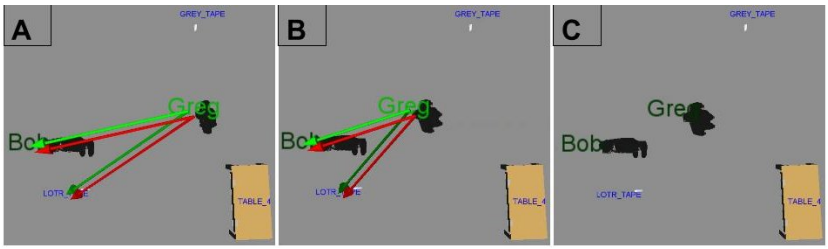

Figure 12: Representation of the facts IsMoving and IsMovingToward.

Distance: this fact is computed only if the agent is not moving. To compute this, we basically compute the 3d distance between the joint monitored and the other entities of the environment.

The *property* is *Distance*, the *propertyType* is *position*, the *subProperty* is set with *3D*, the *subjectId* is the id of the monitored agent's joint, *subjectOwnerId* is set with the id of the monitored agent. The *targetId* is the id of the entity we compute the distance with. The *time* is set with the perception time of the monitored agent, the *valueType* is set to zero, the *stringValue* is set to *"reach", "close", "medium", "far"* or *"out"* according to the distance value. These threshold can be changed using ros dynamic reconfigure. The *doubleValue* is set with the actual distance value between the agent joint's and the entity.

example: *RIGHT_HAND BOB Distance BLUE_BOOK reach ...*

Note: To get reliable data for *isMovingToward*, you may want to combine both facts (direction and distance).

10.3 Inputs

This component of TOASTER reads the topics published by PDG and uses it as inputs to compute required facts.

10.4 Outputs

It publishes facts like IsMoving, IsMovingToward, IsLookingToward, Distance on topic /agent_monitor/factList.

10.5 Services

Main services of agent_monitor are :

- **add_agent** - This service adds agent to the list of agents to be monitored by this module. It takes string id of the agent to be added. Similarly, add_joint_to_agent service adds joint of an agent to the list of joints to be monitored closely by the agent_monitor.

  Shell command:

  rosservice call /agent_monitor/add_agent "id: ""

- **remove_agent** - This service removes agent from the list of agents to be monitored by this module. It takes string id of the agent to be removed. Similarly, working of other services like remove_all_agents, re-

move_all_joints_to_agent, remove_joint_to_agent, monitor_all_agents can be understood by their names.

- **pointing** - It gives the id of the entity towards which the given joint of an agent is pointing with some level of confidence. It is computed for given threshold of pointing distance and angle threshold.

  Shell command:

  rosservice call /agent_monitor/pointing "pointingAgentId: 'HERAKLES_HUMAN1' pointingJoint: " pointingJointDistThreshold: 0.0 angleThreshold: 0.0"

  Similarly, the service pointing_time shows the id of agent towards which given joint of an agent is pointing at a given pointing time.

10.6 Examples

Using the circular buffer we are able to know what the human was pointing at, at a given moment. This can be useful for a fusion component of a dialog system. As an example, if a human asks "give me that", if the speech recognition is able to provide the time when the human said "that", we can request to the agent monitor component what objects were pointed by the human at this time. This fact is computed on request. The request returns a list of entities with a probability on each candidates.

10.7 Future work and possible improvement

The computation in agent_monitor are basically using the data of the agent or entity at a previous time and we compare it with the last data. A better way to compute facts concerning motion would be to use more than two point of time and using filters like Kalman or better way to use the circular buffer to avoid windowing effect.

As mentioned before, we could also have conditional computation to minimize the resource consumption.

11 move3d_facts

NOTE: This is currently a work in progress.

One of the components is responsible for the spatial relation computation. It makes a 3d representation of the world state and computes properties to describe relations between objects (such as *isIn*, *isOn*, *isNextTo*) and affordances of agents toward objects (*canSee*, *canReach*). One can make requests concerning the relative position of entities toward an agent's point of view. This kind of requests enhance the robot with a geometrical perspective taking ability to talk to the human with his own references, improving the social aspect of the system.

11.1 Facts computation

The fact list will be given once the module integrated to the toaster architecture.

11.2 Inputs

This module subscribes to topics /agent_monitor/factList and /area_manager/factList.

11.3 Outputs

It does computation in 3-dimensions and publishes facts like isVisible, isReachableBy, isOn, isIn on /move3d_facts/factLsit topic.

11.4 Services

This component has no services.

11.5 Examples

As an example the facts (simplified here) ¡*BLUE_BOOK isIn BOX1*¿ or ¡*HUMAN1 canReach BOX1*¿ can be generated by this module.It can also tell the relative position of an entity, compared to another, from the agent's perspective. For example, it makes possible for the robot to tell to the human "the mug you are looking for is **on your right**" or "please give me the book which is **for you on the left of the red mug**".

11.6 Future work and possible improvement

12 belief_manager

This component is an older version of *database_manager* module of TOASTER. It updates the world states on observing the changes in agent's surrounding. It enhances the robotic system with Theory of Mind skills and more specifically conceptual perspective taking. To do so, it maintains a belief state for the robot, which comes from the world state, and assesses the belief state of the other agents. In the latest version of TOASTER, this functionality has been embedded into database_manager node.

12.1 Implementation details

A belief state is a symbolic representation of a world state, i.e. a set of facts that the agent believes to be true at a given time. Agents can have a different representation of the world, resulting in different Belief states.A belief state is a symbolic representation of a world state, i.e. a set of facts that the agent believes to be true at a given time. Agents can have a different representation of the world, resulting in different Belief states.We implemented a rule based fact reasoning process in order to build belief state of each agent and update it when needed. Human belief models are updated using the awareness on the human perception affordances. When the robot's belief model is updated, the robot checks that the humans are able to perceive the changes in the environment before updating their belief state. To be aware of the changes in the environment a human needs either to observe the *event* (there is an *event* when a change in the environment occurs, such as an action performed by an agent), or to directly observe the new world state and the changes produced in the. The management

of the former case of update is made by the supervision layer which assesses if the human is able to perceive the event. This decision can rely, (1) on the fact that the human is in the same room, and (2) on the visual affordances of the human toward entities implied in the event. For the later case of update, we use the human visibility toward the subject of the property, combined with the observability of the given property, to assess if the human is able to update his belief on the world state. The confidence on the fact that the human is aware of the new property is computed by combining the confidence on the human visibility toward the subject of the current fact to update and the observability of the fact itself. This way, if an object is hard to perceive by the human (partially occluded), or if the property is hard to observe (low observability), the robot will have a low confidence on the fact that the human actually holds this belief.

12.2 Examples

As an example, if a human H is not aware that a cup has been filled with hot water, and if he can't see properly the cup, the property *hotCup* will have a low confidence in H belief's state. An other example, if the human H is away when an agent A fills the cookie-box, when H comes back, if the cookie-box is closed, the fact that it is filled won't be observable (observability null). Consequently, H mental state will still have the fact *cookie-box isEmpty* set at *TRUE* in his belief state (resulting in a false belief situation).

13 database_manager

To store and manage the facts generated from Spatial and Temporal Reasoning components, and to add a memory to our system, we created a module based on a SQL Temporal Database. Apart from the conceptual perspective taking (as described in belief_manager), another capacity of the database component is the event based time management. This provides memory to the system. When facts are received, the time of detection is present in one of their variables. When the database manager detects a shift in a property, it updates the belief tables as explained in previous section, but it also records the event. To do so, we add a table filled with each event that occurs, recording the time when the property changes. This is a significant component of TOASTER as it is used for testing and debugging this situation assessment framework.

13.1 Implementation details

This module continuously reads facts from PDG, area_manager, agent_monitor and move3d_facts. All these facts are updated in the robot's belief state. Once the robot belief state is updated, it updates the other agent's belief state by the idea of conceptual perspective taking. Refer to Fig. 13 for better understanding.

13.2 Inputs

To maintain robot's own belief state, the information produced by perception, geometrical reasoning and inferences are collected by the database management

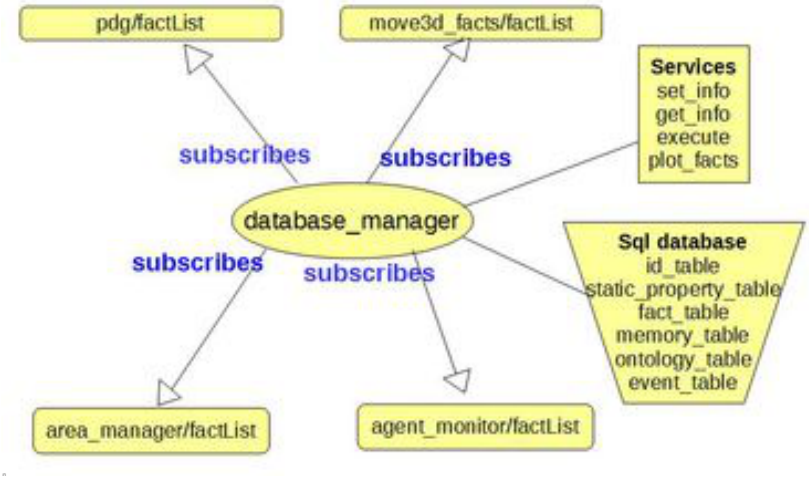

Figure 13: Implementation of database_manager module

module. Facts computed from area_manger, agent_monitor and move3d_facts are stored in this module.

13.3 Outputs

The output of this component is a sql database with fact table and memory table for each agent present in ID table. The fact table stores all the fact true for the agent at that time instant. While, the memory table stores facts that the agent has believed to be true. There is an event table as well to track the occurrence of events like when robot started moving and when it stopped moving. However, during an interaction, many properties that describes an object or an agent may not evolve in time and thus, be considered as static (color, name, age, ownership). To store these static properties, an extra table is present in the database and can be loaded at the start of the interaction, or filled online (if the robot acquire new knowledge on entities). For more details, check `https://github.com/Greg8978/toaster/blob/master/database_manager/doc/database%20doc.pdf`.

13.4 Services

Services of database_manager allows you to add entity, facts and events in respective tables and view all the tables. Its services are described below :

- **execute** - This service enables users to access the data in tables in various ways by putting different commands.

  Shell command:

  rosservice call /database/execute "command: " type: " facts: - property: ", propertyType: ", subProperty: ", subjectId: ", targetId: ", subjectOwnerId: ", targetOwnerId: ", valueType: false, factObservability:

0.0, doubleValue: 0.0, stringValue: ", confidence: 0.0, time: 0, timeStart: 0, timeEnd: 0 agent: " order: " areaTopic: false agentTopic: false move3dTopic: false pdgTopic: false"

Set the *areaTopic, agentTopic, move3dTopic, pdgTopic* as per your requirement. Using command "ARE_IN_TABLE" with agent and facts, it checks if the given facts are present in agent's fact table. "SQL" command with the query in order request message, executes the SQL query in database and displays the results. The command "EMPTY" with type "AGENT" for a given agent removed that agent from ID table and other tables related to this agent. With "ALL" type, it does the same for all agents. Using command "PRINT" with type "AGENT" prints tables of the given agent, while type "ALL" prints all tables in database.

- **get_info** - This service helps to get data from any given table. With the combination of type as "FACT" and subType as "ALL", it displays all the facts in the given agent's table at current time and before that also. While subType "VALUE", shows value of the given fact for the specified agent at current time. The subType "CURRENT" gives all the current facts of the given agent while "OLD" displays older facts. Similarly, combination of "ALL" subType with type as "EVENT", "ONTOLOGY", "ID" and "PROPERTY" displays respective tables from database. While subtype "VALUE" with all possible type gives values from respective tables. The request message looks like this :

  Shell command:

  rosservice call /database/get_info "type: " subType: " agentId: " reqFact: property: ", propertyType: ", subProperty: ", subjectId: ", targetId: ", subjectOwnerId: ", targetOwnerId: ", valueType: false, factObservability: 0.0, doubleValue: 0.0, stringValue: ", confidence: 0.0, time: 0, timeStart: 0, timeEnd: 0 reqEvent: property: ", propertyType: ", subProperty: ", subjectId: ", targetId: ", subjectOwnerId: ", targetOwnerId: ", valueType: false, factObservability: 0.0, doubleValue: 0.0, stringValue: ", confidence: 0.0, time: 0 id: 0 idString: " name: " entityClass: ""

- **plot_facts** - This service is developed with the purpose of examining the functioning of TOASTER in real-time environment. It plots the given fact for the given subject and target entity in specified time window. This service creates "fact.dat" file in your home directory. To plot the graph install gnuplot and run "gnuplot" command on terminal. In gnuplot terminal, run following commands:

  gnuplot¿ set yrange[-1:2]

  gnuplot¿ plot "fact.dat" using 1:2 with lines And you will see the desired plot.

  Shell command:

  rosservice call /database/plot_facts "subjectID: ", targetID: ", timeStart: ", timeEnd: ", reqFact: ""

- **set_info** -

  Shell command:

rosservice call /database/set_info "add: false infoType: " agentId: " facts: - property: ", propertyType: ", subProperty: ", subjectId: ", targetId: ", subjectOwnerId: ", targetOwnerId: ", valueType: false, factObservability: 0.0, doubleValue: 0.0, stringValue: ", confidence: 0.0, time: 0, timeStart: 0, timeEnd: 0 event: property: ", propertyType: ", subProperty: ", subjectId: ", targetId: ", subjectOwnerId: ", targetOwnerId: ", valueType: false, factObservability: 0.0, doubleValue: 0.0, stringValue: ", confidence: 0.0, time: 0 id: " name: " type: " ownerId: ""

13.5 Examples

As an example, if a Mug was on a table and the robot detects that the Mug is now in Bob's hand, it will remove the fact $<Mug\ isOn\ Kithen\_Table>$ from the belief tables of agents able to perceive the change, and add the event $<Bob\ pickUp\ Mug>$ in the event table. In memory table, we use as starting time the time when the agent noticed a property, and as ending time the moment when the agent detects a change in it. One of the application can be to understand when a human asks "Where is the mug that was on the table" by detecting in his memory table which mug she/he is talking about and looking in the current fact table where it currently is. In addition, the robot could even tell the related event that created the change: "It is now in the sink, Bob picked it up 5 minutes ago".

13.6 Future work and possible improvement

This module is one of the last implementation of TOASTER. More tests should be done to ensure the database is able to answer fast enough to the requests.

14 toaster_visualizer

To understand the situation and visualize the internal processes, a visualization module is rendering the names and 3d models at the position of the entities and the areas with their names, [4]. This visualization simplifies the debugging process and allows a faster setup of the framework parameters. This module is designed to create a simple 3d representation of toaster's physical elements in RVIZ.

14.1 Implementation details

Fig. 14 explains the implementation of this module. To use rviz with TOASTER, follow the configuration instructions at `https://github.com/Greg8978/toaster/blob/master/toaster_visualizer/doc/toaster_visualizer_doc.pdf`

14.2 Inputs

It uses the position of the entities published by the data gathering component and the areas published by the area manager component Toaster_visualizer use informations taken form toaster_msgs published on specifics ros topics to create

[4] http://wiki.ros.org/rviz

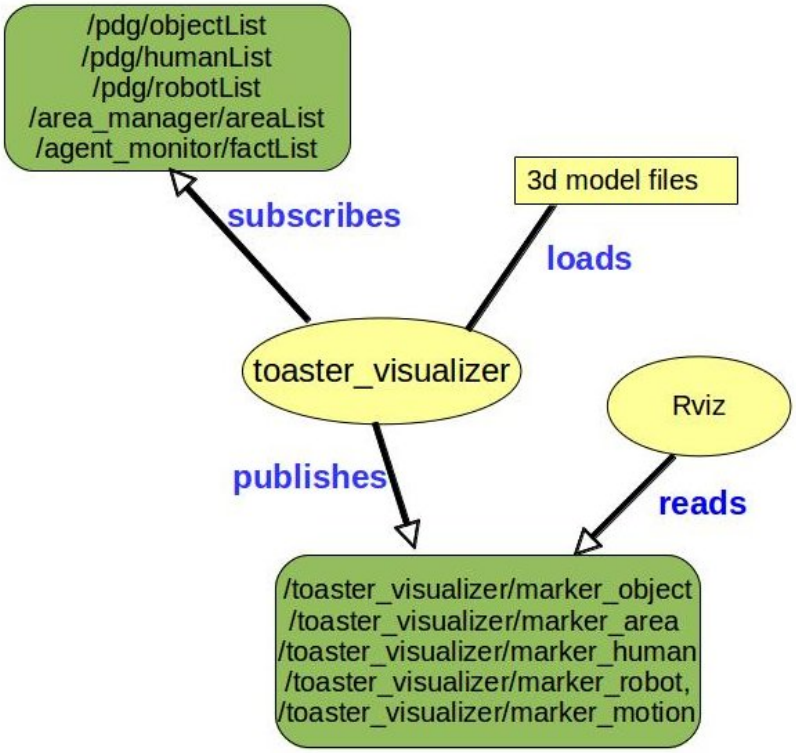

Figure 14: Implementation of toaster_visualizer module

markerArray. toaster_msgs concerned: AreaList, ObjectList, HumanList topics concerned: /area_manager/areaList, /pdg/objList, /pdg/humanList

14.3 Outputs

It publishes markerArray on topics like /toaster_visualizer/marker_area, marker_human, marker_robot, marker_motion and marker_objects viewable with RVIZ as shown in Fig. 15 and Fig. 16. For more details, you can check https://github.com/Greg8978/toaster/blob/master/toa

14.4 Services

This node has no services.

14.5 Future work and possible improvement

In a future work, we would like to improve this visualization by adding the rendering of other facts, such as the one generated by the agent monitoring component about the attention.

Figure 15: Data from simulated sensors in MORSE (right) and its representation in TOASTER (left)

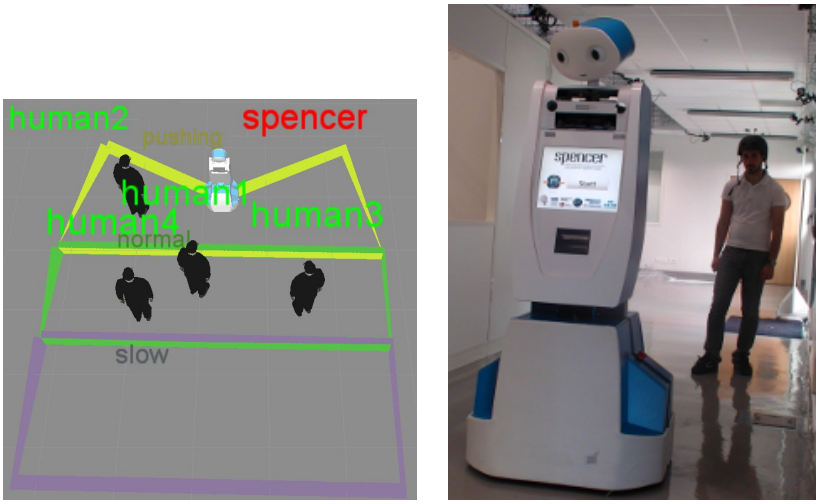

Figure 16: Spencer robot and its attached areas for guiding humans, in TOASTER (left) and in the real world (right)

15 Facts computation recapitulation

To help understanding and to give a global view of the facts computation, we give in the following table the recapitulation of all the facts computed in TOASTER along with the module computing it.

node	Property	PropType	Subproperty	subjectId	TargetId	stringValue	doubleValue	brief description
PDG	IsSeen	affordance	$\emptyset$	*obj.id*	$\emptyset$	true	$\emptyset$	Object is perceived from sensor(s)
PDG	IsInHand	position	object	*obj.id*	*joint.id*	true	$\emptyset$	From pdg service put_in_hand
area_man	IsInRoom	position	room	*ent.id*	*area.name*	true	$\emptyset$	If subject is inside room area
area_man	IsAt	position	location	*ent.id*	*area.name*	true	$\emptyset$	If subject is inside support area
area_man	IsInArea	position	location	*ent.id*	*area.name*	true	$\emptyset$	If subject is inside other area
area_man	AreaDensity	density	ratio	*area.name*	$\emptyset$	$\emptyset$	*EntArea/Ent*	compute if area.factType=density
area_man	IsFacing	posture	angle	*ent.id*	*ent.id*	true	*angle*	compute if area.factType=interaction
agt_mon	IsLookingTwd	attention	agent	*agt.id*	*ent.id*	true	$\emptyset$	angle cone from agent's head
agt_mon	IsMoving	motion	agent	*agt.id*	$\emptyset$	true	$\emptyset$	*speed* > 0.12 m/s
agt_mon	IsMoving	motion	joint	*jnt.name*	$\emptyset$	true	$\emptyset$	*speed* > 0.12 m/s
agt_mon	IsMovingTwd	motion	direction	*agt.id*	*ent.id*	true	*angle*	trajectory of agent is toward ent
agt_mon	IsMovingTwd	motion	direction	*jnt.name*	*ent.id*	true	*angle*	trajectory of *jnt* is toward *ent*
agt_mon	IsMovingTwd	motion	distance	*agt.id*	*ent.id*	true	*dist*	distance agt/ent decreasing
agt_mon	IsMovingTwd	motion	distance	*jnt.name*	*ent.id*	true	*dist*	*dist jnt*/ent decreasing
agt_mon	Distance	position	3D	*jnt.name*	*ent.id*	*distStr*	*dist*	distance *jnt*/ent decreasing

Table 1: Facts computed by TOASTER framework

References

[1] G. Milliez, M. Warnier, A. Clodic, and R. Alami, "A framework for endowing an interactive robot with reasoning capabilities about perspective-taking and belief management," in *Robot and Human Interactive Communication, 2014 RO-MAN: The 23rd IEEE International Symposium on*, Aug 2014, pp. 1103–1109.

[2] M. Fiore, H. Khambhaita, G. Milliez, and R. Alami, *Social Robotics: 7th International Conference, ICSR 2015, Paris, France, October 26-30, 2015, Proceedings*. Cham: Springer International Publishing, 2015, ch. An Adaptive and Proactive Human-Aware Robot Guide, pp. 194–203.

[3] M. Fiore, G. Milliez, and R. Alami, "Using agent belief awareness and context for intention recognition and proactive behavior," in *Proceedings of the 25rd IEEE International Symposium on Robot and Human Interactive Communication*, 2016, in review.

[4] G. Milliez, E. Ferreira, M. Fiore, R. Alami, and F. Lefèvre, "Simulating human robot interaction for dialogue learning," in *SIMPAR*, 2014, pp. 62–73.

[5] S. Lemaignan, M. Hanheide, M. Karg, H. Khambhaita, L. Kunze, F. Lier, I. Lütkebohle, and G. Milliez, "Simulation and hri recent perspectives with the morse simulator," in *Simulation, Modeling, and Programming for Autonomous Robots*. Springer International Publishing, 2014, pp. 13–24.