Contingent Planning for multi-medial Human-Pepper Interaction in a teaching assistant scenario

Lorenzo Nicoletti (1797464) - Olga Sorokoletova (1937430)
Leandro Maglianella (1792507) - Giorgia Natalizia (1815651)
Sapienza University of Rome

August 18, 2022

Abstract

The goal of this project was to create a framework where the robot acts as a teacher for children. The fact that, on some restricted tasks, social robots can achieve outcomes similar to those of human tutoring served as a motivation for choosing the educational domain. In particular, the proposed teaching scenario is focused on learning rules, fundamentals and advanced skills of basketball through adaptive delivery of learning material: personalized tests and lessons according to the level of experience of the interacting user. The robot we have chosen to perform our experiments is Pepper — a humanoid robotic structure with a wheeled mobile base for the motion and a tablet mounted on its torso. The robot is also equipped with a variegate set of sensors to perceive and interact with the surrounding environment. The Human-Pepper Interaction is built on top of a reasoning architecture based on Contingent Planning, implemented with a non-deterministic variation of PDDL able to navigate in the state space according to the value of predicates that can dynamically change during the interacting session. Our experiments were conducted both in simulation using Android Studio emulator and with a real Pepper. The related results along with the advantages and limitations of our approach will be discussed in details in this report.

The authors equally contributed to the project.

Contents

I. Introduction

During the passage of past years, the presence of robots in our modern societies has been rapidly growing time after time [1]. In Figure 1 it is shown the rising of the amount of operating industrial robots worldwide: in 2020 they were more than 3 million and today their amount keeps on increasing correspondingly.[1]

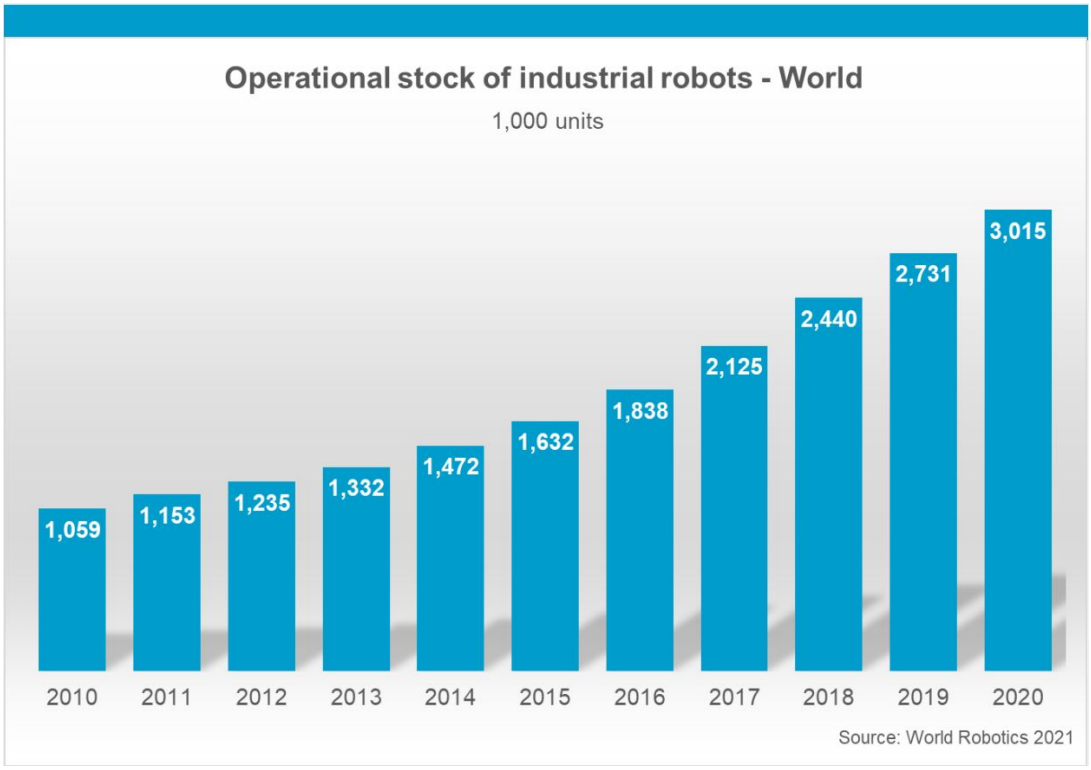


Figure 1: *Image showing the annual growth of active industrial robots globally employed during the past years.*

When the outbreak of this field first occurred, the major needs for industries laid in efficiency and speed and they found in robots a perfect candidate for performing simple and repetitive tasks. However, over time the robotics application domain evolved and now researches are conducted with the aim of developing new methods to delegate to robots increasingly complex and critical tasks, to be carried out even in private environments, which clearly requires some high-level reasoning skills with great precision and accuracy. For instance, we can consider the widespread use of robots in Amazon's fulfilment centres that manage the cataloguing and movement of the infinite amount of objects that the company sells and ships all over the world; or we could consider even more critical and important domains: many robots in fact find their application in hospitals and centres for the care of the elderly [2], still others are employed as teachers and support for children [3]. As we can imagine, in these kinds of tasks robots must be specifically designed so to be able to have the best possible interaction with humans: the efficiency of the task is not the most relevant aspect anymore but, as a matter of fact, new important ones related to social sciences as psychology, communications, relational and cognitive science must be now closely considered while evaluating robots' performances.

The main goal of this project consist in the development of an framework comprising effective and entertaining methods having the purpose of allowing robots to interact and reason efficiently

[1] As a further reference in addition to the figure, consider that industrial robots' popularity began approximately in 1970: in 1973 there were only 3 thousand active robots, in 1983 66 thousand, in 1993 575 thousand and in 2003 800 thousand [1].

with users about the basketball sport and its domain. We will be using Pepper (Figure 2), a human-like robot developed by *Softbank robotics* equipped with various devices and sensors as lasers, sonars, *RGB* and depth cameras. In our project, Pepper is supposed to be approachable and usable by any target audience, anyway we would suggest its use especially for children who are trying to start their basketball learning path as beginners. In the following of this report, we will first consider some of the relevant related works for our approach, then an in-depth documentation about it will be performed where we will consider our solution and its implementation. Finally, all the results we were able to achieve will be displayed.

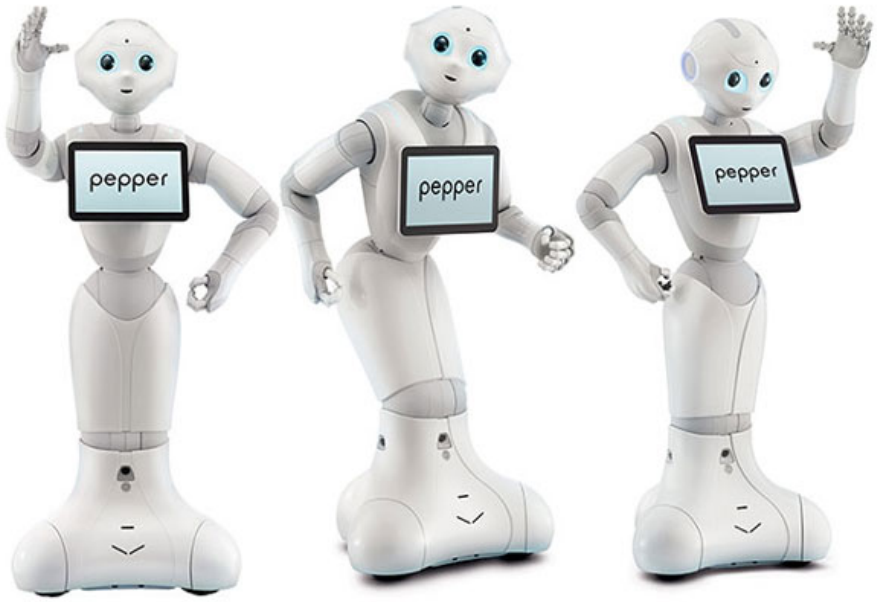


Figure 2: *Pepper robots showed in three different poses.*

II. Related Work

Let us consider some of the relevant studied paper, from which a lot of concepts were extracted and used as fundamental principles for our approach.

i. Human-Robot Interaction

Due to the reasons aforementioned in the Introduction section, methods are being built to allow constructive human-robot interactions and co-operations. In fact, robots can be used to carry out some individual tasks assigned by humans, for instance giving them some requested object, but can also be used to help in cooperative tasks where a human can demand a robot's action but the robot can ask for the human's help too: hence, in cooperative tasks robots have to be able to reason on human's feasible actions and how they can help themself based on the environment context [4]. Moreover, some studied crucial aspects about collaborations concern the fact that they have to be performed by robots while trying to ensure and understand the visibility and accessibility concepts regarding the used objects [5].

Approaches are also studied to ensure that robots try to maximise safety: the manipulation should always be *human aware* [6]. This means, for example, that robots should attempt to stay as far as possible from humans to avoid collisions while moving and to only approach them from the front. Moreover, they should also manage to recognise the *"category"* of the users interacting with them to act for their best comfort: for instance, if the current users are elderly, the robot will have to try to be even more cautious and to get closer to them [7].

In our project, the main ideas to develop a robot teacher come from [3] and [8], where a *RoboSapien* robot was used in a Taiwanese elementary school to help teaching children in English classes. In particular the authors highlight how a robot can be used in an educational environment (defining different categories as *Learning materials*, *Learning companions/pets* or *Teaching assistant*) and create different interaction models with the goal of both teaching and entertain the students in an efficient way.

As it is highlighted in [9], robots are often a natural choice when the aspects to be taught requires direct physical manipulation of the world, e.g. when *tutoring physical skills* such as basketball. Furthermore, authors of [10] expand the topic of developing robots aimed at helping children to excel at physical skills by pointing out that, in order to do this, the suggested approach is not to actuate on the students, but coach them through *hands-off modalities* such as verbal advice and demonstrations − exactly the approach we exploit in our project. On the opposite, another paper [11] (but still relevant in our project) targets the *Online Teaching* topic, focusing on methods for hand gestures and motions recognition and text and natural language understanding.

ii. Reasoning Agents

ii.1 Non-deterministic Planning

Teaching assistant scenarios inevitably introduce uncertainties, mainly present in the human side. User interacting with the reasoning robotic agent may indeed vary their answers according to their level of experiences about the discussed topics and, consequently, the teacher-robot should be able to guarantee different and customized learning routines that have to dynamically adapt to the specific session of interaction with the user.

This leads to a relevant branch of planning called *non-deterministic planning*, where action non-determinism is responsible of introducing uncertainty when navigating into the associated state space. More specifically, one or more actions are said to be non-deterministic when the resulting state (i.e. the state reached as the result of action execution) is not uniquely determined due to the presence of multiple successor states.

In [12], the authors define a *non-deterministic planning domain* $\mathcal{D}$ as a tuple

$$\mathcal{D} =< Act, Prop, S, s_0, f > \tag{1}$$

where:

- Act is the finite set of actions;
- $Prop$ is the finite set of propositions;
- $S \subseteq 2^{Prop}$ is the set of states;
- s_0 is the single initial state;
- $f : S \times 2^S$ is the state-transition function and defines the non-determinism of the planning domain.

Even if the problem faced in that paper was definitely more complex than ours, with goals expressed in *Linear Temporal Logic* and reduced to *Linear Temporal Logic Strong Cyclic Planners*, we were inspired by their combination of non-deterministic domains and PDDL, that we have indeed adopted and adapted for our specific use case.

ii.2 Contingent Planning for partial observability

Another important property of our planning problem is represented by its *partial observability*, meaning that the entire space is not fully visible due to one or more propositions whose value is unpredictable a priori.

Two fields of planning efficiently overcome the limitations introduced in partially-observable domains, *Conformant Planning* [13, 14] and, even more properly, ***Contingent Planning*** [15]. The former extends classical planning by adding the ability to model uncertainty of initial states expressed as a CNF formula, while the latter is an expansion of the former and introduces the possibility to treat partial observability using the so-called *observation actions* − actions, able to "monitor" the propositions' values at runtime. Eventually, a tree-shaped plan is created offline.

III. Solution

i. Human-Robot Interaction

The architecture of the solution is hierarchical and mainly represented by the six interacting modules: 1) **main** module; 2) **functional** module; 3-4) two **plan executors** and 5-6) two **collections of utilities** for interfacing two different approaches to Pepper programming (***pepper_tools*** interface and ***MODIM***). The flow of information passing between different modules is depicted in Figure 3.

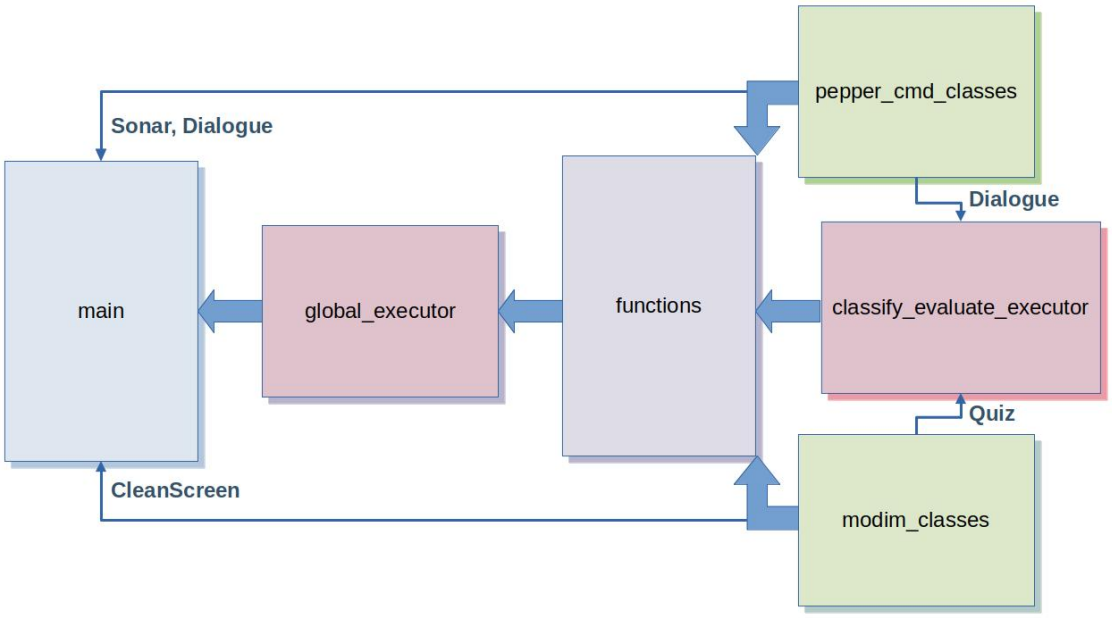


Figure 3: *Architecture of the solution for the HRI part of the project. Arrows represent the direction of information exchange (solid arrows — import of the entire module, narrow arrows - import of the specific functionalities of the module).*

i.1 Main Module

The main module (blue block in Figure 3) is placed at the highest level of hierarchy and it is responsible for the default behavior of the robot, i.e. behavior running on the *background* during the entire period of robot's activity when no signals are detected, *pre-interaction* behavior and handlers of the two specific signals: *start* of interaction and *end* of the period of activity.

i.2 Plan Executors

Two plan executors (red blocks) provide a bridge between pre-computed offline outputs of the corresponding reasoners and calls of the actual Pepper's services undertaken online during interaction with the user. In other words, they are the *custom-built proxies* that ensure the actual action execution.

The `global_executor` encapsulates function calls for the actions managing general logic of interaction. It is paired with a global problem search tree drawn in Figure 14 and described in details as the result of the Reasoning Agents part of the project. Similarly, `classify_evaluate_executor` is a proxy for the four other planning problems: user classification in Figure 15 and three level

evaluations[2] in the Figure 16.

i.3 Functional Module

The functional module (purple block, named functions) contains the definitions of the seven functions, corresponding to a seven unique action names in a global planner search tree to be called from inside the `global_executor`. As it could be seen in Figure 3, this module aggregates all the lowest level utilities foreseen by the framework's architecture.

i.4 Human-Pepper Interfaces

Two modules of software architecture, called `pepper_cmd_classes` and `modim_classes` (green blocks), represent two different ways to perform communications between our *designed application* and *robot services* of Pepper programming – using `pepper_tools` and `MODIM` interfaces, respectively. Both of them collect a number of classes reported in Figure 4 and are responsible for implementation of a specific unitary functionality each.

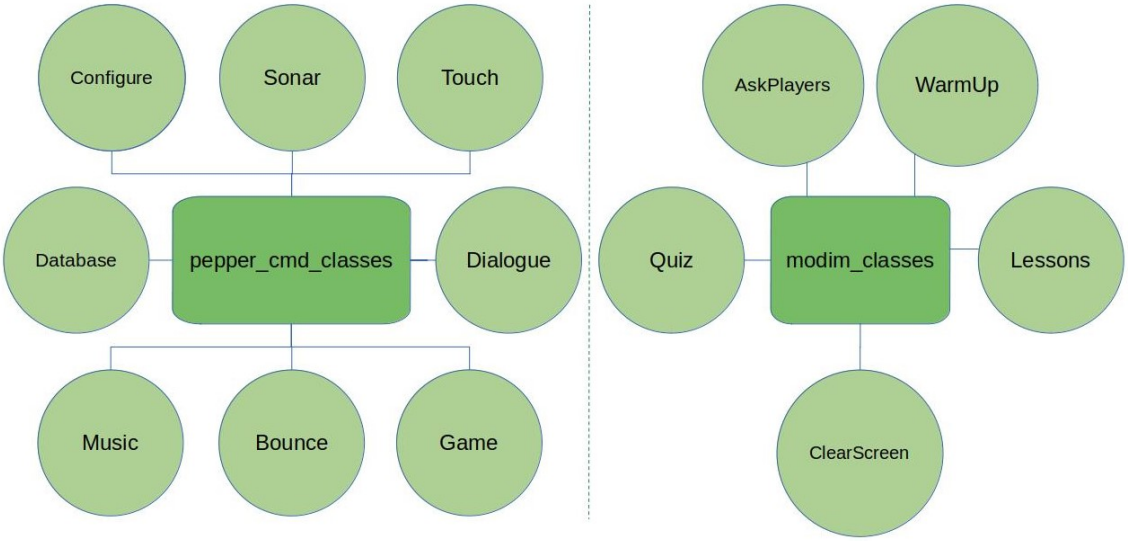


Figure 4: *Set of classes implemented in the project using `pepper_tools` (left side before dashed line) and `MODIM` interfaces (right side).*

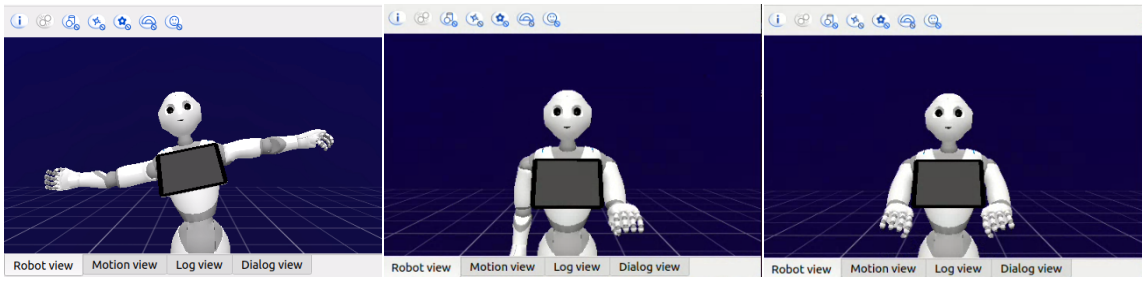
1. Pepper_cmd_classes. In more detail, the `pepper_cmd_classes` group of eight classes implements Pepper's perception and action modalities that do not involve interaction with the tablet.

Five of them are the core modalities, and the other three are the break handlers. The motivation that we pursued introducing the break handlers was to account for a specificity of the target audience, i.e. children. In addition to the delivery of educational materials, inclusion of the strategy to promote learning in robot-child tutoring is needed. In our framework, to increase the level of child's engagement, the following option is provided: user can launch a break handler at any moment during the lessons. Overall, the classes implemented with `pepper_tools` interface are:

- Core: *Configure()* – initial set-up of the robot;
- Core: *Sonar()* – people detection;

[2]For the sake of short notation later on we say "quiz" meaning "level evaluation"

- Core: *Dialogue()* – conversation support;
- Core: *Touch()* – touch monitor and reaction;
- Core: *Database()* – internal data storage and operations on it;
- Break handler: *Music()* – Figure 5a – Pepper is playing a song and dancing;
- Break handler: *Bounce()* – Figure 5b – demonstration of basketball dribble performed by the robot with a randomly chosen hand;
- Break handler: *Game()* – Figure 5c – hand game with a following scenario: robots shows two hands and asks the user to guess in which one it hides an invisible candy, the game continues until the candy is found.



(a) *Music()* **(b)** *Bounce()* **(c)** *Game()*

Figure 5: *Break handlers snapshots taken during execution in Android Studio.*

2. Modim_classes. MODIM client-server architecture is employed to manage tablet part of robot-user interaction exclusively. In particular, `modim_classes` module covers implementation of the four classes, corresponding to the eponymous actions in a global plan, and a standalone class, responsible for the post-interaction processing of the tablet state:

- Action: *Quiz()* – corresponds to evaluation part of `CLASSIFY_EVALUATE_USER` action in Figure 14 (defines it), i.e. provides definitions of the actions for the quiz problem search tree depicted in Figure 16 since we model the hierarchical relationship between the planners;
- Actions: *AskPlayers()* – defines an `ASK_PLAYERS` action in Figure 14, which allows the users to choose his/her favorite NBA season leader and is intended to get them moe involved into the interaction;
- Action: *WarmUp()* – defines a `WARM_UP` action in Figure 14 that is a safe start of the part of interaction where user is mastering physical skills by demonstrating warming up exercises to help to reduce muscle soreness and lessen the risk of injury during the training;
- Actions: *Lessons()* – defines a `LESSONS` action in Figure 14 that shows to users YouTube video tutorials according to the classification result obtained from a corresponding leaf of classification planning problem (Figure 15). All the lessons presented in the different videos on YouTube are self-produced and specifically designed for our approach. It encounters five different video lessons with a possibility to skip or trace back to any of them, or interrupt for a break and then recover with lessons immediately after break is finished, or quit the lessons and hence finish the interaction session. When all five videos have been watched, course is considered to be completed, and user is presented with a choice to finish the course or re-watch some of the previous lessons;
- Standalone: *CleanScreen()* – cleans up all the modalities that are currently being executed by the tablet and returns the screen to its default state (provided in Figure 6). This is needed to have the default screen on every time when a new user approaches.

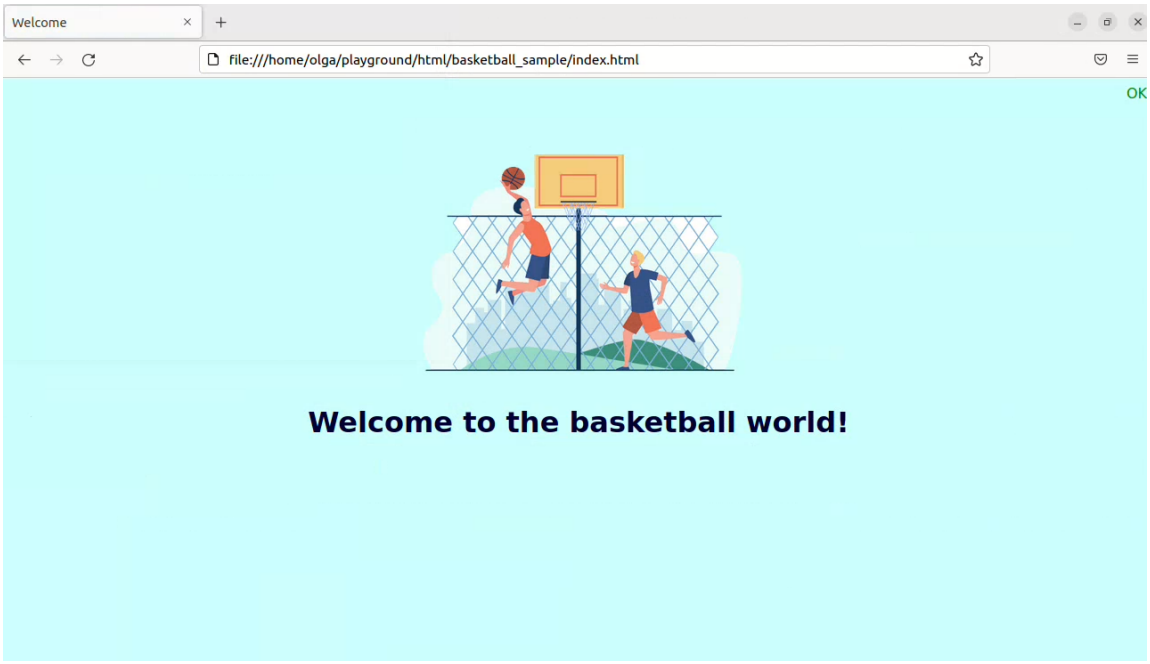


Figure 6: *Default tablet screen.*

ii. Reasoning Agents

As mentioned before, the nature of our reasoning task requires special care to simultaneously take into account both action non-determinism and partial-observability. A practical solution is given by **Contingent-FF**[3], a modification of the state-of-the-art planner, *Fast-Forward (FF)*, that enables Contingent Planning. In particular, it is able to model observations by defining *observation actions* to monitor the varying values of specific predicates. The output plan will be a tree-shaped solution that starts from the root node and creates new branches according to the value of such predicates. The search tree is built to incorporate all possible paths from the root to each leaf node. Meanwhile, only one of the paths will be executed at runtime, depending on how the observable variables will be resolved.

Contingent-FF is based on PDDL and has two specific instruments to monitor observations represented by the `:observe (<pred>)` construct to be combined with the `(unknown (<pred>))` item, given a partial observable predicate `<pred>`.

All the observation actions are clearly defined in the PDDL domain, while the associated `unknowns` are specified in the PDDL problem.

When the planner is executed and a feasible solution is found, it creates a tree-based plan by defining a new branch for every observation action to be executed to reach the goal. For instance, assuming we are at depth `i` of the tree and we reach an observable action, the output plan will be composed by the following line:

```
i||0 --- ACTION_NAME <PARAMETERS> --- TRUESON: i+1||0 --- FALSESON: i+1||1,
```

meaning that `ACTION_NAME` defines two possible next states depending on the observed predicate's value that can be either `True` or `False`; the successor is then defined as `TRUESON: i+1||0` or `FALSESON: i+1||1` accordingly.

[3]The Linux executable we have used for this project is freely available from the official website at https://fai.cs.uni-saarland.de/hoffmann/ff/Contingent-FF.zip

The potential effectiveness of Contingent-FF is immediately perceived when focusing on the objectives of our use case. Since we aim to create a teaching assistant framework where the robot is able to "switch" actions and choose the best successor according to user's previous answers, we simply need to define observation actions to check the value of unknown predicates that can be dynamically changed by the decision of the user during the interaction. As it will be analyzed in the following section with more details, the reasoning part of our project is based on the simple yet well-suited idea to strictly match the possible inputs from the user to the `:observe` actions/`unknown` predicates in PDDL.

IV. Implementation

The project has been first developed in a simulation environment using Pepper SDK emulator provided by Android Studio[4] and then tested on a real Pepper robot. As for the tablet part, MODIM client connection with local server and the Firefox browser have been used. All the source files of the project together with the instructions to run them through the local connection and video demonstration of the interaction session are collected in our GitLab repository: EAI2_project.

i. Human-Robot Interaction

This subsection is devoted to the detailed description of the implementation of each module of the diagram in Figure 3. Let us consider them one by one in a sequential order starting from the highest level of architecture.

i.1 Main Module

SOURCE FILE NAME: `main.py`.

IMPLEMENTATION: This module starts the period of activity of the robot and can be stopped at any moment by sending to the program a `KeyboardInterrupt` signal (the corresponding signal handler is provided). An infinite loop is run during the period of activity, importing `Sonar()` functionality from the `pepper_cmd_classes` and using it to monitor whether someone approached the robot. An approach is only considered to be performed if the person has stayed in the vicinity of the robot (defined as a circle with a 1 meter radius) for more than 3 seconds. Both rear and front sonars are involved, but different behavior is implemented depending on which one of them the robot used to perceive the presence of the human: if the person approached the robot from its back, Pepper uses `Dialogue()` functionality from the `pepper_cmd_classes` to ask the person to come in front and then continues monitoring, otherwise, it greets the person and launches the interaction by calling the global executor module. After and only after the interaction with the current user is finished, Pepper continues listening the sonar signals in order to detect a new approach.

i.2 Plan Executors

SOURCE FILE NAME: `global_executor.py` and `classify_evaluate_executor.py`.

IMPLEMENTATION: Once the required plans are generated by the reasoner and saved to the text files, we need to read them, navigate through the search tree and find the correct successor state depending on the specific branch of the tree we are following at runtime. This file parsing procedure is implemented by plan executor scripts where, starting from the root action, the inner nodes are traversed in a depth-wise fashion until a leaf node is reached.

As it is explained in the previous section about the solution for the reasoner, in the output files, generated for some chosen planning problem, each action at its depth is associated with a unique ID of the form `depth||child` (for instance, the root has ID `0||0`). Both plan executors, always starting from the first root node ID, perform the associated action and update the value to the next action's ID (jumping to such corresponding line in the plan text file). In other words, the plan executor scripts act as action schedulers, whose goal is to individuate the appropriate successor while navigating into the search tree.

Finally, the routines of the actions are defined in different modules and files that are called by the executor:

1. In case of *global executor*, the routines are collected in a `functions.py` module and are called using `options` dictionary imported to the executor module;

[4] Although in some cases the currently being designed functionality did not require the visual feedback. In these cases, NAOqi server has been executed directly from the docker container.

2. In case of *classification-evaluation executor*, there are only two actions to be called: `ASK_QUESTION` or `ASK_QUIZ` (to implement classification or evaluation, respectively). The former is performed through the oral communication and imports the `Dialogue()` from `pepper_cmd_classes`, meanwhile the latter involves interaction with Pepper's tablet and imports `Quiz()` from `modim_classes`. Hashmaps are used to store the plan paths in internal storage, announcements to inform the user about the classification result and Q&As for both classification and quizzes.

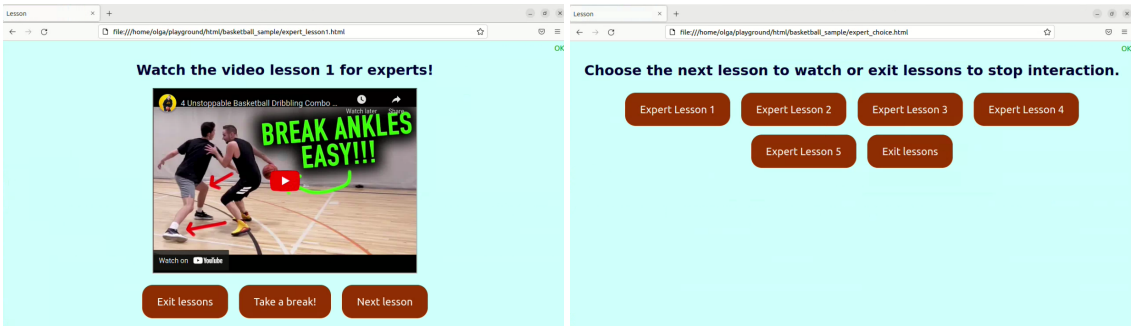
All the dictionaries are serialized and deserialized using the `Python` library `Pickle`, with the purpose of maintaining an efficient static data storage.

i.3 Functional Module

SOURCE FILE NAME: `functions.py`.

IMPLEMENTATION: The functional module imports the classification-evaluation executor and all the possible classes from both `pepper_cmd_classes` and `modim_classes`. It consists of definitions of the seven functions corresponding to the seven unique action names in a global plan that can be seen in Figure 14:

- `ask_start()` — uses `Dialogue()` and `Touch()` from `pepper_cmd_classes` to clarify whether the approaching user is indeed willing to start an interaction session and did not just approach for the sake of curiosity. It proposes to provide a confirmation by touching the robot's head and launches a listener for the corresponding sensor for a limited amount of time. As the result, the resolution between the most-right branch of the global search tree and the other two is obtained;
- `check_new()` — uses `Database()` from `pepper_cmd_classes` to request the internal storage if the user has already been interacting with the robot before (in which case some of the actions should be skipped since they are intended to be performed only one time per user) or this is his/her first interaction session. For already known users, the robot derives information about their classification result because it is later needed to open the lessons of the corresponding difficulty. As the result, the disambiguation between the two left branches of the global search tree is obtained;
- `classify_evaluate_user()` — uses `classify_evaluate_executor()` and `Database()` from `pepper_cmd_classes` to define the class of the user, evaluate his/her level of basketball knowledge inside the class and create a record in a database that maps user's ID to the classification result;
- `ask_players()` — uses `AskPlayers()` tablet interaction from `modim_classes`;
- `warm_up()` — uses `WarmUp()` tablet interaction from `modim_classes`;
- `start_lessons()` — the recursive function that uses `Dialogue()`, `Music()` and `Game()` from `pepper_cmd_classes` and `Lessons()` from `modim_classes`. It first informs the user about which level of difficulty for the lessons are being opened and then opens the lessons, which logic is described in the Solution section. For new users the first lesson of the corresponding class is launched (e.g. an expert user will see the screen as in Figure 7a), otherwise the user is presented with a choice of lesson to be launched as in Figure 7b.
  To provide a well-functioning switch from the tablet interaction to the break handler, the information about the button pressed by the user to choose the break modality is read from a file called `status.out` and specifically maintained for this purpose. After the break, `start_lessons()` is recursively called but now with the *old user* flag so that the screen with the lesson's choice would be opened as opposed to always the first lesson. The return from the recursion is organized through the same `status.out` file and is *event-based*: the returning event happens when the user presses the specific button named `Exit lessons`;
- `terminate()` — uses `Dialogue()` from `pepper_cmd_classes` and `CleanScreen()` from `modim_classes` to communicate to the user that the current interaction session is finished and return to the initial configuration of the tablet.



(a) *New user* **(b)** *Old user*

Figure 7: *The screen presented to the user after pressing the* `Start lessons` *button.*

i.4 Human-Pepper Interfaces

Since the "labor division" paradigm is exploited to entrust tablet and out-of-tablet interactions to the different entities, the tablet is always controlled by the `MODIM` interaction manager and `pepper_tools` tablet is never used as well as `im.robot.<fn>` pointer to `pepper_cmd.robot` is never used from inside the `MODIM` to manage out-of-tablet HRI devices. Therefore, implementation of two parts can be described separately as it is done later in a subsection.

However, the utilities of both collections are organized as classes with specific methods to satisfy the specific needs of other modules. In turn, class methods of each class perform calls of elementary library routines aggregated in Table 1. In case of `pepper_tools` the routines that address different robot services are listed, and for the `MODIM` — the client-side interactions.

`pepper_cmd.robot.<fn>()`	`im.<fn>()`
`setAlive`	`init`
`tts_service.setParameter`	`ask`
`start(stop)SensorMonitor`	`display.loadUrl`
`sensorvalue`	`execute`
`say, asr`	
`getPosture, setPosture`	
`normalPosture`	
`session.service("ALAudioPlayer")`	
`motion_service.stopMove`	

Table 1: *Lists of library tools used to implement the* `pepper_cmd_classes` *(left column) and* `modim_classes` *(right column).*

1. Pepper_cmd_classes.

- *Configure()* — this class has only the constructor, and the object of this class is created in the constructor of every other `pepper_cmd_classes` class, because that allows to re-define speed of speech and if the Pepper acts as alive or not according to the current needs;
- *Sonar()* — the class has a recursive method called `listen`, that implements person detection with temporal filtering. The sensor monitor is listening both front and rear sonars, and as an output method returns the corresponding location of the detected person. The time parameter for the temporal filtering is included into method parameters and can be set to any desired value as well as the sonar threshold;

- *Dialogue()* – has two methods intended to provide the support for a two-sided conversation: `say` and `listen`. The former has a parameter called `require_answer` which is important for disambiguation if the phrase the robot is saying is meant to be a statement or a question. In the second case, `listen` method is called and repeats recursively until the answer is perceived by the robot (in other words, `None` returns, that could happen due to the `asr` timeout, are not accepted). This methods takes as input `vocabulary` and `timeout` matching to those of library `asr`. The following tokens are used in the vocabularies throughout the interaction flow: "yes/no" for the user classification, "left/right" in a hand game handler, various names and family names to perform user identification and registration in a database table;
- *Touch()* – this class is again forearmed with two methods: `change_pose` and `monitor_touch`. The `change_pose` method originally was developed to provide the Pepper's touch reaction in a form of movement (e.g. when users touches the head of the robot to confirm the start of interaction in `ask_start` action, robot moves its head up to show the user that a touch was detected by the touch sensor `HeadMiddle`). However, later the scope has been expanded to perform any other desired change of posture (e.g. same method is used to perform Pepper's dancing behavior during the music break handler execution). This method expects as input the current robot pose, as well as a list of indices of joints in a `jointNames` list whose values are requested to be changed and a list of the new joint values. As result, the robot's pose is changed accordingly to these values.
  The `monitor_touch` starts the monitor of a head touch sensor for a desired period of time given in seconds as input parameter (`monitoring_time`) of the method (in our application it is set equal to 20 seconds), and outputs a Boolean result depending on whether the touch was detected or not during the monitoring time;
- *Database()* – the class populated with the most methods, it implements our approach to organize the robot's data storage and to support the number of operations on it. The database itself is a `csv` table called `registered_users` and including three columns: `Name`, `Family Name`, and `Classification Result`. It is created (when it does not exist yet) by the `create_database` method. The `interview_user` method is responsible for collecting the information about user's identification data to be filled in the first two fields of the table. This method is called from inside the `detect_user`, which, in turn, given name and family name of the user scrolls down through the existing records in the database and checks if the corresponding record already exists. If it does, the robot performs some greetings while pointing out that it recognized the user and the method returns the confirmation to the outer function that called it (`check_new`) together with the record itself (guaranteed to be full, because the classification result was filled in during the previous interaction session). If otherwise, the user's record does not exist, Pepper greets the new user and registers him/her into the database by calling `register_user` method that adds to the table record with corresponding name, family name and empty string as a classification result. Finally, `update_result` method is provided to fill in the empty column with the classification result at the later stage when `classify_evaluate_user` is executed.
- *Music()* – the class implements a custom "play and dance" kind of behavior. It takes a path to a music wave as an input to the constructor (in our application the particular one is used, but could be any song in a `wav` (uncompressed) format). In charge of the constructor is to call an intrinsic auxiliary method `get_duration` that measures the duration of a given wave. The main method of the class is called `play`, and its functionality is to exploit audio player to play the music wave while performing a sequence of four moves in a loop up to obtained duration in a synchronized fashion so to simulate a Pepper's dance.
- *Bounce()* – provides robot with ability to demonstrate one of the basketball exercises (dribbling, i.e. bouncing the ball) physically as opposed to in media format. The class has three methods: `bounce_left`, `bounce_right` and `bounce_both` named respectively after the

hands using which the bouncing is performed (`mode` parameter in the class constructor). In methods, the sequence of moves that looks like a bouncing with a corresponding hand is executed the `num_bounces` times (set to 3 by default in our application).
- *Game()* – in a hand game Pepper puts its arms as in Figure 5c, asks user to guess in which one an invisible candy is hidden and waits for a spoken answer ("left" or "right"). The guess is done using a randomizer and the developer can see the correct answer in the terminal window. Guess attempts continue until the correct answer is given, after which the robot starts a dribbling demonstration with a randomly chosen hand. The hand game itself is performed by the `play` method of the class, meanwhile `interaction_handler` is responsible for sequential execution of a hand game and bouncing while communicating with the user.

2. Modim_classes.

`MODIM` interface provides a couple of alternative ways to implement the *screen switching*: one of them is by calling a particular `MODIM` action containing definition of the buttons inside it through the *Interaction Manager* and then executing the response of this action again through the *Interaction Manager*, and another one is by loading the desired `HTML` page that has definitions of the buttons and `onclick` events in its layout. In the simpler classes of `modim_classes`, such as `AskPlayers()`, `WarmUp()` and `CleanScreen()`, only the first way has been used, meanwhile `Quiz()` and `Lessons()` are more complex and exploit a mixed approach.

All the classes are organized as a constructor and one or more other methods intended to be called through the `run_interaction` `MODIM` client tool. The constructor of each class is a `MODIM` client object that first connects to a local `MODIM` server and then runs one of the interactions accordingly to a `mode` parameter of the constructor if the class considers its presence.

Whenever a `MODIM` action is executed through the `im.ask()` option, `timeout` is set to be equal -1 so user could have an infinite amount of time to choose the button. The only exception from this design approach is a `WarmUp()` class, where the warming up exercises are demonstrated through the `GIF` modality which is executed during exactly the `GIF`'s duration.

Follows some more in-depth details about the two most sophisticated classes:

- *Quiz()* – contains six quiz interactions: one per screen corresponding to a question of evaluation quiz, three interactions loading `HTML` screen with the result of evaluation: `good`, `medium` or `bad`, and one initializer to announce the beginning of the quiz. Which screen will be called at a particular timestep is decided in the `classify_evaluate_executor` which works according to the following logic: every time a new successor state is encountered in Figure 16, a new object of the `Quiz()` class is created with a corresponding `mode`. Therefore, the next screen to be displayed is not encoded inside the `MODIM`, but is resolved at the reasoning level by the family of quiz planners. A special file, called `quiz_answer.out`, is used to return information about the pressed button from the `MODIM` class to a `classify_evaluate_executor`;
- *Lessons()* – contains only six interactions: two per class of users: `BEGINNER`, `INTERMEDIATE` or `EXPERT`, where one of two is an `HTML` with a first lesson of the corresponding course and another one is a screen with a choice of any lesson of the corresponding course as it is presented in Figure 7. Navigation between the lessons (proceeding with the next lesson or returning to a previous one) is arranged through the definitions inside the `HTML` layouts exclusively. Unless `Take a break!` or `Finish lessons` button is pressed, in which case respectively `break.html` or `finish.html` is loaded, a signal to `status.out` is sent and the corresponding out-of-tablet handler is launched. The lessons themselves are the YouTube basketball video tutorials provided through `iframe` modality of `HTML` layout.

ii.1 PDDL and Contingent-FF

As explained before, our need to model uncertainty can be easily satisfied by creating observation actions for the Contingent-FF executor in PDDL. In order to keep computational costs low and better organize the reasoning flow, we have designed two different domains and several associated problems. In particular, the implemented PDDL domains are:

- *global domain*, where the whole structure of the interaction is built as a sequence of actions defining the successive steps of our Human-Pepper Interaction pipeline,
- *classification domain*, where the reasoning agent is able to classify the user according to its level of experience and, consequently, perform the appropriate set of quizzes to evaluate him/her theoretical level of basketball skills.

Predicates and actions for the global domain are grouped in Table 2 and Table 3, respectively. Due to the a-priori uncertain identity of the interacting user, two `unknown` fluents have been defined to first get to know whether the human in front of the robot is actually willing to start the learning procedure and then to check if the user is new, i.e. interacts with the robot for the first time. These predicates are observed in the related observation actions, `ASK_START` and `CHECK_NEW`.

Predicate	Description	Unknown
`wantStart`	True if user wants to start interacting.	✓
`newUser`	True if the user is new.	✓
`classifiedUser`	True if the user has been interviewed and has done the quiz.	
`players`	True if the user has chosen his/her favourite player.	
`warmUp`	True if the user has completed the warm up.	
`lessonDone`	True if the user has completed the lessons.	
`interactionDone`	True if the interaction can be terminated.	

Table 2: *Global domain predicates.*

Action	Preconditions	Observations	Effects
`ask_start`	-	`wantStart`	-
`check_new`	`wantStart`	`newUser`	-
`classify_evaluate_user`	`newUser`	-	`classifiedUser`
`ask_players`	`classifiedUser`	-	`players`
`warm_up`	`(players)` or `(not (newUser))`	-	`warmUp`
`start_lessons`	`warmUp`	-	`lessonDone`
`terminate`	`(lessonDone)` or `(not (wantStart))`	-	`interactionDone`

Table 3: *Global domain actions.*

If the user is indeed new, the `CLASSIFY_EVALUATE_USER` action is executed. This is implicitly linked to the second domain we have defined and is responsible for the performing of user classification and level evaluation. User classification is intended to understand if the user is an **expert, intermediate or beginner** in the basketball field by asking three personal questions. Meanwhile, user evaluation is a testing user's theoretical knowledge about the sport by issuing one of the three quizzes corresponding to the previous classification outcome and producing a new

outcome which detects if the user has **bad (low), medium or good (high)** level of knowledge inside his/her class of users. More details about the implementation can be found in Table 4 and Table 5 containing the predicates and actions that are needed to model the `CLASSIFY_EVALUATE_USER` sub-problem inside the global interaction.

Predicate	Arguments	Description	Unknown
`correctAnswer`	`?x - question`	True if user correctly answers to question `x`.	✓
`classificationDone`	`-`	True if the classification/interviewed is completed.	
`correctQuiz`	`?x - quiz`	True if user correctly answers to quiz `x`.	✓
`quizDone`	`-`	True if the quiz session has been completed.	

Table 4: *Classification domain predicates.*

Action	Preconditions	Observations	Effects
`ask_question`	`(not (classificationDone))`	`correctAnswer ?q`	-
`classify_beginner`	`(not (classificationDone))` and the three question must be different.	-	`classificationDone` when all the three questions were answered wrong.
`classify_intermediate`	`(not (classificationDone))` and the three question must be different.	-	`classificationDone` when at least one question was answered correctly and one wrong.
`classify_expert`	`(not (classificationDone))` and the three question must be different.	-	`classificationDone` when all the three questions were answered correctly.
`ask_quiz`	`classificationDone`	`correctQuiz ?q`	-
`evaluate_bad`	`(classificationDone)` and the three quizzes must be different.	-	`quizDone` when all the three quizzes were answered wrong.
`evaluate_medium`	`(classificationDone)` and the three quizzes must be different.	-	`quizDone` when at least one quiz was answered correctly and one wrong.
`evaluate_good`	`(classificationDone)` and the three quizzes must be different.	-	`quizDone` when all the three quizzes were answered correctly.

Table 5: *Classification domain actions.*

In this case, again, we have two `unknown` used to check the correctness of user's answer both for generic questions and for the quiz and the associated `ask_question/quiz` – to check the uncertain predicates' values. The other actions are employed to individuate the level of experience (beginner, intermediate or expert) and the evaluation of the completed quiz (bad, medium or good).

The PDDL construct

```
:effect (when (<antecedent>) (<consequence>))
```

was found to be particularly efficient in this case for choosing the correct outcome according to the correctness (or inaccuracy) of the answers of the user.

We have created one problem for the *global domain*, whose goal is simply `(interactionDone)`, that is reached by following the steps in the interaction pipeline: after user's identification and classification, the robot asks the favourite player, starts a warm-up session, shows some video lectures explaining fundamentals of basketball at different difficulties according to the user's experience and finally terminates the interaction.

For the *classification domain* instead, we have divided the task into four sub-problems: one to actually perform classification and to output the class that user belongs to, and the other three to perform quizzes, one for each of the three difficulty classes we have.

Consequently, once Contingent-FF takes as input the specified PDDL domain and problem, it creates a plan whose structure is a binary tree containing all possible paths in order to comprehend all the possibilities we have modelled as an interaction session. The plans are generated offline and only one branch among the feasible solutions will be traversed during the interaction, according to user's decisions and answers that we observe.

V. Results

i. Human-Robot Interaction

The experiments[5] were made both in simulation, using Android Studio and Pepper SDK, and in a real world situation with the robot Pepper. In order to evaluate the project all the different classes has been tested, starting from the approaching phase and then exploring the interaction implemented and all the possible behavior.

i.1 Pre-Interaction Phase

Since in the class `Sonar()` has been implemented different behaviours we tested them all in several trials. Both In simulation tests and in real world everything showed up to act properly. In all attempts, the subject that enters the robot's range of action is correctly detected but, being a shorter time than the interaction time, the robot listens waiting for a longer interaction.

```
Alive behaviors: True
Alive behaviors: True
Starting monitor, to stop robot send KeyboardInterrupt signal.
Alive behaviors: True
Waiting people...
Person approached. Measuring time...
Person approached, but then left.
Waiting people...
Person approached. Measuring time...
Person approached, but then left.
Waiting people...
```

Figure 8: *Person approaching Pepper for less than three seconds.*

The second set of tests performed were used to demonstrate the appropriate functioning of a complete interaction with the robot, starting from a detection by the rear sensor longer than three seconds and a subsequent detection by the front sensor. After the frontal detection the real interaction can begin. As we can see in Figure 9 Pepper identifies the person approaching him and starts the correct routine.

```
Waiting people...
Person approached. Measuring time...
Person stayed for more than 3.0 seconds.
Alive behaviors: True
Say: Who is behind me? If you came to me, come please in front so I could see you.
Person was detected by rear sonar.
Alive behaviors: True
Waiting people...
Person approached. Measuring time...
Person approached, but then left.
Waiting people...
Person approached. Measuring time...
Person stayed for more than 3.0 seconds.
('Action: ', 'ASK_START')
Say: Hello! I am Pepper, and I can teach you more about the basketball.
Say: Touch my head if you would like to start the interaction.
Waiting a touch to start during 20.0 seconds...
```

Figure 9: *Person approaches Pepper and then stands in front of it.*

[5] A demonstrative video showing our executions can be found here (https://www.youtube.com/watch?v=97sCw_6JNRI).

i.2 Interaction Phase

The second phase of the experiments is the interaction one, which has the goal of verifying all the possible classes that involve an exchange between the person and the robot. Following the flow of the execution, the next class tested is the `ask_start` and the related `Dialogue()` and `Touch()`. Pepper waits for 20 seconds for the user to touch its head to confirm to begin an interaction, and just quit the current conversation if no touch is detected. Instead, if the user in front of Pepper touches its head, the robot proceed to ask the name and surname to the user, executing a speech recognition routine in order to understand if the current user was already met using the `check_new` class. As it is shown in Figure 10, the robot after the speech recognition phase is able to correctly classify the user as an old user finding the corresponding name in the database, also recovering the basket knowledge level previously recorded and stored in an old session.

```
Waiting a touch to start during 20.0 seconds...
A touch is detected.
Pose changed in joints: ['HeadYaw', 'HeadPitch']
Pose is back to normal.
Condition was resolved to a True value.
('Action: ', 'CHECK_NEW')
Say: Right decision! What is your name?
Speech recognition engine started
ASR value =  ['giorgia', 0.3206000030040741] 1659024689.13
ASR value =  ['', -3.0] 1659024689.27
dt 1659024689.649152 1659024689.132468  -  0.516680 30.000000
ASR: giorgia
Name is giorgia
Say: Perfect. Tell me also your family name please.
Speech recognition engine started
ASR value =  ['natalizia', 0.3898000121116638] 1659024696.25
ASR value =  ['', -3.0] 1659024696.54
dt 1659024696.919837 1659024696.245245  -  0.674587 30.000000
ASR: natalizia
Record giorgia natalizia exists.
Say: I am glad to see you again giorgia!
```

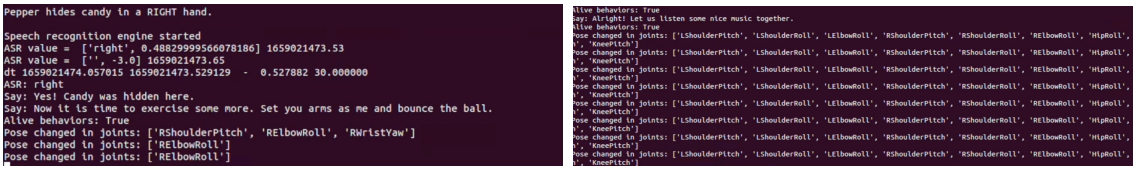
Figure 10: *Person touches Pepper's head, tells his name and the robot correctly categorizes him as an old user.*

After the user identification, the interaction proceeds with another step of speech recognition, both in case of new or old user. The reasoning behind this and the next phase of quizzes, as well as the global planning, are guided by plans generated by means of Contingent-FF: its execution is explained more in depth in the following section. Numerous tests were carried out to verify the correct classification of the user's knowledge in beginner, intermediate and expert and the planning was performed properly. It is possible to appreciate an example of classification in the following Figure 11 where the user is classified rightly as beginner after the questions.

Once the classification is completed, the interaction takes place via tablet. Depending on the user's level, it is possible to attend some lessons and to answer different quizzes in order to allow a personalized learning experience about basketball for the interacting person. In Figure 7b you can see how to access the lessons through the tablet: in particular, for this specific case the displayed lessons are for an expert user. The tests continued verifying all possible break classes implemented such as `Music()`, `Bounce()` and `Game()`. In Figure 12a there is the prompt overview of the execution of the *candy game* followed by Pepper showing how to bounce the ball. Figure 12b shows the correct change of pose of the robot simulating a dance. After the pause session the user can choose whether to continue learning, go back and review the past lessons or, if he wants, to quit the session. At the end of learning session when the user decide to quit, Pepper greets the user as Figure 13 shows and terminates the interaction. All the parameters chosen in the experiment session can be observed in Table 6.

```
Say: Do you usually play basketball during the week?
Speech recognition engine started
ASR value =  ['no', 0.6348999738693237] 1659021182.61
ASR value =  ['', -3.0] 1659021182.63
dt 1659021183.027117 1659021182.613701  -  0.413411 30.000000
ASR: no
('Action: ', 'ASK_QUESTION')
('Question: ', 'Do you usually watch basketball matches?')
('Anticipated correct answer: ', 'yes')
Say: Do you usually watch basketball matches?
Speech recognition engine started
ASR value =  ['no', 0.6425999999046326] 1659021188.03
ASR value =  ['', -3.0] 1659021188.16
dt 1659021188.555647 1659021188.028036  -  0.527608 30.000000
ASR: no
Entered switch plan section.
Found the next plan.
plans/easy_problem.txt
Say: Your level has been defined as a beginner. Welcome on board!
```

Figure 11: *Example of user classification as beginner*



(a) *Game and Bounce* **(b)** *Music*

Figure 12: *Execution of the possible break implemented*

```
Alive behaviors: False
end
Quit Pepper robot.
Alive behaviors: True
Alive behaviors: True
Alive behaviors: True
Alive behaviors: True
The database with this name already exists.
Alive behaviors: True
Alive behaviors: True
Say: I will be glad to see you next time! Goodbye!
```

Figure 13: *End of lessons*

Parameter	Description	Value
`setAlive`	`pepper_cmd` functionality intended to replicate alive behaviour	True
`threshold`	Both front and rear sonars' threshold for people detection	1.0 m
`wait_time`	Temporal filtering time for the person detection	3 secs
`monitoring_time`	Time allocated to a user for taking a decision to start interaction	20 secs
`tts_speed`	Pepper's `tts` server parameter `"speed"` when default is used	80
`timeout`	Timeout of the listener	30 secs

Table 6: *Experiments parameters.*

ii. Reasoning Agents

The execution of Contingent-FF (taking as inputs all the domains and problems that we defined above) has generated five the following solution plans:

- one plan for the global problem,
- one plan for user classification,
- three plans for quizzes, one for each level of difficulty (easy, medium and hard).

The diagrams, corresponding to the output plans, are reported below to provide a visual comprehension of the implemented reasoning flow.

ii.1 Global planner

In Figure 14 the solution tree related to the first global problem is shown. The most interesting part of this simple tree is represented by the two branches in a correspondence to the observation actions, `ask_start` and `check_new`. The remaining part of the tree is straightforward and composed by the standard step-wise sequence of actions.

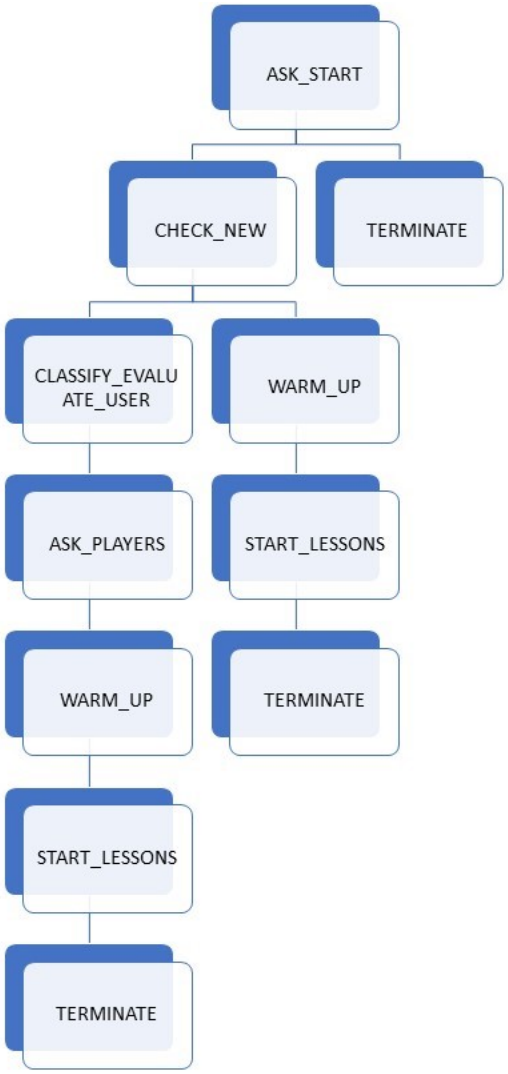


Figure 14: *Global problem search tree.*

ii.2 Classification-Evaluation planner

As it was described in the previous sections, when `classify_evaluate_user` is executed, the `classify_evaluate_executor` is called to parse the plans, generated in the classification domain. The structures of these plans are depicted in Figure 15 for user classification and in Figure 16 for user evaluation, respectively. The latter evaluation is done through asking quizzes which correspond to the level of experience of the human, which is instead defined by the former classification. As it can be noticed from the more articulated architecture of the binary trees, this hierarchical kind of approach represents the most complex part of our work made for reasoning. There are multiple traversable paths from the root to leaves, because each `ask_question/quiz` action is responsible of generating new branches.

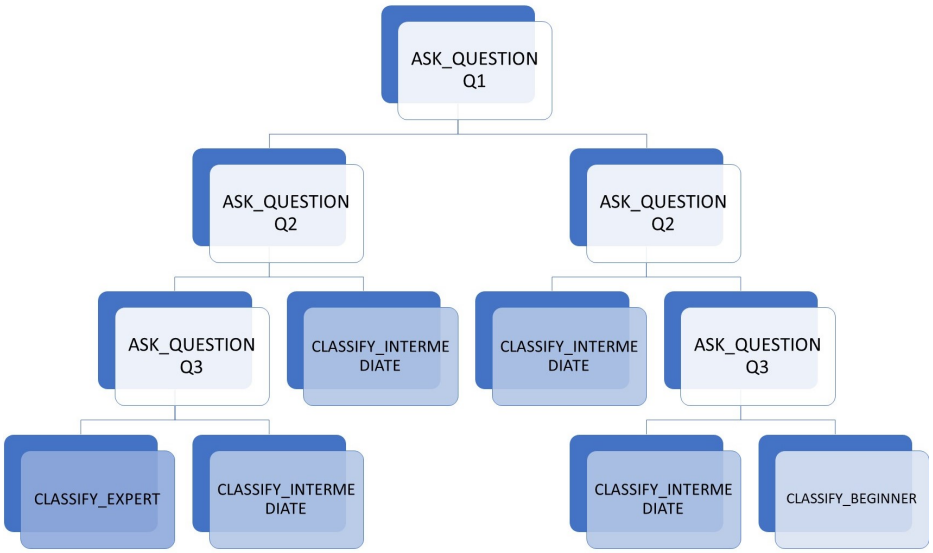


Figure 15: *Classification problem search tree.*

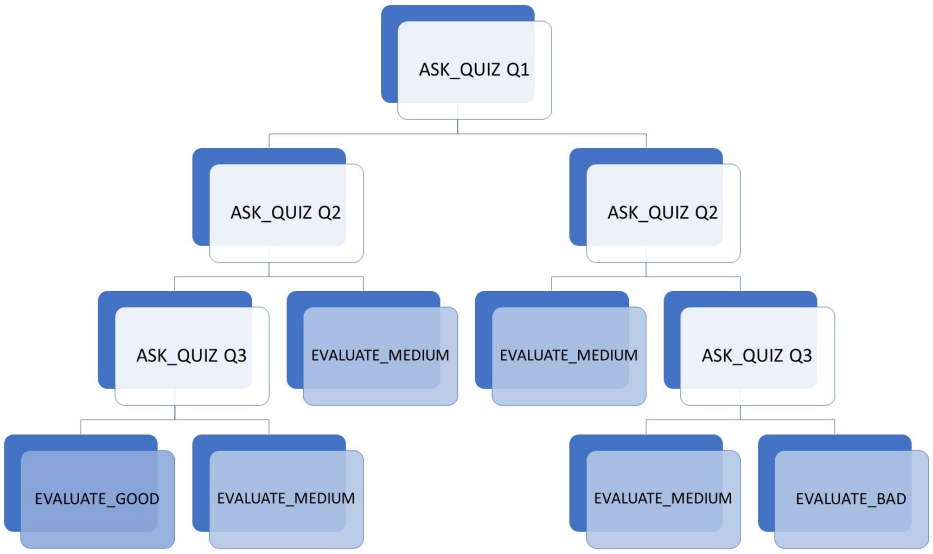


Figure 16: *Quiz problem search tree.*

The `ask` actions have a parameter, indicating the question to propose to the user. The three questions in the first part of user's interview (classification) are standard, while the other three (in the quiz session) we have arranged we have arranged in a difficulty-increasing fashion. The three questions of the easy quizzes, i.e. the ones for the users classified as beginners, are related to the basketball fundamentals, the other three − of the hard quizzes, i.e. for the users classified as experts − are concerned about the more complex aspects of the sport, while the three of the medium quiz, i.e. intended for the intermediate users, are a mix of the previous two sets of quizzes.

When traversing the tree at runtime, the `classify_evaluate_executor` properly schedules the next action and outputs the user's class (`classify_beginner/intermediate/expert`) in the end of the classification part and the user's skill level (`evaluate_bad/medium/good`) in the end of the quizzes, respectively. This is performed synchronously with the HRI part of the project thanks to the ability of the robot to integrate the automatic reasoning and understanding of the correct flow of the interaction with the services for the asking and listening to the user's answers.

VI. Conclusions

Teaching children can be really tricky, especially when trying to keep their attention at a high level. Using robots has been an open and very broad field of research for several years. In particular, a robot can arouse a lot of curiosity in children, making them focus on lessons more easily. We believe that by continuing to work in this area it could become possible to match human teachers with robots and make the learning experience even more enjoyable. Our approach's main goal is to show that basketball theory can be taught easily, indeed we included different learning levels starting from the beginner up to children who already have a good level of knowledge about basketball, personalizing their learning experience through the use of a Contingent Planning approach. We also developed a hierarchical architecture to manage the flow of information between the various modules making the execution smooth and also provide the planner with the correct information. The excellent results obtained, with perfect interactions and adaptation to the user, demonstrate how valid this type of teaching can be and that it can become part of the life of each of us.

In future works the goal would be expanding all the possible Human-Pepper interactions, maybe adding some more break categories, making the robot more entertaining in order to attract even more children attention. It could also be nice to integrate a further module to allow Pepper to interact with another human teacher to make the experience evolve and make it more complete. Finally, including more lessons to improve the learning level would be a great addiction.

References

[1] International Federation of Robotics. Ifr presents world robotics 2021 reports, 2021.

[2] Neil Savage. Robots rise to meet the challenge of caring for old people, 2022.

[3] Zhen-Jia You, Chi-Yuh Shen, Chih-Wei Chang, Baw-Jhiune Liu, and Gwo-Dong Chen. A robot as a teaching assistant in an english class. In *Sixth IEEE International Conference on Advanced Learning Technologies (ICALT'06)*, pages 87–91, 2006.

[4] Mamoun Gharbi, Raphaël Lallement, and Rachid Alami. Combining symbolic and geometric planning to synthesize human-aware plans: toward more efficient combined search. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 6360–6365, 2015.

[5] Lavindra de Silva, Mamoun Gharbi, Amit Kumar Pandey, and Rachid Alami. A new approach to combined symbolic-geometric backtracking in the context of human-robot interaction. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3757–3763, 2014.

[6] Emrah Akin Sisbot, Luis F. Marin-Urias, Rachid Alami, and Thierry Simeon. A human aware mobile robot motion planner. *IEEE Transactions on Robotics*, 23(5):874–883, 2007.

[7] Jim Mainprice, Mamoun Gharbi, Thierry Siméon, and Rachid Alami. Sharing effort in planning human-robot handover tasks. In *2012 IEEE RO-MAN: The 21st IEEE International Symposium on Robot and Human Interactive Communication*, pages 764–770, 2012.

[8] Chih-Wei Chang, Jih-Hsien Lee, Po-Yao Chao, Chin-Yeh Wang, and Gwo-Dong Chen. Exploring the possibility of using humanoid robots as instructional tools for teaching a second language in primary school. *Journal of Educational Technology & Society*, 13(2):13–24, 2010.

[9] Aditi Ramachandran Brian Scassellati Fumihide Tanaka Tony Belpaeme, James Kennedy. Social robots for education: A review. *Science Robotics*, 2018.

[10] Brian Scassellati Alexandru Litoiu. Robotic coaching of complex physical skills. *ACM/IEEE International Conference on Human-Robot Interaction*, pages 211–212, 2015.

[11] Guanglong Du, Mingxuan Chen, Caibing Liu, Bo Zhang, and Ping Zhang. Online robot teaching with natural human–robot interaction. *IEEE Transactions on Industrial Electronics*, 65(12):9571–9581, 2018.

[12] Fabio Patrizi, Nir Lipovetzky, and Hector Geffner. Fair ltl synthesis for non-deterministic systems using strong cyclic planners. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, IJCAI '13, page 2343–2349. AAAI Press, 2013.

[13] Ronen Brafman and Jörg Hoffmann. Conformant planning via heuristic forward search: A new approach. In Sven Koenig, Shlomo Zilberstein, and Jana Koehler, editors, *Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS-04)*, pages 355–364, Whistler, Canada, 2004.

[14] Jörg Hoffmann and Ronen Brafman. Conformant planning via heuristic forward search: A new approach. *Artificial Intelligence*, 170(6–7):507–541, 2006.

[15] Jörg Hoffmann and Ronen Brafman. Contingent planning via heuristic forward search with implicit belief states. In Susanne Biundo, Karen Myers, and Kanna Rajan, editors, *Proceedings of the 15th International Conference on Automated Planning and Scheduling (ICAPS-05)*, pages 71–80, Monterey, CA, USA, 2005.