A Dialogue System for Human-Robot Interaction

Christoforos Prasatzakis

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science degree in Computer Science and Engineering

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, 700 13 Heraklion, Crete, Greece

Thesis Advisor: Prof. Dimitris Plexousakis

This work has been performed at the University of Crete, School of Sciences and Engineering, Computer Science Department.

The work has been supported by the Foundation for Research and Technology - Hellas (FORTH), Institute of Computer Science (ICS)

UNIVERSITY OF CRETE
COMPUTER SCIENCE DEPARTMENT

A Dialogue System for Human-Robot Interaction

Thesis submitted by
Christoforos Prasatzakis
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: _____
Student

COMMITTEE APPROVALS

_____
DIMITRIS PLEXOUSAKIS
Professor, Thesis Supervisor

_____
THEODORE PATKOS
Researcher B, ICS-FORTH, Committee member

_____
ANTONIS ARGYROS
Professor, Commitee Member

DEPARTMENTAL APPROVAL

_____
POLYVIOS PRATIKAKIS
Associate Professor, Director of Graduate Studies

Heraklion, July 2023

A Dialogue System for Human-Robot Interaction

Abstract

Human-machine interaction and communication is one of the “hottest” topics in Computer Science. As technological advancements are getting closer to the non-expert human, the need for implementing free-form verbal communication between a user and an intelligent system is becoming more important, due to its simplicity and naturalness of interaction. One method of human-computer communication appropriate for non-expert users is “robotic conversation agents”, the job of which is to engage in conversation with the human: the user asks questions in a natural language (e.g., English), and the chatbot answers the user’s question in the same natural language as the question itself.

In this thesis, we designed a chatbot architecture (which we will call P-Chat), which uses the Event Calculus, a formal action language, in order to represent the chatbot’s knowledge. It relies on the Clingo reasoner, which implements ASP (Answer Set Programming) rules, in order to infer new knowledge about the robot’s world and about the agents inhabiting it, exploiting causal, temporal and epistemic reasoning.

Our architecture is adaptable to diverse domains that require chatbots for human interaction with machines. It enables users to ask questions not only about the chatbot’s environment, but also about other agents’ beliefs (epistemic questions) or about retrospective events, thus enabling users to form a more complete picture about the world and the events that occur and modify it. Finally, when it comes to training our system, we present a rigorous training method, so that a question can be reused with different agents/objects/domains, reducing the cost of retraining from scratch for each individual query.

We present the implementation of a dialogue system for the interactions between two agents – a human and a robot – which interact with other objects in the environment. All participating agents have partial observability of the world, therefore may possess incomplete or erroneous beliefs about the current world state or about the events that took place. The implemented system supports a wide variety of query types, ranging from polar to wh- questions, being either non-epistemic (i.e., about the state of the world), 1^{st} order epistemic (i.e., about what the robot believes the state of the world may be), or 2^{nd} order epistemic (i.e., about what the robot believes that other agents believe the state of the world is).

Keywords: Event Calculus, Answer Set Programming, Chatbots, Clingo, Knowledge Reasoning, Epistemic Questions, Human-Machine Interaction.

Ένα Διαλογικό Σύστημα Ανθρώπου-Ρομπότ

Περίληψη

Η επικοινωνία μεταξύ ανθρώπου και μηχανής είναι ένα από τα πιο “καυτά” θέματα στην Επιστήμη των Υπολογιστών. Όσο η τεχνολογία εξελίσσεται ολοένα και πιο πολύ προς την πλευρά του απλού ανθρώπου, η ανάγκη για τη διενέργεια ελεύθερου διάλογου μεταξύ ανθρώπου και μηχανής γίνεται ολοένα και πιο σημαντική, λόγω της απλότητας του και της “φυσικότητας” της αλληλεπίδρασης. Μία μέθοδος που εφαρμόζεται για επικοινωνία ανθρώπου-μηχανής με γνώμονα τον απλό και μη-ειδικευμένο χρήστη είναι τα “συστήματα ρομποτικού διαλόγου”, ή αλλιώς chatbots, των οποίων δουλειά είναι να συνδιαλέγονται με τον άνθρωπο σε φυσική γλώσσα (π. χ. Αγγλικά) δίνοντας κατάλληλες απαντήσεις στην ίδια φυσική γλώσσα με τις ερωτήσεις.

Σε αυτήν την εργασία σχεδιάσαμε μία αρχιτεκτονική για chatbot (την οποία ονομάζουμε P-Chat) η οποία χρησιμοποιεί Event Calculus – μία τυπική γλώσσα περιγραφής γεγονότων - για να αναπαραστήσει τη γνώση που κατέχει το chatbot. Η αρχιτεκτονική μας βασίζεται στο εργαλείο συλλογιστικής Clingo, το οποίο υλοποιεί κανόνες ASP (Answer Set Programming) για να συμπεράνει νέες γνώσεις για τον κόσμο και για τους πράκτορες που τον κατοικούν, χρησιμοποιώντας συλλογιστική αιτιότητας, χρόνου και γνώσης.

Η αρχιτεκτονική προσαρμόζεται σε διάφορα πεδία που απαιτούν τη χρήση chatbot για απάντηση ερωτήσεων από ανθρώπους. Επιτρέπει στο χρήστη να πραγματοποιήσει ερωτήσεις όχι μόνο πάνω στη γνώση του “κόσμου” του chatbot, αλλά και πάνω στη γνώση κάθε εμπλεκόμενου πράκτορα ξεχωριστά, είτε για αυτά που πιστεύει είτε για γεγονότα σε προηγούμενες χρονικές στιγμές (επιστημικές ερωτήσεις), επιτρέποντας έτσι στους χρήστες να αποκτούν μια πιο ολοκληρωμένη εικόνα του κόσμου και των γεγονότων που λαμβάνουν χώρα σε αυτόν και τον επηρεάζουν. Όσον αφορά στην εκπαίδευση του συστήματος, παρουσιάζουμε μια δυναμική μέθοδο “εκπαίδευσης”, τέτοια ώστε μια ερώτηση να μπορεί να επαναχρησιμοποιηθεί με διαφορετικούς πράκτορες/αντικείμενα/τομείς, μειώνοντας έτσι το κόστος για να εκπαιδευτεί ξανά μια ερώτηση από την αρχή.

Παρουσιάζουμε την υλοποίηση ενός συστήματος διαλόγου για την αλληλεπίδραση δύο πρακτόρων – ενός ανθρώπου και ενός ρομπότ - οι οποίοι επίσης αλληλεπιδρούν με διάφορα αντικείμενα στο περιβάλλον αυτό. Οι πράκτορες έχουν μερική αντίληψη του κόσμου, δηλαδή μπορούν να έχουν ατελή ή λανθασμένη αντίληψη για τον κόσμο ή τα γεγονότα που λαμβάνουν χώρα σε αυτόν. Το υλοποιημένο σύστημα υποστηρίζει μια μεγάλη γκάμα από τύπους ερωτήσεων, από απλές ερωτήσεις ναι/όχι ή ερωτήσεις του τύπου “που” και “πότε”, είτε μη-επιστημικές (για την κατάσταση του κόσμου), επιστημικές πρώτου βαθμού (τι πιστεύει το ρομπότ για την κατάσταση του κόσμου), ή δευτέρου βαθμού (τι πιστεύει το ρομπότ για το τι πιστεύουν άλλοι πράκτορες για την κατάσταση του κόσμου).

Λέξεις κλειδιά: Λογισμός Συμβάντων, Answer Set Programming, Chatbots, Clingo, Συμπερασμός Γνώσεων, Επιστημική Λογική, Επικοινωνία Ανθρώπου-Μηχανής.

Thanks

I would like to thank the following people for helping me fulfill this thesis:

- Prof. Dimitris Plexousakis for giving me the knowledge to understand the main theories involved with the subject of this thesis, and for offering me his support throughout the process of writing it.
- Researcher B of ICS-FORTH Theodore Patkos for the same reasons.
- All the people in the Computer Science Department of the University of Crete and ICS-FORTH for their support.
- And finally, all my friends and family, who advised me to move on and never give up in the face of failures and other setbacks.

INDEX OF CHAPTERS (pages)

Chapter 1. Introduction

Human and computer interaction is a rather “hot” topic in modern computer science research. To be precise, researchers always look for new and revolutionary methods of aiding the communication between humans and machines, every time proposing new methods and practices that make this feasible.

One of the many applications of human-computer communication technologies are chatbots - computer programs that engage in dialogue with human operators, who can ask them questions and receive appropriate answers. Especially in the area of human-robot interaction, where various entities observe, interact and potentially alter the environment they inhabit in, chatbots intend to offer a natural means of knowledge exchange. This is proven particularly useful when the humans that communicate with the artificial agents are non-experts and the environment is open, limiting the ability for the involved entities to maintain a clear and complete picture of what is happening or how current observations relate to past events.

A closely related area of chatbot research is that of *visual chatbots*. These chatbots present an image or video to the user, who can then ask questions on what they see and receive appropriate answers. These chatbots perform reasoning operations on the images they “see”, so that they have a “computerized” picture of what the picture shows. Such chatbots can be deployed in a vast variety of areas, such as navigation and medical applications.

Thus, we find ourselves having the need to take the already-existing chatbot context and expand it further, by adding new functionalities and concepts to it. We want to create a (visual) chatbot that can adhere to all of the above principles, while having a simple and intuitive way of describing the knowledge it receives from the outer world. And, while the above may seem rather menial tasks for chatbot research, we want even more. We do not just want a chatbot that can “read” its world and extract knowledge out of it. We want to answer the following questions: if the chatbot’s world has more than one agent, how do they interact with each other? What does each agent believe about the world? And also, how does past interactions and/or beliefs affect what the agents perceive and communicate now?

To answer the above questions - and many more - we introduce a system that can both handle this knowledge, as well as an expanding spectrum of user questions on it, while at the same time offering support for inferring new knowledge from an already-existing knowledge collection. To achieve this, the system integrates two things: a powerful chatbot along with a strong reasoner. In this thesis, we shall present the benefits that such an integration can have. Unlike other chatbot systems, that answer direct and simple questions, our system is able to go much further: it is able to answer complex questions based on knowledge it infers from what it already “knows” using the reasoner (not just what it perceives with its camera); more importantly, it can handle retrospective questions of the type “what if you did this in the past”; finally, it can answer questions about *other agents’* individual beliefs - a functionality that existing systems do not pay

much attention to.

In this thesis, we will introduce P-Chat, an experimental ICS-FORTH chatbot system, a sample application of which uses a camera to observe an area and “reason” on what it sees, so that it may answer users’ questions on what they observe or believe. To be precise, in the application scenario covered here, it monitors a table with an assortment of objects, upon which a human and a robotic agent interact. P-Chat reasons on the state of the world, as well as what the two agents notice and believe. Out project contributes to the following:

- We introduce an innovative way of reasoning on visual data using the Event Calculus, an easy-to-understand formal language for describing causal events in dynamic domains.
- We present an innovative chatbot architecture, which can be used with virtually any visual chatbot application.
- We also present an in-depth application of Event Calculus, showing how we can manipulate it better in order to describe a world, real or otherwise.
- We also introduce a rather interesting-for-a-visual-chatbot functionality; the ability to ask questions on what other agents believe and observe and receive appropriate answers to them (epistemic questions).
- Finally, we present a rigorous training method, so that a question trained once can be applied to a wide range of objects and entities, without the need to train it again for new objects/entities.

Certain queries supported may look simple and trivial at first, but our proposed system, in fact, is rather powerful and has lots of room for innovation and improvement in more than one areas. Our work opens the door for a rather intuitive chatbot framework, which is easy to implement and maintain thanks to its modular architecture and open-source nature, as well as improve and debug. In this thesis, we will first present an overview of the entire system and present related-to-it works, before we delve into its architectural design and describe how each of its components work. Then, we will present an assortment of use cases, in which our architecture will be evaluated on a vast array of applications and scenarios. Finally, we shall discuss implementation details for our application, such as what every component is made of and how they are set up to interact with each other, providing details on the exact nature of their inner workings and processes.

Chapter 2. Background

Section 2.1. Chatbot and Visual Chatbot Basics

Let's begin by explaining some chatbot basics first. Chatbots, also called *Dialogue Systems* or *Conversational Agents* are an integral part in human-computer interaction. The user engages in conversation with them in a Natural Language (NL) and the chatbot will have to answer in a Natural Language as well. Chatbot theory dates back to the 1950s, when Alan Turing tested whenever a machine can behave or converse like an actual human[LN6]. Chatbots can be utilized in many professional areas, such as medicine and shopping assistance, and come in many types, depending on the desired application. One such type of chatbot is the "visual chatbot".

The history of visual chatbots (of which the implementation we will present in this thesis is one) is rather new, and essentially, it's an expansion of the already known chatbot principles. Since the beginning, chatbot developers always wanted to add vision abilities to existing chatbots, so that they may be able to extract data from the real world as they see it. Despite the research however, it was until the 2010s when these technologies were starting to be adopted to then-current chatbot research and development.

One step in such research is the Visual Dialog[1] framework, developed in 2016. Visual Dialog's concept is pretty simple: given an image, a dialog history, and a question about the image, the chatbot has to ground the question in image, infer context from history, and answer the question accurately. Visual Dialog is essentially a standard for chatbot development, presenting a good framework that can be used to develop any kind of chatbot.

Section 2.2. An Introduction to Event Calculus

Now, how can a chatbot extract the information it needs from the image? This can be done through a variety of Computer Vision algorithms and methods. But even then, how can it describe it? A good way for a chatbot to programmatically describe events and situations as they happen is the Event Calculus[2]. The Event Calculus is a formal approach for reasoning about events and time within a logic programming framework. The notion of event is taken to be more primitive than that of time and both are represented explicitly by means of statements expressed in some logic-based language. Apart from events, Event Calculus (which we will also refer to as EC) can also describe *fluents*. A fluent is not something which happens *on the spot* at a given timepoint, but rather something that *holds* for a given period of time. An example of a fluent is the state of a lightbulb during a given time period. Lightbulbs can be either on or off. Therefore, if the lightbulb is on between two timepoints, we can say that the fluent that dictates that the lightbulb is on *holds* between them. If it's off, the fluent *doesn't hold*. This flexibility of EC allows us to use it to describe the knowledge the chatbot extracts from visual data in an accurate and intuitive way. In many recent implementations, the Event Calculus is written in the

Answer Set Programming (ASP) language and is processed by an ASP reasoner, such as Clingo (see next subsection).

It must also be noted that, in this project, the Event Calculus dialect that we are using is the Discrete Event Calculus[3], where we have a sort E of events (things that happen spontaneously), a sort of fluents F (properties that exist for a given time frame) and a sort of time points T (points in the time line, used to implement a linear time line). It also makes use of the *inertia principle,* which states that a fluent “holds” over time until an action causes it to hold no more. DEC also specifies domain-independent axioms that define events, fluents and enforce inertia in the reasoning process, while the specific domain is defined by a *domain axiomization*, which includes domain-specific axioms, world properties and initial state observations.

Section 2.3. ASP, Clingo and Reasoning Basics

ASP stands for Answer Set Programming[4]. It is a form of declarative programming oriented towards difficult search problems. As an outgrowth of research on the use of non-monotonic reasoning in knowledge representation, it is particularly useful in knowledge-intensive applications. ASP programs consist of rules that look like Prolog rules, but the computational mechanisms used in ASP are different: they are based on the ideas
that have led to the creation of fast satisfiability solvers for propositional logic. When basic EC rules are expressed in ASP, they are capable of creating a wide range of EC statements and predicates, capable of describing an entire domain in a powerful yet easy-to-understand manner. ASP, and by extent, EC rules and predicates can be processed by a solver program that, given an initial state, a set of predicates and a set of rules, can reason about what can happen in future time points and return appropriate predicates for those time points. Clingo is such a reasoner for ASP and EC.

Clingo[5], which is a part of the Potassco (Potsdam Answer Set Solving Collection) answer set solving tools, is such a solver app. Clingo essentially merges grounding with solving; by grounding we mean the ability to remove variables from an ASP program, since current answer-set solvers work on grounded(variable-free) programs. After the grounding process, the solver can assume the rules, the initial state, and generate models that predict future changes in our world. This functionality results in programs that can easily change their logic over time, as well as an advanced control interface for the reasoner, that can be accessed by a huge number of scripting languages, such as Python. Thus, we can use the reasoner to its maximum capabilities, providing powerful reasoning capabilities to any chatbot that may require them.

Section 2.4. Epistemic vs Non-epistemic statements and reasoning

What is also noteworthy is the ability to ask agents about what *other agents* notice or believe. In most traditional chatbots, the users always directly ask about what the state of the world is. This kind of question is called *non-epistemic questions*. In a non-epistemic question, we ask a system directly about the world state; in contrast, when we ask an agent about what it believes the world state is or what it believes other agents believe etc., , then we issue an *epistemic* question. Epistemic questions are just like non-epistemic questions, with the only difference that we now ask an agent about what *it* believes or notices. The usage of epistemic questions in a chatbot is a rather new concept, since most chatbots do not tackle the area of answering user questions about agents, but mostly, about the world they observe.

Unlike non-epistemic questions, epistemic questions can be leveled. By leveling means whenever we are asking an agent (level 1 observer) about another agent (level 2 observer). In this work, we will cover epistemic questions up to level 2. That is, questions of the type “Did agent A notice…”, which are level 1, and “Did agent A notice that agent B noticed…”, or similarly “Does agent A believe that agent B does not believe that..”, which are level 2.

We have already described what we need in order to perform reasoning for non-epistemic axioms. When it comes to epistemic axioms, however, we have to expand our already-existing non-epistemic axiomization, in order to account for epistemic reasoning as well. In order to do so, according to [3], we need to use the Discrete time Event Calculus Knowledge Theory (DECKT, Patkos and Plexousakis, 2009), which defines meta-axioms for the following: I) when an action occurs, if all preconditions of an effect axiom triggered by this action are known, the effect will also become known, ii) if at least one precondition is known not to hold, no belief change regarding the effect will occur; iii) in all other cases, i.e., when at least one precondition is unknown, but none is known not to hold, then the state of the effect will become unknown too. While the DECKT theory is sound and complete, the epistemic reasoning done is approximate; since the task is non-trivial for possible world-based implementations, the inferences are sound but potentially incomplete (since we only compute level 2 epistemic inferences). This is to alleviate computational complexity issues; we sacrifice completeness for less complexity.

Section 2.5. And it all boils down to…

Thus, we come to an application of the above principles, which is no other than our chatbot which uses EC to describe what it “sees” on a camera (in this sample implementation). And with the ability to answer epistemic questions, it has the ability to “read” what agents have observed or believe, allowing us to ask agent-specific questions, in addition to image-specific ones. This way, we provide a framework for a chatbot that can use an intuitive way of describing the knowledge it receives from the environment - presented in the form of events and fluents, while also allowing us to question individual agents on their own knowledge.

Chapter 3. Related Work

The works related to ours can be split into three categories: those that are related to the domain, the reasoner and the way the dialogue is handled.

Section 3.1. Works related to our domain

Domain-wise, the most relevant paper we are aware of is “Cut & Recombine”[6]. In this paper, the researchers propose a method for generalizing robotic manipulation actions in an ever-changing domain. They propose an ontology-based system that receives textual instructions on manipulating objects, then it generalizes them so that individual manipulation scenarios can apply in situations with different objects. The paper’s domain is very close to ours; we have a robot that has to learn information about objects in an ever-changing environment, which means that it has to notice vital details such as their positions and forms, as well as its own position perpendicular to the objects. Their approach is to define three data structures and two procedures. The data structures consist of an instruction ontology (with verbs and nouns for actions and objects for synonymy and instruction parsing issues), and ADT (action data table) database (a structure containing information from previous robot executions down to control level parameters) and an action template library (which holds abstract action encodings for a set of actions). The procedures used are symbolic processing (recognize action and objects in user question) and sub-symbolic processing (look for ADTs for actions similar to what the user asked). The purpose of the procedure is not only for the robot to perform an action, but also define a new ADT for future executions of the same action set. The aforementioned “Cut and Recombine” method is used to filter essential information from existing ADTs and then recombine it in order to produce the new ADT. The pro in this approach is that the symbolic processing produces mistakes, yet they are isolated. Also, it makes the instructions more clear for the robot to perform, resulting in better actions. The con, however, is that this procedure is more specific for a robotic arm; while it can be also used (in combination) with a chatbot, the general architecture is somewhat domain-specific, in particular, that of a robot with a camera and and arm. Still, domain-wise, this is a pretty good application of basic reasoning for a robot – and a chatbot of course, since it’s essentially a robot. But yet, it will have to be somewhat modified in respect to the data structures if it’s to be used in a chatbot application (use ADTs for question answering instead of performing general actions), since it mostly concerns a generic type of robotic agents.

A similar paper by Hatori and Kikuchi[7] deals with exactly the same problem; once again, we have a robot that has to manipulate objects in the environment, and it has to learn about them in order to do so, something which may not require reasoning, but it has a similar domain to ours. Here, the authors use a robotic arm and a camera as well, and the arm has to grab or move different objects on a tray. Apart from a dataset containing images of objects that the robot will

have to interact with (along with the objects' bounding boxes), Training is done with CNN neural networks, and, unlike the previous paper, where new knowledge is inferred, here we are using a dataset to train the robot in the new knowledge. And yes, while this may offer better training for robots that operate in domains with objects (in particular, robots that also interact with such objects), it has the con of not using reasoning, something that may conflict with chatbot (or other) tasks that use reasoners. Still, this paper is similar domain-wise to our work, and can provide a useful insight on how a robotic agent should see its world and how should it recognize entities within it.

Section 3.2. Works related to reasoning

In the reasoning field, CORPP[8] is the reasoning-related work that is closest to ours. The paper proposes a domain-independent approach to robotic reasoning in an ever-changing world using Answer Set Programming (ASP) and Partially Observable Markov Decision Processes (POMDPs). The way the paper represents ASP predicates is pretty much similar to our Event Calculus approach; in fact, with little-to-no modifications, the proposed system can be used to build a fully functional reasoner that can offer the same functionality and results just like ours. The theories proposed by "Cut & Recombine" can be also used to fine-tune Event Calculus instances in order for them to be reused in a more flexible manner.

The authors' approach here is to collect facts about the world, calculate possible worlds using an ASP-based logical reasoner that uses domain-specific axioms and the facts in order to generate new worlds for the agent to work into. It also makes use of P-log for probabilistic reasoning (random selection rules) and POMDPs for probabilistic planning (generate policies and models specific to actions, while also maximizing the reward values). The pros of this approach is that reasoning takes as little time as possible and it also offers high accuracy with as little cost as possible. It has also been tested in an actual robot (a shopping assistant), with the results being that the robot was able to fulfill a set of requests efficiently, and becoming more confident as its actions were approved. As for cons, the only thing we can note is that the procedure is somewhat complex, yet it's powerful enough to produce viable and reliable results for any kind of robotic task.

Compared to our reasoning approach, while their approach is a lot more different than ours – and it does not utilize Event Calculus at all, it's definitely a good insight on probabilistic robotic reasoning, and while our approach does not use probability, it can still be a worthy alternative to the Clingo Reasoner that we currently use; it may not use Event Calculus, but it can be further extended to utilize it. In other words, CORPP can be a valuable addition to our framework, if we ever have need to compare the already state-of-the-art Clingo and DEC(KT) with more probabilistic methods.

Section 3.3. Works related to dialog handling

When it comes to dialog handling, one of the closest papers is, again, [7], since it makes use of an external service (Chrome Web Speech API) in order to tokenize dialog and receive it in a parseable form. Our approach is somewhat similar both in premise and setup, only that we are using Facebook's wit.ai platform to perform the dialog handling job, whereas the Chrome Web Speech API used by this system is closed-source, whereas wit.ai is open-source, and the "robot" only observes how the world changes – it does not interact with it directly Also, apart from Machine Learning processes used to "train" the robotic arm, the system lacks reasoning abilities, therefore, being somewhat unable to answer certain questions that require reasoning (such as "what would happen if the arm put object A on top of object B?"), resorting solely on direct orders (e.g "Get me the tissue box"). While this may look like a good system for recognizing user questions and answering them based on already-trained information (of a certain kind), its lack of reasoning functionality may make it somewhat hard to use in ever-changing environments, where new situations happen and entities change.

Another paper that handles dialog in a similar manner is PathVQA[9]. PathVQA uses Stanford's CoreNLP toolkit in order to process dialogue and identify entities in the text it receives. Unlike [7] and our approach, however, PathVQA does not only match the parsed dialog to visual information, but it also uses it to form answers to any possible questions on it. This way, PathVQA can offer directed and more informative answers to users' questions, unlike other approaches, that offer generic-type answers with only different values. Just like [7]. however, PathVQA has the con of lack of reasoning abilities (which in the case of PathVQA's domain would enable a doctor to predict the progress of a disease), and therefore is not that suitable for asking questions that predict changes in the domain. This can be further reinforced by the fact that it offers answers it learns along with the questions on images in the training set. Thus, compared with the above two papers, we can see how our system achieves novelty by offering state-of-the-art reasoning abilities, while also utilizing a free and open source Natural Language processing platform in order to pre-process users' queries for proper question answering in both static and ever-changing environments.

Section 3.4. An extended look on Visual Dialog

In the area of dialog handling, it would be also noteworthy to analyze Visual Dialog as well, as it's one of the basic principles on which our architecture is based on. Visual Dialog is not essentially a chatbot, but rather, an Artificial Intelligence (AI) task where humans ask the bot about details on a picture and the bot answers the questions by identifying entities in the picture and evaluating them according to the question's context. This task is used to build a simple visual chatbot. The chatbot's role is pretty much obvious: it is the agent that accepts the user's questions, analyzes them and offers proper answers.

The task is trained using a dataset (VisDial) that consists of pictures and questions on the entities that appear in them. The system then uses a series of Neural Network answerer models in order to generate the answer that will be returned to the user. It uses three encoders to transform image input and answers into a vector space: Late Fusion (LF), Hierarchical Recurrent Encoder (HRE) and Memory Network (MN). After the image is encoded into vectors, the system will have to return an answer listing, sorted on whenever the answers answer the given question (in vectorized form too). The best answer is the one that will be returned to the user.

The difference between this system and ours is that there is no reasoner; all predicates are extracted from the images as vectors, and those vectors are then used to answer the question. It also has a wider range of applications, not just for conversion between a human and a robot; this means that it's rather context-free – any image can be used. This flexibility is what gives it its characteristic wide range of applications, which means that it can be used in our sample implementation's context as well.

Overall, Visual Dialog presents an essential task when it comes to visual question answering. Its flexibility allows adaptation to any possible context, including ours, with the robot and the objects. While it definitely offers infinite possibilities, it seems that it still has some room for improvement, especially when it comes to efficiency - which can be improved by tweaking certain parameters in the encoders. Nevertheless, improvement or not, it's a system worthy of any application and interesting to research thoroughly.

Section 3.5. PathVQA and our system

While not technically a chatbot itself, PathVQA is a dataset that can be used to answer questions on medical images, in particular, pathology images. The need for the dataset arises from the need to automate pathological diagnosis, as well as the need to develop new, cutting-edge technologies that would prove beneficial for the medical area.

The dataset can be used to create a chatbot that, given a pathology image of a patient, can accurately describe everything associated with it, identify possible diseases, and, in general, act as an "AI Pathologist". Much detail is used in the generation of the dataset, so that a possible chatbot that utilizes it could pass the American Board of Pathologists exams.

For the generation of dataset, the first task is to extract images from pathology media, along with their captions. This is done with tools such as PyPDF2 and PDFMiner. Using Stanford's CoreNLP toolkit, long sentences in the captions are simplified into shorter ones, and all their components (subjects, verbs, clauses, etc.) are then rearranged - using Tregex from the CoreNLP toolkit - into questions. Each question is then mapped to its respective answer, and both are mapped to their respective image. Many question types are used throughout the dataset, with the most popular being "Yes/No" ones, "whats", "wheres" and "hows". In a nutshell, the data is represented as such: every image is accompanied by questions on it, along with their answers.

We can see some similarities with our proposal here, despite the project being just a dataset. First, just like the sample implementation covered here, we need to answer questions on an image. And then, we can use “where” and “what” questions, as well as “Yes/No” questions, although in a completely different context. We also notice that the dataset is completely generated automatically, whereas in our project’s case, it has to be trained by supplying questions and identifying entities in them manually (although the finished dataset can be extracted and imported later as-is). Also, given that still images are used, there is no mention of reasoning of any kind, implying that this is a task that should be handled by the chatbot that will use the dataset (although a third-party reasoner can be considered, if needed).

To sum up, PathVQA may not be a chatbot, yet, theoretically, it sets the foundation for a basic visual chatbot task (perform pathological diagnoses based on patient images). What is also interesting, given the way the questions and answers are represented, is that if we represent the answers as Event Calculus instances, and then train the questions in wit.ai (in an automatic manner, of course), we can set the bases for a “virtual pathologist” that can answer questions on medical images (providing that we have a reasoner that can analyze new images as well). While this may seem a bit “haphazard”, it nevertheless presents a chance to test and use P-chat’s abilities in a more advanced and purposeful field.

Chapter 4. Architecture

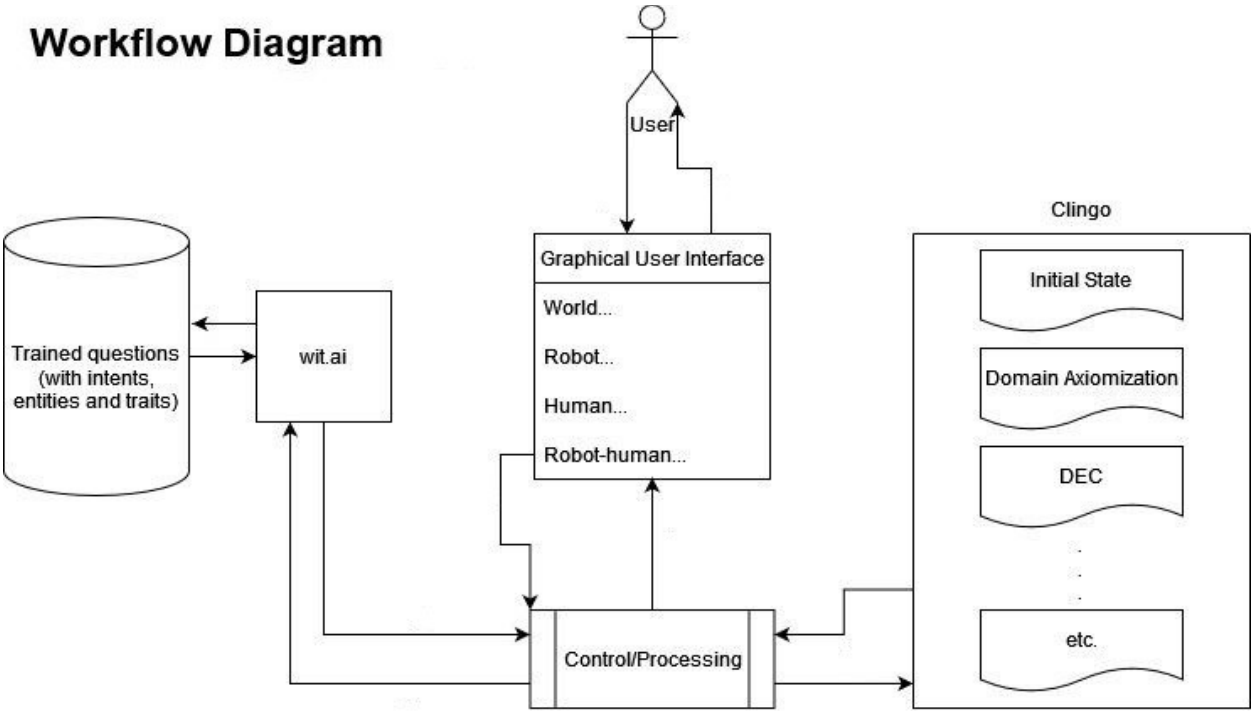
We begin this chapter with an analysis of each component comprising our system, whose architecture is shown in the diagram that follows (in section 4.1).

Section 4.1. Architecture, Workflow and Component Analysis

Before we describe the architecture workflow of the P-Chat system, we should first specify what types of users are involved in its operation. Running and using a chatbot system involves many people, each with their own role, assigned to individual tasks. In our project, we can distinguish users in three types. The first type is the *knowledge engineer (KE).* The KE is responsible for writing the Event Calculus theorems and functions that define the chatbot's domain. As we have specified earlier, the domain may be different from setup to setup, so the KE has to account for each and every object, agent or any other tangible entity associated with the domain (as well as the domain's environment) so that they may represent the domain in EC as accurately as possible. The second type of user is the *chatbot engineer*, whose job is to train the chatbot in the Natural Language utterances it should handle, as well as their respective question types. In order for the chatbot engineer to train the correct utterances and question types, they must know what EC theorems represent the given domain, so as to "match" these theorems to their respective Natural Language questions. Thus, the chatbot engineer's job requires that the KE should also do their job correctly, since they need their theorems in order to train the appropriate questions. And finally, we have the end user, who uses the trained chatbot once it has been set up and is online.

In the implementation described in this thesis, we assume that we have a robot that operates in a specific domain and interacts with other users in it. This robot can infer knowledge (which it represents using EC) using its own reasoning capabilities. Users can then ask questions to this robot about what it has inferred, whenever this knowledge is about the world it observes or the agents' individual knowledge.

The figure below shows the architecture of our project in a nutshell:



Section 4.1.1. Wit.ai: the main chatbot component

We first discuss the chatbot. What the chatbot does in our system is rather straightforward: it receives a user query written in a Natural Language (English, for now), recognizes possible entities in the query (objects, agents, time points, etc.), recognizes the query's intent (whenever it's a question on knowledge, a question of what would happen if something, a greeting, a question on what an agent observes, etc.) and assigns the query a trait (a boolean variable, often with the "true" value) that describes what exactly does the question concern (whenever it is about an object's position, whenever an agent moved, whenever an object is in a position relative to another object, etc).

Once the chatbot has recognized everything of the above, it prepares a JSON object which contains all of this info and returns it to the client that sent the query in the first place. This way, the client receives the query fully analyzed with everything represented in a clear format, ready to be processed by any code running in the client. Thus, the chatbot handles the job of pointing out the main points of a Natural Language question, so that a client code can manipulate them in order to "understand" what exactly the user asked.

Section 4.1.2. Clingo: our system's "mind"

Another crucial component in our architecture is the reasoner, in our system, *clingo*. Clingo handles the job of *reasoning* new knowledge from already existing ones. Which means that, it

takes the knowledge our system currently knows and expands it with new knowledge, which it generates by inferring on the already known knowledge. In our system, knowledge is stored in a set of files, which clingo processes. These are:

- A DEC (Discrete Event Calculus) file containing general-purpose Event Calculus predicates.
- DECKT files: these contain predicates and functions for epistemic queries (queries on other agents' knowledge).
- A file containing domain-specific EC predicates. This may be different from domain to domain.
- A file containing an initial state, which is a snapshot of the world that will be loaded when the chatbot starts. The system can modify it for "what would happen if" questions.

The reasoner is called when the system starts (in order to compute the initial state) or when questions of the "what if" kind are asked, in which case the initial state (mostly) is modified with new predicates (in a temporary file) and then the reasoner reloads it with the new predicates, in order to answer the user's question.

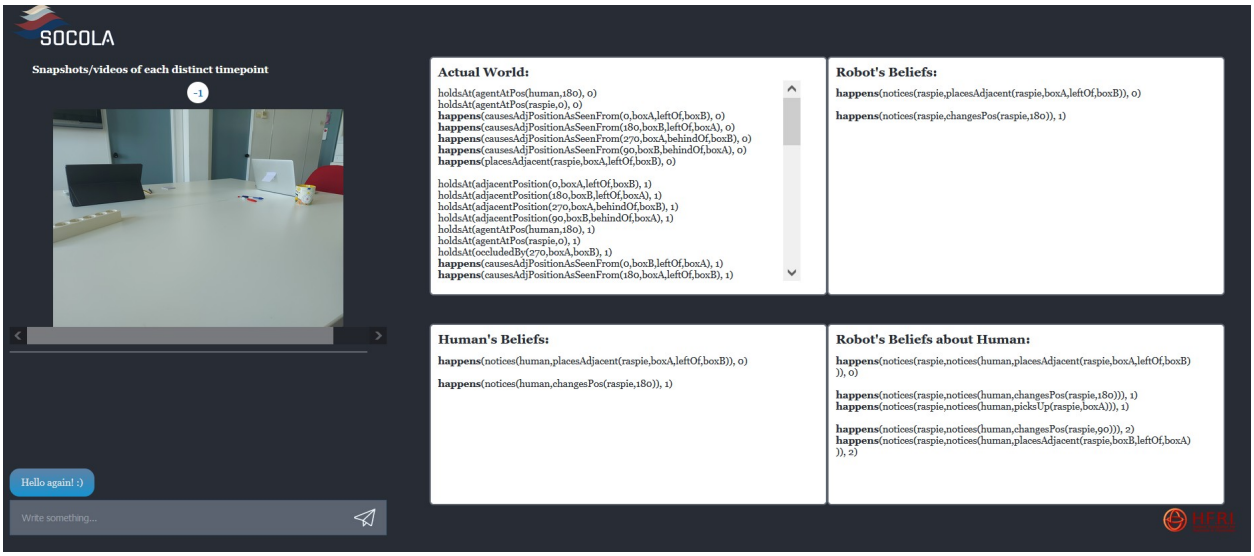
Section 4.1.2.1. The Application of Event Calculus in our pipeline

In order to perform reasoning, Clingo needs to consult a set of files containing domain-specific and domain-independent Event Calculus and ASP predicates. In this project, it needs the following files (i.e., rulesets): a DEC (Discrete Event Calculus) axioms, DECKT files for first and second order epistemic inferencing (described later on), a domain axiomatization, which contains the domain-specific information, and an initial state file. The DEC file contains domain-independent Event Calculus axioms, which can be applied to any domain that works with Event Calculus without the need for modifications. The DECKT files - one for level 1 and one for level 2 epistemic queries - contain rules defining epistemic predicates. The domain axiomatization specifies the dynamics and causal properties of domain that the robot operates in; it defines rules and axioms about everything in the robot's environment (agents, objects, etc.) and handles domain-specific fluent and event operations. P-Chat should be able to interpret all these knowledge, in order to respond to queries issued by the end user with respect to the domain. Finally, the initial state contains predicates for the state the robot will find the world when it comes online, and it acts as the "base" that the reasoner will use to generate knowledge up to the time point the user will specify.

Section 4.1.3. What the end user sees: the Graphical User Interface

Another vital component of our architecture is the Graphical User Interface. This is the main component the user uses to communicate with the chatbot. The users can input their queries on the GUI and receive possible answers on them. Apart from the users themselves, the GUI also communicates with the control/processing unit, sending queries to it and receiving results from it. Essentially, it is the "middleman" between the user and the control and processing unit, which is the "core" of the architecture. It receives the user's query, forwards it to the controller for

further processing and then takes the answer from the controller and shows it to the user. The GUI also holds the current state of the world and what agents believe about the world and other agents. This information is shared with the other components in the architecture when requested. Here's an example GUI session, with a sample initial state:



Let's take a closer look at what we have here. In the leftmost area, we see an image captured by the chatbot's camera, and it shows the world as the chatbot sees it: the objects, the agents and the environment in general. Right below it, we have a Facebook Messenger-style chat interface. This is where the users ask their questions and receive their answers.

In the right area, we have four windows. Each of this windows contains Event Calculus predicates that describe the current and past states of the world, as well as what the human and the robot (our primary agents) believe and observe. We have a window for the state of the world, two windows for what the human and the robot believe and observe, as well as a window about what the robot believes and observes about the human.

Let's see some predicates seen in the above picture. The holdsAt(agentAtPos(human,180),0) seen in the "Actual World" window states that in the actual world, agent "human" was at position 180 at time 0. In a similar vein, happens(notices(placesAdjacent(raspie,boxA,leftOf,boxB),0) in the "Robot's Beliefs" window states that the robot noticed that the raspie (the robot itself, raspie is an alias for the robot) placed boxA left of boxB at time 0. In this manner, we can easily organize predicates not just about the state of the world, but also predicates about what every individual agent believes or notices.

Section 4.1.4. The Controller – the heart of the system

The last component in our diagram is the controller, which is the "heart" of the entire

architecture, seeing as every component sends and receives information from it. The controller handles the job of sending the user's raw question to wit.ai, receiving the processed output, creating an Event Calculus predicate out of the question's intent and entities and then looking into the predicates representing the current world knowledge to find all predicates that match with what it generated from the user's question.

If the user's question concerns what would happen if something happened, it appends the EC predicate resulting from the user's question into a temporary copy of the initial state and then calls the reasoner in order to get knowledge on how the world would look if the action the user described happened. It then looks for the answer in this new knowledge.

In any case, if the controller has an answer for the user, it sends it to the GUI, which then displays it for the user. If there is no answer to the user's question, it just sends a negative response to the GUI.

To sum it up, the controller is the main action hub of our architecture. It has the greatest degree of responsibility in our system, seeing what information it processes and how it interacts with each other component in the architecture. And while this may look centralized a lot, it enables for a higher degree of independence for the other components, which means that modifying a component other than the controller will have little to no consequences in the operation of the system as a whole, something which also enables us to intervene other components in the pipeline as well, with minimum consequence (for example, we can add a "swear filter" between the controller and wit.ai that can filter out expletive words - something which will not have any consequence in the system's operation). We can see, this way, that this centralization opens the door for a great degree of modularity, which is crucial for ever-evolving systems like ours.

Section 4.2. Types of questions the system can handle

In this subsection, we describe the types of questions our architecture can handle. These questions are written in a controlled language, which specifies keywords and NL query architecture for different kinds of questions. Controlled language keywords include mostly domain-specific information, such as agents, objects, angles, etc., and given that they are domain-specific, they can be defined by the user. Currently, our architecture supports the following kinds of domain-independent questions (we will analyze them in detail in later sections):

- Polar questions: These non-epistemic questions are direct questions on events and fluents on the actual world. When asking a polar question (or a non-epistemic question in general), the user asks the system what happened in the actual world, not what the agents believe or observe. An example of a polar question is: "Was the door open at time 1?". Here, we are asking the system on what happened in the actual world at a certain time point, therefore, the answershould be in the "Actual world" knowledge

section, as seen in the GUI above. A polar question's answer is either a "Yes" or a "No", sometimes followed by additional information (such as the time points where the question's answer is a "Yes"). As we will see later, like all non-epistemic questions, polar questions can become epistemic as well.

- Epistemic questions: these questions concern what agents believe and observe - their individual knowledge. An epistemic question has two principal parts; the epistemic part and the non-epistemic part (which can be a polar question, a "where" question or any other kind of non-epistemic question). For example: "Does the doctor believe that the cancerous cell mutated at time 2?". Here, the epistemic part is "Does the doctor believe that" and the non-epistemic part is "the cancerous cell mutated at time 2". The non-epistemic part corresponds to an already trained non-epistemic question type - which means that epistemic questions are not a separate kind of questions, as seen by wit.ai, but rather an extension of non-epistemic questions. In the epistemic part, the agent specified is the one whose knowledge we explore, and is called the *observing agent*. If the non-epistemic part also concerns another agent's actions or beliefs, this agent on the non-epistemic part is called the *actuating agent*. It must be also noted that epistemic questions can be *nested*, that is, we can interrogate agents on what *other* agents have seen or believe. In the above example, a nested variant could be this: "Does the nurse (observing agent 1) believe that the doctor (observing agent 2) believes that the cancerous cell mutated at time 1?". In this case, the answer lies in the first observing agent's knowledge, which holds knowledge about the *second agent*'s knowledge.
- "When" questions: These non-epistemic questions ask *at what time* something happened (or a fluent held) in the actual world. The answer to this question type should always be a time point on which what the user asks for happened - if it did. Being a subcategory of non-epistemic questions, the same rules as with non-epistemic questions apply here.
- "Where" questions: This kind of question is non-epistemic as well, and it asks *where* an agent or object is (or was) at a certain time point. The answer here should be a location. This question type is rather domain-specific (for applications where locations are needed), thus, it does not apply to all use cases of our architecture, only in implementations where it's needed.
- WHATIF questions: These questions ask the system what would happen if a certain event happened (or a fluent held) in a past or future time. When processing these questions, the system forms an event/fluent predicate out of the user's question, appends it to a new initial state and then calls the reasoner, which computes models on that new initial state and returns an answer to the user's question from those models. An example of this question would be: "What if agent A had pushed the button at time 4?"

Section 4.3. Summing the architecture up…

When it comes to forming a general picture of the system, this architecture describes a modular system for a chatbot that answers users' questions on knowledge gathered by a robotic agent.

Our sample implementation here belongs to a special chatbot category called “visual chatbots”; chatbots that answer users’ questions on a picture or a video. The architecture described here is to a large extent context free and can be applied to any kind of chatbot implementation. For example, we can use it to implement a visual chatbot that answers doctors’ question on medical imagery, or we can use it to build a system that tasks users with a virtual “treasure hunt”. Implementation requires only trivial use of wit.ai and JSON objects, it can be “installed” easily and it can be modified easily as well - all components are independent of each other, which means that we can add new ones or change existing ones with minimal consequence.

As for the users, all they have to do is to enter their questions on the system’s GUI and get their answers on the spot, on the same GUI. The implementation covered here is targeted mostly at chatbot/AI/ML researchers that investigate how does a chatbot make its reasoning and language handling, but other implementations can be catered to an even larger audience, as described in the two implementation examples above.

It must be noted, however, that our system has certain limitations, that may make it “far from perfect”. Namely, it will answer user questions only if these questions belong to a question type that has already been trained; anything else will just return an error message. Also, the accuracy of the answer is dependent on the degree of training for the respective question type – if the question type is not trained well enough (for example, if it’s not trained with a sufficient number of example questions), wit.ai may “misjudge” the question type it receives and return an answer for a wrong EC predicate or set thereof; it may also fail and return an error message instead. Speaking of which, if users ask questions that are of types not yet trained in wit.ai, the chatbot makes suggestions for further training of such questions, but it’s not done automatically; the chatbot engineer should do this by hand – a somewhat tedious process. Error recovery is also an issue – if the system recognizes a question wrongly and it fails with an error message (unlike a case where it may return a wrong answer), the system will block and continue to provide the same wrong answer until restarted.

In spite, however, of the above limitations and “bugs”, we believe that our system has a high enough potential for improvement and further expansion, allowing us to address the above limitations the best way possible, as well as add further functionality that may help it become a “state-of-the-art” chatbot system.

Chapter 5. Methodology

Section 5.1. General methodology idea

Our system is a chatbot that observes the world and describes events and situations that happen on it using Event Calculus (an ASP-based language designed for describing events), along with what the agents (world actors) see and believe - their knowledge, described in Event Calculus as well. The user can then ask questions on what the chatbot "sees" in the world and the agents' knowledge. Apart from events, the system can also describe "fluents", assumptions that hold for a time period between two time points. Our system is not solely restricted to visual chatbots; it *can* be used to implement one, but it can be used for implementing any kind of chatbot in general.

Section 5.2. Training the system

Training the system, in order to recognize different categories of questions, different types of questions of a given category, or to accurately identify entities and intents within a given query, is an important aspect of this work. The training process of our system concerns two components: wit.ai and the controller. What makes the training procedure for this system ideal is the fact that no code needs to be modified. The only thing the user has to do is to train new utterances in wit.ai and then add these new utterances in the controller's parameter file. This way, the controller can automatically detect the new utterance's properties and work accordingly. Let's see how this training can be done.

Section 5.2.1. What kinds of utterances can we train?

Before we start describing the training process step-by-step, we need to specify what kinds of utterances our system can support, and how these utterances are specified. Our system complies to a *controlled language*, that is, a certain coded syntax that defines the kind of questions we can ask in our domain. Each question has a different type and meaning, and they have both non-epistemic and epistemic versions, with both versions having the same intent. The controller language also specifies the *entities* a question can have, that is, every object that can appear in our domain, such as time points and agents. In the following controlled language specification, we will use the "PFQ" code to refer to polar (P) question types that refer to fluents (F), i.e., world aspects that hold for a certain time range, and "PEQ" for questions that refer to events (E), i.e., changes in world aspects at given times. Now, let's present the controlled language for our implementation's domain:

Section 5.2.2. Entities

In a question, entities represent objects, agents, and in general, everything that can be

described in our world. These also include other non-visible data, such as time, an object's or agent's position in the world, the world's weather and other information. An entity can be easily realized as a variable that accepts a certain set of values. For example, an entity called "daytime", that can describe whenever it's day or night in the chatbot's world, can be defined as <daytime> = {day, night}, where <daytime> is the entity's name, and {day, night} are the keywords that the chatbot will associate with them. Below we have the entities for our implementation's domain:

<agent> = {agent, human, raspie, robot}
<time> = {0, 1, …}
<angle> = {0, 90, 180, 270}
<object> = {table, boxA, boxB, laptop, pen}
<spatialRel> = {onTopOf, leftOf, behindOf}

Section 5.2.3. Polar questions

A polar question is a non-epistemic direct question on the world's state. When users ask a non-epistemic question, they inquire about what they currently see *in the world*, not what an agent believes. Polar questions are the simplest non-epistemic questions that P-Chat can handle, and are likely the *first* questions a user may ask, so, it is important that we train as many polar question categories as possible. An example of a polar question could be: "Are the lights on?". In this question, we are asking directly about the *lights*, which is an entity in the chatbot's world. We can further generalize this question if we replace "lights" with any other valid entity keyword for this question. For example, we can say "Are the monitors on?" or "Are the fans on?". The answer to a polar question should either be a "Yes" or a "No". As we can see, our question model allows for question boilerplates, which can accept a wide range of entity values as arguments.

Below are some polar question types our implementation can handle, along with their possible answer values (since every question may return a different type of answer depending on its context). Remember that PEQ questions are questions about events (which happen at a certain time point and no other) and PFQ questions are questions about fluents (which ask about a state that is valid between two time points). All polar questions begin with the "P" prefix, specifying that the question is a polar:

PFQ1.

Is/was <agent/arg1>
located at <angle/arg2>
[at time/timepoint <time>]?

Answer:
Yes
Yes, at times T = 3, 4, …

PFQ2.

Is/was <object/arg1>
<spatialRel/relation> <object/arg2>
[as seen from <angle/arg3>]
[at time/timepoint <time>]?

Answer:
Yes
Yes, at times T = 3, 4, …

PFQ3. Combination of observations

Is/was <object/arg1>
occluded
[by <object/arg3>]
to the <agent/arg2>
[at time/timepoint <time>]?

Answer:
Yes
Yes, at times T = 3, 4, …

It is important to note here that the general pattern of a question is only given for reference. The chatbot can be trained to recognize a given PFQ type, even if a different phrasing or wording is used by the end user. For example, instead of “occluded” in PFQ3, the user may use the word “hidden”; the better the training is, the higher the confidence is that the question type has been correctly recognized. Section 4.X later on provides more details about how proper training can be performed.

PEQ1.
Did <agent/arg1> change its/her position [towards <angle/arg2>] [at time <time>]?

e.g
Did human change her position at time 12?
Did the agent change its position towards 90?
Did the agent change its position towards 90 at time 2?

Answer

Yes.
Yes, towards angle X1 at time T1, towards angle X2 at time T2, etc

PEQ11.
Did <agent>
put/place <obj1>
<spRel> <obj2>
[at time <time>]

e.g
Did the raspie place boxA leftOf boxB at time 15?
Did the human place the laptop behindOf the pen at time 3?
Did the agent place the pen onTopOf boxA?
Did the robot place boxA leftOf the pen?

Answer
Yes, at time(s) <time>

PEQ2.
Did <agent/arg1>
pick up <object/arg2>
[at time <time>]?

e.g
Did human pick up boxA?
Did robot pick up laptop at time 7?
Did raspie pick up boxB at time 11?

Answer:
Yes.
Yes, at time <time>.
No.

Section 5.2.4. When questions

"When" questions are non-epistemic questions that don't ask about just *anything* in the world, but about *when* something happens or holds - that is, they're asking about *a certain time point or intervals of time points.* These questions - whose codes always start with "WN" - in order to be clear in terms of format, always start with the keyword "when". An example of such a

question is “When did the window open?”. Here, we want to learn at what time point(s) did the action of opening the window take place, not how it opened or who opened it. Below we have sample categories of “when” questions handled by our implementation.

WNEQ1.
When did <agent/arg1>
change its/her position
[towards [angle] <angle/arg2>]?

Answer: a timepoint or a set of timepoints.

Examples:
When did the robot change its position towards 90?
When did the human change her position towards angle 180?
When did the human change her position?

WNFQ1.
When was <agent/arg1>
located at [angle] <angle/arg2>?

Answer: a timepoint or a set of timepoints.

Examples:
When was the human located at angle 90?
When was the robot located at 180?

Section 5.2.5. Where questions

These questions are also a type of non-epistemic questions that ask “where” was an agent or object during a certain time point or period, and as such, their codes always start with the “WR” prefix. The answer to these questions is a relevant agent/object position (or a set of them). These questions are rather domain-specific, as they concern domains that involve asking questions on agents’/objects’ positions. These questions, just like “when” ones, have the main characteristic that they always start with the keyword “where”. An example would be “Where is the pen?”. Sample “where” question types that our implementation supports follow:

WRFQ1.
Where <is/was>
the <agent/object>
located at time <time>?

NOTE: if “is” is used after “Where”, the “time <time>” part is omitted, since we are talking about the current timepoint.

Answer: the agent’s location at timepoint <time> (or the current timepoint)

Examples:
Where was the robot located at time 5?
Where was the human located at time 12?
Where is the robot located at? (current timepoint)
Where was boxA located at time 1?

Section 5.2.6. Epistemic Question types for polars

An epistemic question is a question that does not ask directly about something in the world, but about something a certain agent (observer) sees or believes (at a certain time point). They consist of two parts: an epistemic and a non-epistemic part. The epistemic part specifies *who* the observing agent is, while the non-epistemic part is a non-epistemic question, which is what we are asking the observer about.

Currently, our system supports two levels of epistemic questions: 1 and 2. 1st order epistemic questions are about asking *a single observer* about what it believes or sees. 2nd order epistemic questions are about asking an observer about *what another agent (actuator)* believes or sees. These are also called “nested” epistemic questions, since we are asking someone about someone else. It is possible in future revisions of this project to add support for greater levels of epistemic nesting, given the degree of flexibility our question model offers.

All the aforementioned question types can become epistemic, if the question starts with statements, as the ones below, maybe with some rephrasing. The answer remains the same. Below we present two intents for level 1 and level 2 epistemic, which can apply to all of the above PFQ and PEQ queries. Keep in mind that EpF refers to questions about fluents (PFQs) and EpE refers to questions about events (PEQs)

EpFLv1.

Does [the] <observing agent> believe that/According to [the] <observing agent>
PFQxxx

Some trained examples (non case sensitive):

- Does the agent believe that the human is located at angle 90? (PFQ1) ● Does the robot believe that the laptop was ontopof the table at time 0? (PFQ2) ● According to agent is the pen leftof the laptop as seen from angle 0 at time 0? (PFQ2)

- Does the robot believe that the pen was occluded by the table to the human at time 2? (PFQ3)
- According to the robot is boxa behindof boxb as seen from angle 0? (PFQ2) ● According to raspie, is the laptop occluded by boxb to the human? (PFQ3)

EpFLv2.

According to [the] <observing agent>, does [the] <acting agent> believe that
PFQxxx

EpELv1 and EpELv2

EpELv[1,2] are defined the same way as EpFLv[1,2], but since we are asking about events, the agents don't believe, but notice. That is, EpELv1 will be "Did [the] <agent> notice that <PEQ>" and EpELv2 will be "Did [the] <observing agent> notice that [the] <acting agent> <PEQ>".

Section 5.2.7. Epistemic when questions

"When" questions can also become epistemic. The rules for this are the same as with any other type of question. Below we have some examples of such epistemic questions from our domain:

WNEpEQ1.
According to the <observer/arg1>,
<WNEQ1>

Answer: a timepoint or a set of timepoints.

Examples:
According to the robot, when did the human change her position towards angle 90?
According to the human, when did the robot change its position towards 180?
According to the robot, when did the robot change its position?

WNEpFQ1.
According to the <observer/arg1>,
<WNFQ2>

Answer: a timepoint or a set of timepoints.

Examples:
According to the human, when was the robot located at angle 90?
According to the robot, when was the human located at 90?

Section 5.2.8. “What if” questions

“What if” questions (which always start with the WHATIF prefix) constitute an important type of question that most systems fall short in handling, as they require retrospective reasoning. These questions ask “what would happen if…” types of hypotheses, where the user wonders what would happen to the world if something happened or held (at a certain time point) in the world. An example of such a question would be “What if the fans turned off?”.

These questions are also unique in the way they are answered. In any other kind of question, epistemic or not, the controller searches the set of Event Calculus predicates that the user currently sees in order to procure an answer. In the case of “what if” however, the user is asking about a “hypothetical” state of the world - which is not directly available to the controller. In order to generate this world state, the controller has to generate an EC predicate corresponding to the event or fluent the user asks, append it to a temporary initial state, and then run the reasoner in order to generate EC predicates for this hypothetical world. It’s within those predicates that the controller will look for the proper answer to the user’s question. Below we have an (implemented) example of such a “what if” question:

WHATIF1
What if <agent/arg1>
Moves <obj1/arg2>
<position/arg3>
<obj2/arg4>
[at time <time>]?

Answer:
The chatbot should run the reasoner and generate predicates for the event that the object is moved towards the designated position at the designated time (if no time supplied, assume current time). Those predicates are the result that the user should receive. If the question is invalid, it returns an error message instead.

Section 5.3. Definition of a new question type (with an example)

In all of the above paragraphs, we saw different examples of question types that our system can handle. But what if an operator desires to implement their own question types in order to match their needs? In this section, we will present a methodology for developing new question types suitable for any kind of use of our architecture.

Before we declare our new question type, there are some key points that we must keep in our mind:

- What will be the type of the question we want to implement? Will it be a polar question? A “when”? A “where”? An epistemic version of the above? Or a “what if”? The above

decision will be the cornerstone of our question type, as we essentially define its context.

- Will our question be about an event (something that happens instantly) or a fluent (something that "holds" between two time points)? This is the second integral part of our decisions, since we define what kind of objective the question is about.
- What entities will we involve in our question? To be more specific, *what entity* is our question about? All questions are about *something*, and this *something* is obviously something that cannot be left out of our thinking. This also involves *any possible keywords* that define these entities, as well as the fact that we may have more than one of a certain entity in the question.
- How shall the system respond to such a question? Not all questions have a "yes/no" answer, and not all accept the same type of answer. Every question has a different type of answer, and that's something we have to account for when defining new question types.
- Is it possible to omit entities from our question? If yes, what entities can be optionally omitted?

Once we have thought of all of the above, it's time to realize what we have in mind. We begin by defining some examples of our desired question. Suppose that we have a question whenever something is open. Let's see some examples of such questions:

Is the door open?
Is the window open?
Is the bag open?
Is the drawer open?
Is the water tap open?

We then go through the examples and try to form a general "boilerplate" for this question type. As we can clearly see, all of the above examples have this boilerplate: *Is the X open?*, where X is an object that can "open". This X is an entity, so we need its entity name. Let's say this name is <openable>. Thus, our boilerplate becomes *Is the <openable> open?*.

Then, we have to define an appropriate answer for our question. This answer must be *what the user expects to hear* from our system. If, for instance, the user asks about when something happened, the answer should be a time point (or a set thereof), not a "yes" or "no". This answer should also offer as much information as possible to the user, so as to satisfy any possible information need. In our example above, since we are asking whenever something is open or not, the answer to this question should be either a "yes" or a "no".

Finally, we have to decide on an identifier for our question type. This will also serve as the question's intent during the training phase later on. This identifier should fully define our question. It should start with a prefix that defines its type: this being either a "P" (polar), "WN" (when), "WR" (where), "Ep" (epistemic) or "WHATIF" (what if). This will be followed by either "E" or "F", which defines whenever our question is about an event or a fluent respectively. And finally, the code ends with the "Qx" suffix, where "Qx" stands for "Question X", where X is a number unique to this question type identifier. If the question type is epistemic, we may want to use LVx instead of Qx as a suffix, where LVx stands for "Level X" (see the definition of epistemic questions above for more information). Keep in mind that the above process is not

always the norm; we have seen question types such as WnEpELvX or WHATIF that do not conform with it - this norm will work with most common question types. In general, we want our identifier to define these key integral parts about our question: the exact type, whenever it's about an event or fluent and an unique number or if it's an epistemic, it's level of nesting. In our example above, which is a polar question about an event, let's give it the identifier PEQ7.

And now that we have all needed information, let's present a formal definition of our question question type:

Polar question PEQ7: *Is the <openable> open?*
Answer: "Yes" or "No"
Examples:
Is the door open?
Is the window open?
Is the bag open?
Is the drawer open?
Is the water tap open?

As we can see, defining new question types is a pretty straightforward thinking process, which allows for great flexibility in terms of defining question variations. Question types, along with entity definitions define the chatbot's *domain*, which is the context on which it will operate. Although here we presented our domain in the examples, a client that will use our architecture will find it rather easy to define their own domain in order to suit their particular needs. Defining the domain is only the first part of setting up the chatbot; the other most important part of the setup process is training. That is, we have the theory - let's now jump to practice.

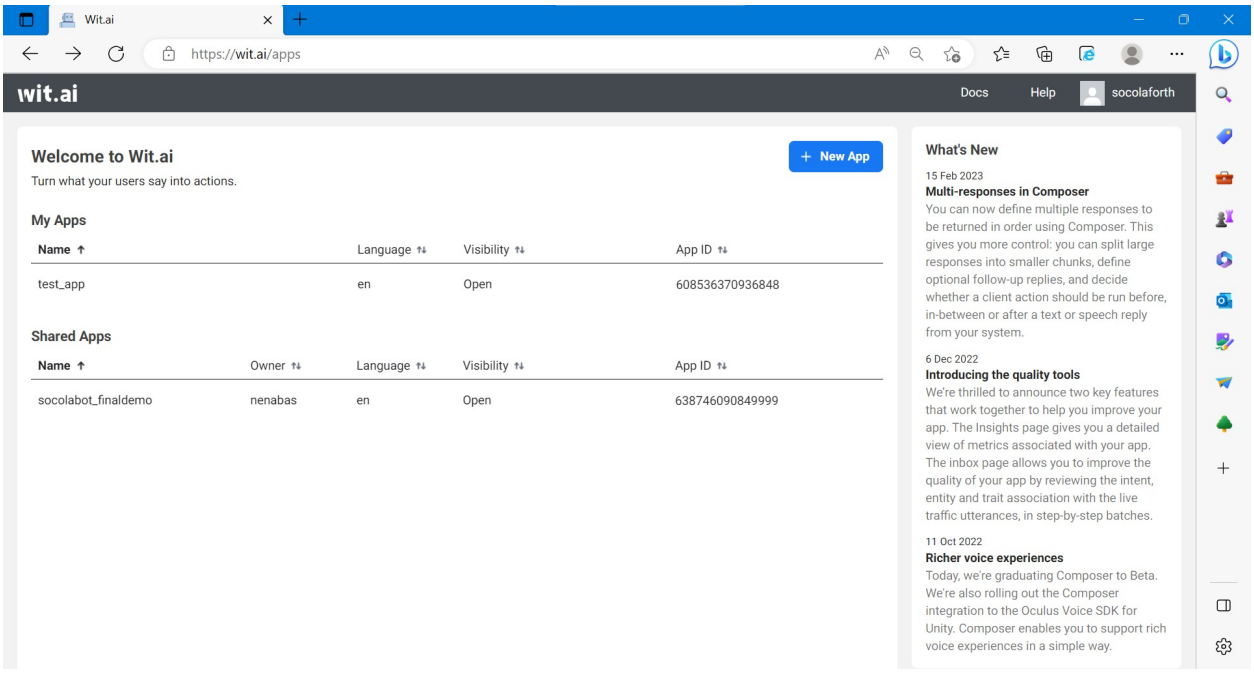
Section 5.4. Training a new (polar) question

As we stated above, training a new question comes in two parts: in the first one, the question must be trained in the chatbot (wit.ai), and in the second, its Event Calculus predicate (and some other information) have to be defined in the controller's parameter file. Let's begin by showing how can we train a new question in wit.ai

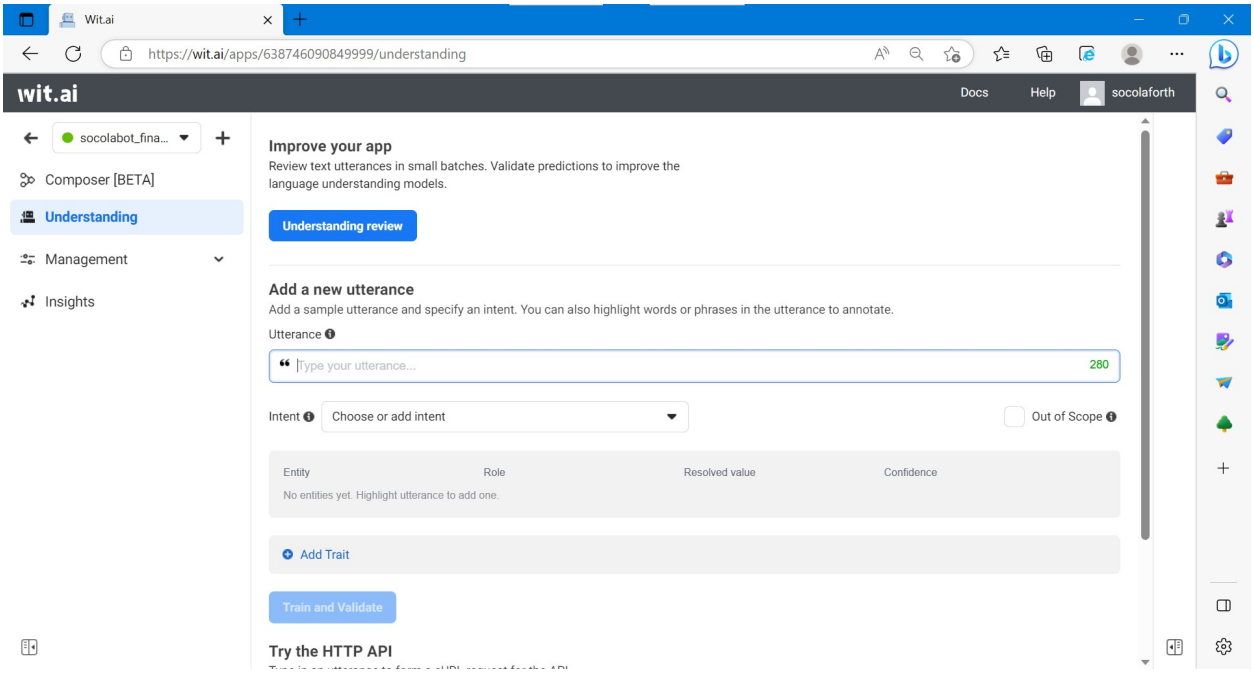
Section 5.4.1. Wit.ai training

The first and most integral part of training a new question is to train the chatbot to recognize it. If we do not do this, the chatbot will not recognize the new question at all. Our "chatbot" here is wit.ai, and we have to train the question there.

First, we go to http://wit.ai and log in with our Facebook account. Once we do this, we see what apps we have already created:

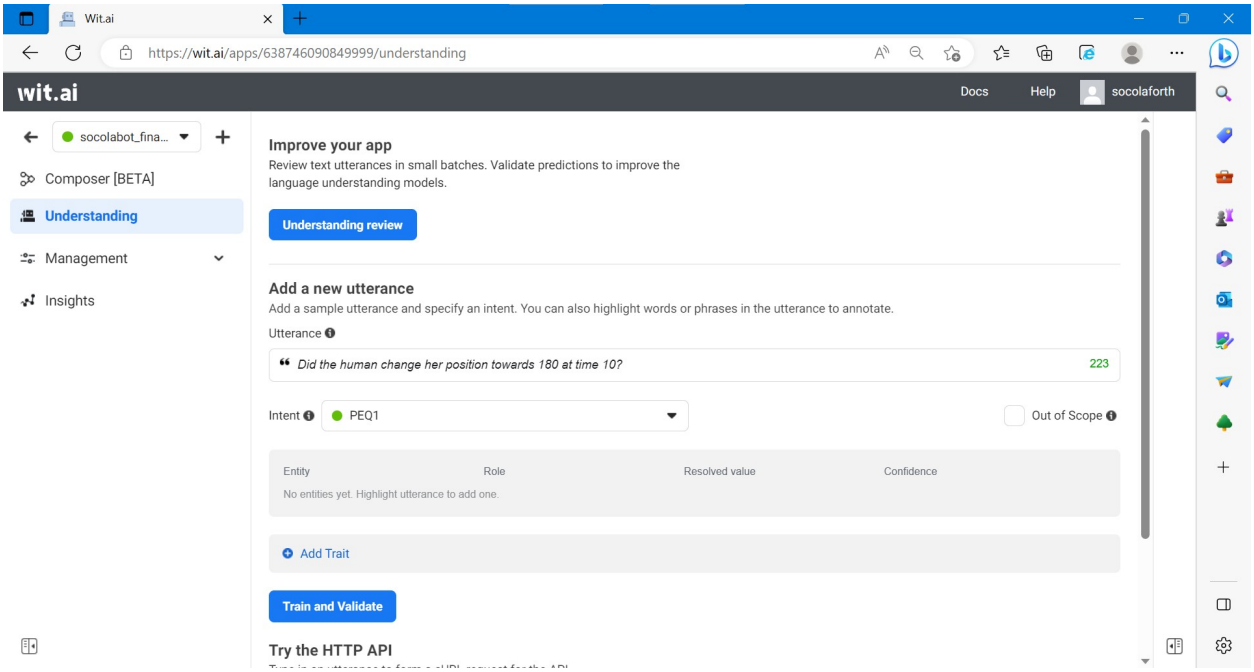


In our case, our app is socola_finaldemo. We click it, and then click on “Understanding” in the left-hand pane. We will be greeted by this training interface:



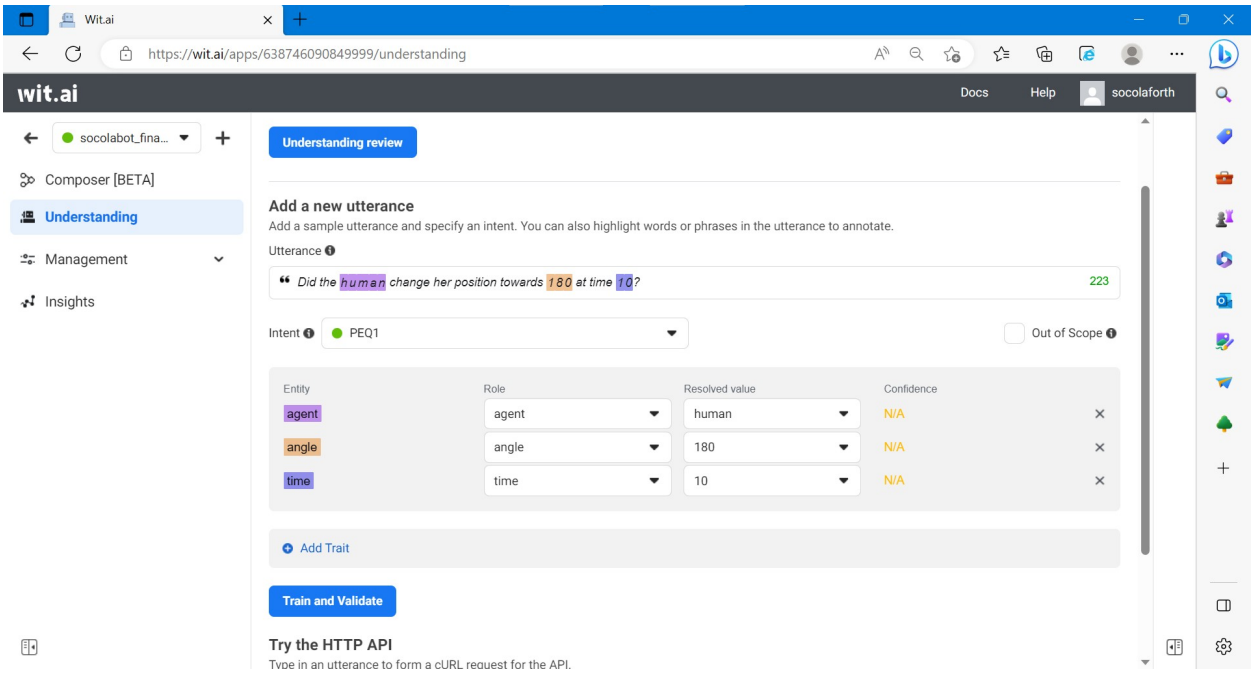
Here, we can enter a new question that *complies to the controlled language specified above,* define its intent, entities and traits and then hit the “Train and validate” button. Let’s input a sample question: *Did the human change her position towards 180 at time 10?* (Remember that

in this example, training is done in our test domain; in an actual application, the domain may be different):

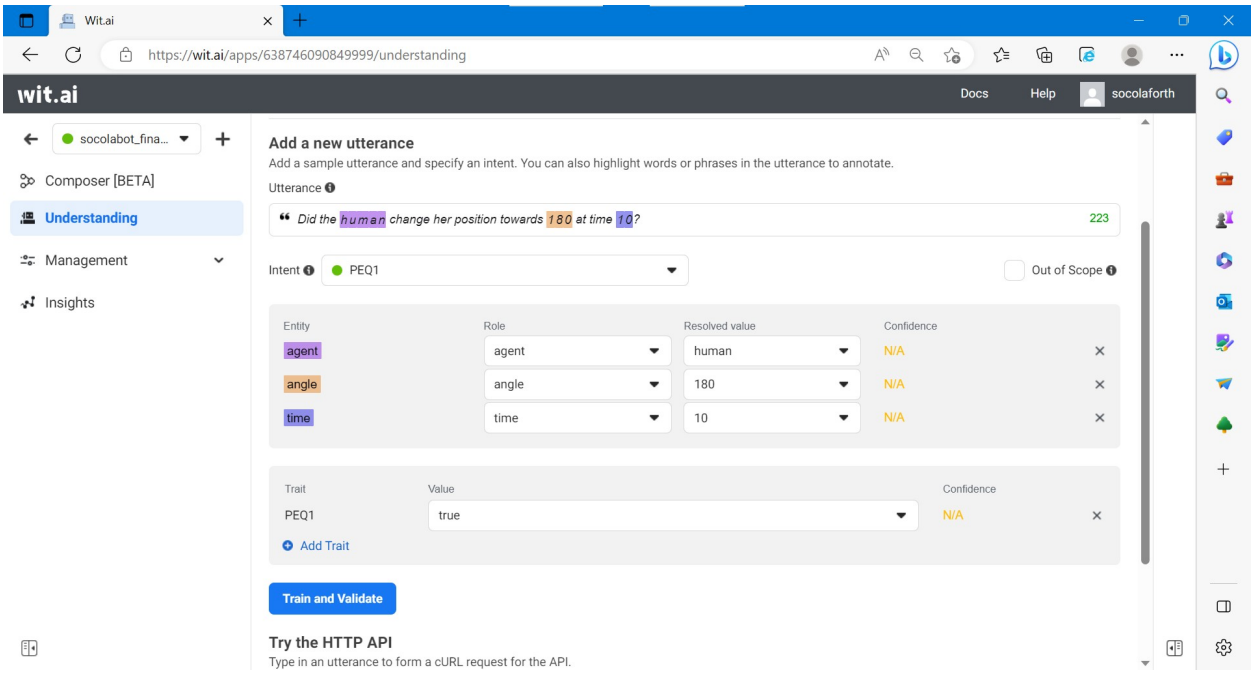


Once we do this, we need to specify the intent, the entities and the trait of the question. We click on the intent menu and choose the question's intent, which is its PEQ or PFQ code (epistemics use an epistemic code such as an EpELv - we will see that later). Here, we choose PEQ1. If the intent isn't in the list, we create it ("Create Intent" button).

Then, we have to specify the question's entities, according to the controlled language. In this question, we have the following: an agent (human), an angle (180) and a time point (10). If we mark one of these, we shall get a drop-down menu with entity types. If the desired entity type is not in the list, we create it. Here, we have "agent" for "human", "angle" for "180" and "time" for "10":



That's all we need entity-wise. Now, for the trait. Clicking the "Add trait" option will present us with just another drop-down menu. We select the trait with the same name as the question's intent (if there is no such trait, we create it). Then, we will have to assign a value to that trait. Clicking on the area beneath "Value" will give us a list of possible trait values. We select "true" (if it doesn't exist, we create it). This trait will be used by the controller to determine the type of the question after it has received its processed version from wit.ai.



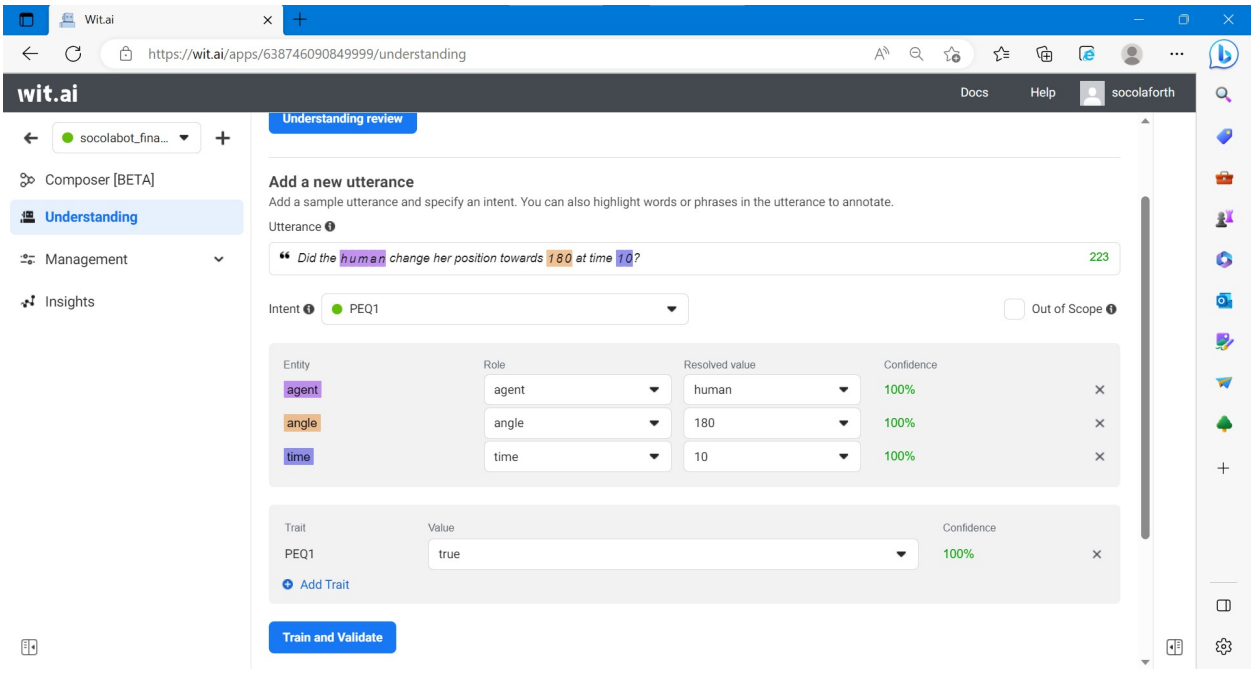
Once everything is set, we hit the “Train and Validate” button, and our utterance will be entered in a training quota, to which more utterances can be added.

In order for wit.ai to understand our utterance better, one instance is not enough. We have to train lots of variations of this utterance, each time with the same intent, entities (it is possible to skip some, depending on the question’s nature) and traits. The entities, however (as well as some parts in the question’s structure) will have to have different values per variation. Let’s see some variations of our example utterance:

Did the human change her position towards 180?
Did the human change her position at time 10?
Did the robot change its position towards 90 at time 1?
Did the human change her position towards 270?
Did the human change her position towards 0 at time 3?
Did the robot change its position towards 90 at time 23?
Did the robot change its position towards 0?
Did the robot change its position towards 270 at time 15?
Did the robot change its position at time 0?
Did the human change her position at time 5?

We train as many variations as possible, so as to make wit.ai understand our question, no matter how we specify it. This way, if the controller receives our question and an entity value is missing (for example, some variations above do not have a value for the angle), the controller will put a “wildcard” on the user question’s missing values and match it with whatever matches with the other entity values, therefore, giving the user more general answers to their question.

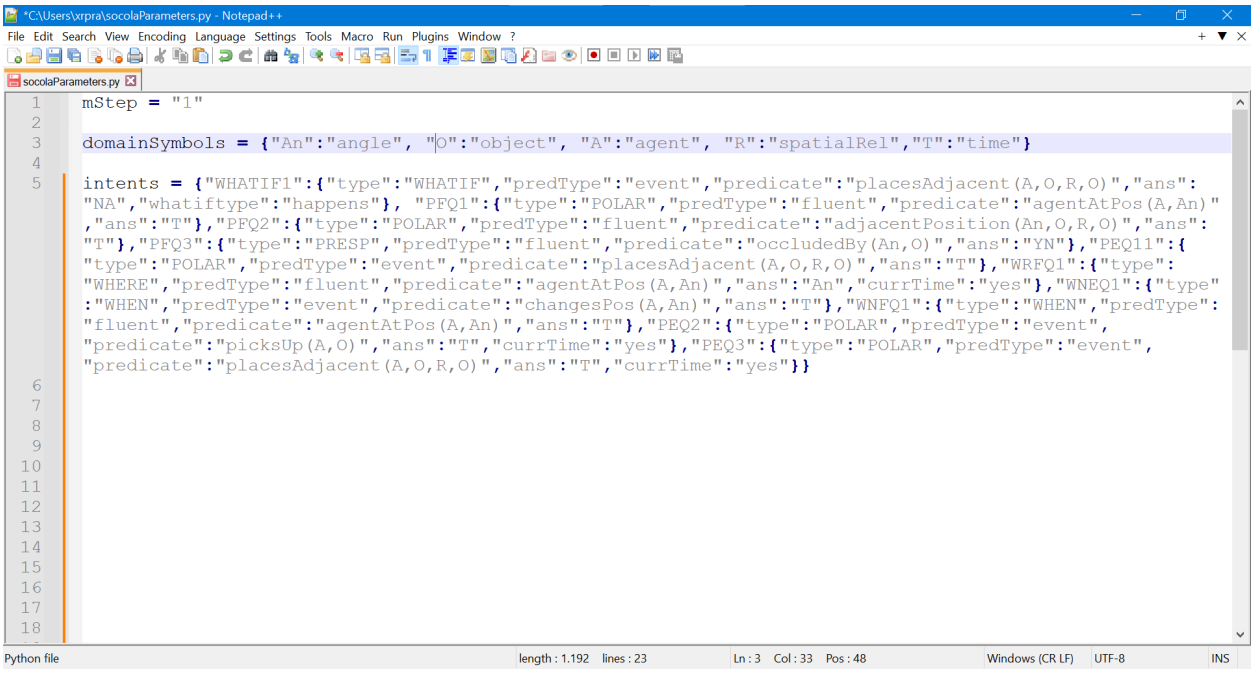
If we train enough variations, in the end, wit.ai will be able to recognize our question’s intent, entities and traits at once. In our example, after we have trained enough variations, this is what we will get if we enter the question in the “Understanding” tab:



We see now how wit.ai has instantly recognized our question, with the correct entities, intent and traits. Now, if the controller sends "*Did the human change her position towards 180 at time 10?*" to wit.ai, the platform will return a JSON to the controller, specifying the questions intent, entities and traits accurately. Now that we have successfully trained our question in wit.ai, let's see how we can get the *controller* to recognize it.

Section 5.4.2. Controller training

Training the controller to recognize a new question type is not a hard job either. All we have to do is to specify the question's intent and Event Calculus predicate in the controller's parameter file, along with some other information, for example, what kind of answer it returns. First things first, let's take a look at the controller's parameter file (socolaParameters.py):



Let's see it line-by-line. On top, we have a maxStep value for starting up the system. This is a deprecated value we don't need, however, we keep it there for testing purposes (we give the maxStep when running the reasoner on the initial state from the command line). Then, we have a set of domain symbols. These symbols match with wit.ai entity names, and are domain-specific. And below, we have exactly what we need: a JSON object defining all question intents. As we can see, each intent has a number of properties, and these are:

"Type": The exact type of the question. Currently, our system supports these types:

- WHATIF: Questions asking "what if <some event occurrence>". These questions are the only ones that make use of the reasoner in order to find proper answers by appending their predicate in a copy of the initial state and then running the reasoner in order to get a new set of predicates, from which they find the appropriate ones that answer them.
- POLAR: Polar questions, like the one we trained above. These questions have either a PFQ or PEQ code and the answer is either a "Yes" or a "No".
- WHERE: Questions asking "where was". These are rather domain-specific for certain domains. The answer to this kind of question is (almost) always a positional argument, such as an angle or a position in the world in general.
- WHEN: Questions asking "when something happened". The answer is (almost) always a time point or a set of time points.
- PRESP: Another kind of domain-specific question that asks how an object is seen from a certain angle. This type of question is specific to our domain, although it can be also used in similar domains.

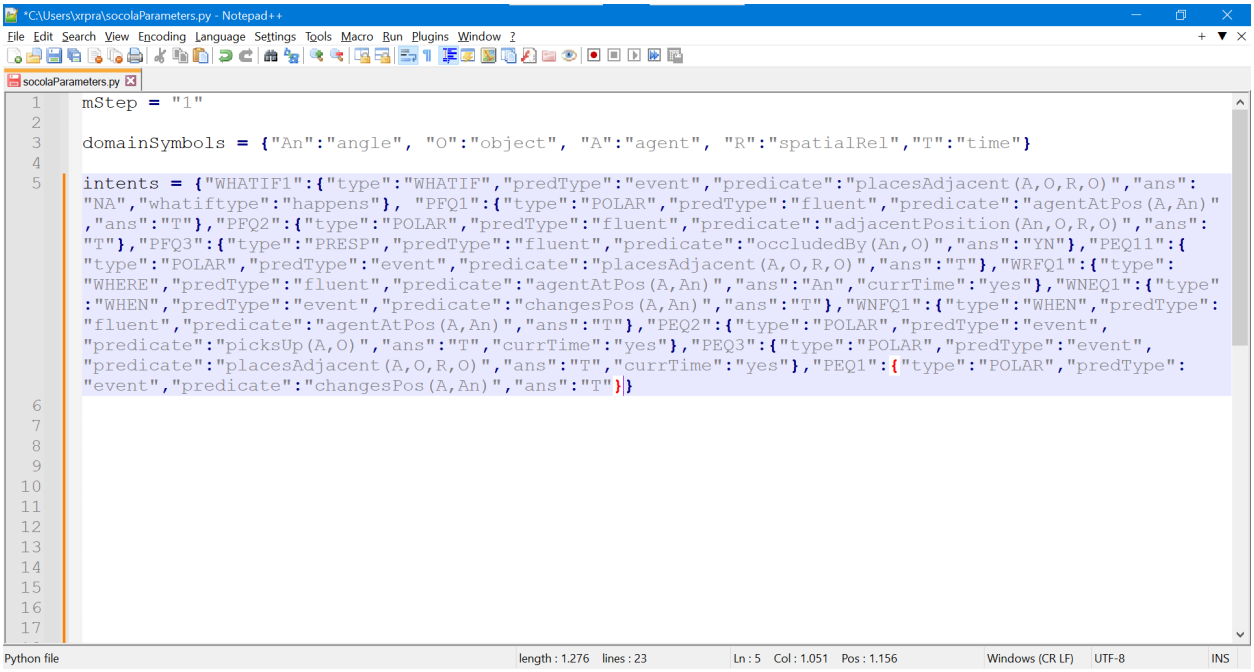
"predType": The kind of event our predicate refers to. It can be either a "fluent" or an "event".
"Predicate": An Event Calculus predicate that corresponds to what the controller has to look up in the predicate list (it has received from the GUI) in order to answer our question.

“Ans”: What kind of answer (entity type) will the controller return to the user if it finds matching predicates. A value of “NA” indicates that the question returns no discrete answer (for example, WHATIF questions). A value of “YN” indicates a “Yes/No” type of answer.
“Whatiftype”: What kind of event happens on a certain what if question. It can either be “happens” (will an event happen if) or “holds” (will a fluent hold if).
“currTime”: If no time point is specified in the question, do we refer to the current time (the latest time point)? If no (or if not defined), return matching answers regardless of time point.

Now that we know exactly what to define in the parameter file, let’s put an entry for our newly-trained question. First of all, it’s a polar question, so its type is “POLAR”. Also, it’s an event. Its predicate, if we mind the predicate list, is changesPos(agent,angle). In our domain notation, it should be chagesPos(A,An). As an answer, according to the controlled language, it returns a time point or a set of time points, so its answer is “T”. And, if we don’t specify a time point, we want to return answers for all possible time points, therefore, we don’t need to specify the “currTime” property (as well as the “whatIfType” property, since it’s not a WHATIF question). Thus, it all boils down to this:

"PEQ1":{"type":"POLAR","predType":"event","predicate":"changesPos(A,An)","ans":"T"}

And we append it to the “intents” JSON as such:

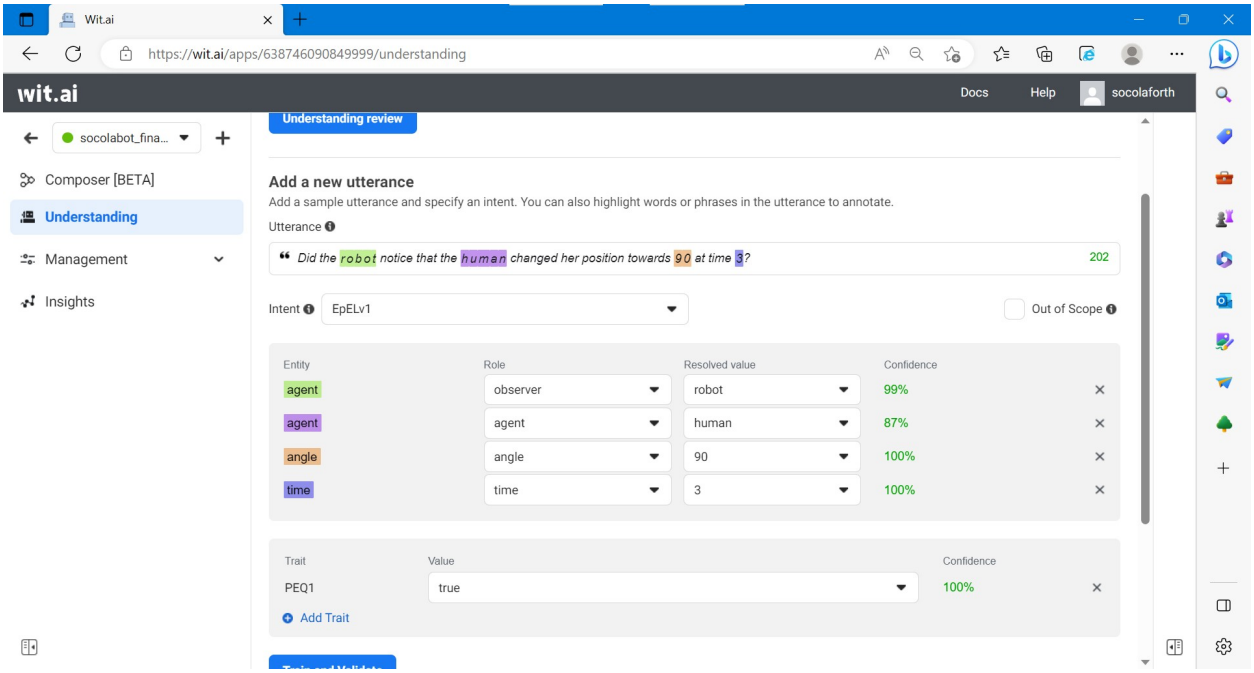


And this concludes the second part of the training process, where we define the newly-trained question in the controller’s parameter file. Now, our controller will be able to handle such questions appropriately, without the need to modify a single line of code in the controller’s executable file.

Section 5.4.3. Training epistemic questions

Training an epistemic question is the same as training a non-epistemic one. The difference here is that we are not asking a question about the world, but we are asking a question on an individual agent's knowledge. As we have noted above, an epistemic question consists of an epistemic part (e.g Did the agent notice…) and a non-epistemic part (an already-trained polar, WHEN or WHERE question).

Let's say that we have this question: *Did the robot notice that the human changed her position towards 90 at time 3?* This is a level 1 epistemic question, with *robot* being the observing agent. We input this to wit.ai as above, and we specify entities, intents and traits:



Compared to the non-epistemic question, there are many differences here. First of all, we need to use an intent that expresses a level 1 epistemic question about an event (the non-epistemic part is a PEQ1, which is an event). According to the controlled language, that should be EpELv1. Also, in addition to the non-epistemic question's agent (human), we now have another agent, the robot. Since we are asking a question on *a certain agent's* knowledge, we will have to designate that agent as an *observer*. Thus, we click on the robot's role and select "observer". We also designate the human as an *actuator,* since it *does* something. Level 2 epistemic questions are about asking an agent about *what another agent* believes, in which case, the first two agents are both *observers*. As for entities other than the agents and the trait, it's all the same as with the original non-epistemic question.

When it comes to the controller part, since we have *already* trained the non-epistemic part of the question, no changes are needed to the controller's parameter file. If so, how will our controller know

that it has received an epistemic version of our question?

The answer lies in the controller's programming itself. When it receives a processed question, it checks the intent to determine whenever it's an epistemic. It does so by looking for the "Ep" prefix in the intent's text. If it finds it, it treats the question as an epistemic. As for the level, it checks whenever in addition to the "Ep" prefix it also contains a "1" or a "2", upon which it treats it as a level 1 or a level 2 epistemic question respectively. It then encloses the respective non-epistemic question's predicate in an epistemic wrapper, which it determines whenever the question is about an *observation* (notices) or a *belief* (believes). For instance, for an event question with predicate pred(X,Y,Z), if it receives a level 1 epistemic, it should look for this:

happens(notices(observer1,pred(X,Y,Z)),timepoint).

If it's a level 2:

happens(notices(observer1,notices(observer2,pred(X,Y,Z))),timepoint).

If it's a fluent, it just replaces *happens* with *holdsAt* and *notices* with *believes.* This way, our controller can easily adapt the non-epistemic predicate to an epistemic one and answer epistemic questions with accuracy and validity.

Section 5.5. Information flow during runtime

Now, let's see how the information flows from component to component when the user asks a question. We will describe the exact paths the information takes from the initial stages, from when the user asks the question until the final answer is delivered to the user.

First, the user asks their question in the GUI. When the user clicks the "Send" button, the raw question is first sent to the controller (along with the current state of the world in Event Calculus, which is kept in the controller's memory), which then forwards it to the chatbot (wit.ai). The chatbot then proceeds to analyze the question and identify its intent, entities and traits. Once this is done, it responds to the controller with a JSON object that contains all of the above information.

The controller then proceeds to use this information to generate an Event Calculus predicate that corresponds to what the user is asking for. It then searches all world predicates and finds the ones that match with this one. If it finds any, it compiles an answer, with the exact information the user asked for (e.g a time point or an angle). If there are no matches, then the answer is a "No results found" message. In any case, the controller sends the answer to the GUI, which then shows it to the user.

The above procedure is the standard procedure for all kinds of questions, non-epistemic and epistemic alike. In the case however of WHATIF questions, things are a lot more different. Here, we want to answer a question in a hypothetical world where what the user asks for happens at a certain time. To create this hypothetical world, we need to modify the initial state by adding the user's

desired event or fluent in Event Calculus notation and then running the reasoner in order to create predicates that describe the state of this hypothetical world.

In this case, the controller calls clingo in order to perform reasoning operations (after adding the user's desired Event Calculus predicate in the initial state) in order to generate the hypothetical world that will be after the user's event happens. Clingo performs its reasoning operations and returns a set of Event Calculus predicates to the controller, which then looks inside it for the answer to the user's question. The answer will be found the same way like other types of questions, albeit the predicates searched will be those of the hypothetical world. The answer will be then delivered in the same manner as with other questions.

Section 5.6. The architecture in action: running example

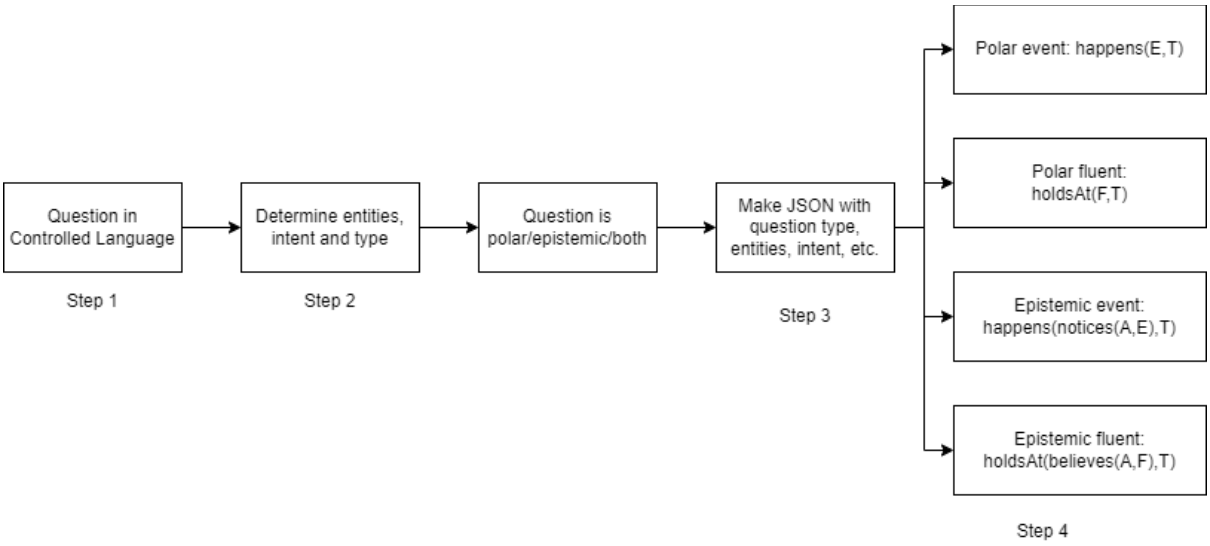
Now we'll take a look at the information flow during runtime step-by-step and part-by-part. First, the user enters the question in the GUI and hits the "Send" button.

The controller then receives the natural language question and forwards it to wit.ai for processing. Wit.ai receives the natural language question and marks its intent, entities and traits. Suppose the user gave what we trained above. If it's trained correctly, this is what the JSON wit.ai will send to the controller look like:

{"intent":"PEQ0","entities":{"agent":["A"],"object":["B","C"],"spatialRel":["onTopOf"]},"traits":
["PEQ0"]}

Now, the controller knows exactly what the user asked for, so it opens the parameter file and gets the entry for "PEQ0" - seen above. It now has the following knowledge about the user's request: it's a polar question, asking about an event, corresponding to predicate move(A,O,R,O) and with the entities and traits described above. Also, it should check the latest timepoint for possible answers. Thus, the controller proceeds to find the latest timepoint in the environment, as well as replace arguments in the EC predicate with the user-given values. It must then form the EC predicate it must look for (or at least, whatever matches it the best). Since it's an event question, and an event *happens*, it should enclose it in a *happens(*, thus, the controller will have to look for *happens(move(A,B,onTopOf,C),T),* where T is the latest timepoint. The above process can be summarized in the following diagram:

In step 1, the user gives the question in a natural language, conforming to a Controlled Language rule. Step 2 is where wit.ai recognizes the question and processes it. In step 3, it sends a JSON with the processed question to the controller and in step 4 the controller forms the query it must look for.



Once the controller has the EC predicate ready - and has received the environment's EC predicates along with the user's question in step 1 - it proceeds to look for anything that may match with the "user-provided" EC predicate. If it finds anything that matches, it checks for what the answer requires. If it requires a timepoint, it records the match's timepoint. If it asks for something else - e.g an angle - it checks the matching predicate for the value of what the answer calls for and records it. Once it finishes searching for matches, it prepares an answer on what it found and if. This answer is sent to the GUI, and the user reads it.

Note that in this case, we did not use the reasoner. The reasoner is used only if the user's question requires reasoning to be done, such as if the user asks what would happen if someone changed something at a certain timepoint. In this case, the EC predicate from the user's question is added to the initial state, the reasoner is run on it and the ensuing search - and answer formation - is done on what the reasoner returns, not what the controller received from the GUI. The answer is sent to the GUI regardless.

Section 5.7. Summing up

In this section, we took a total look at our system and described in detail how one can train new question types for use with our system, along with a detailed example of how a question is processed. As we saw, the training process is pretty much straightforward and requires no changes to the controller's (or any other component's) code, since the controller can modify predicates to match certain question types by itself, thus saving user time when it comes to training the system to recognize new questions. This degree of flexibility also enables one to train huge volumes of questions in a short time, resulting in powerful chatbots with a panoply of capabilities. We can now see how our pipeline exploits flexibility and ease of use to create state-of-the-art chatbot systems, capable of any task that requires human-computer dialogue.

And thanks to our system's modular nature, there are opportunities for a vast number of improvements in future revisions, allowing us to add and remove new components at will, without the fear of breaking everything. If something malfunctions, it's pretty easy to isolate the defect and debug it. This adds an additional degree of security, making debugging and maintenance an easy task.

Chapter 6. Scenarios and Use Cases

We will now present a number of usage cases that all pertain to a common scenario: one where the user asks a repertoire of questions about the world and the agents. We assume that the user is familiar to some extent with the domain that the chatbot can explain, meaning that the questions will contain references that the chatbot can correlate with the predicates of the underlying logic-based axiomatization; yet, we do not require the user to know exactly the names or the signature of these predicates. Proper training of the chatbot is a key factor of any chatbot system, as it both enables end users to express themselves in a free style, while also helps the system recognize more accurately the query's intentions. These use cases demonstrate the chatbot's abilities to their maximum extent, testing both epistemic and non-epistemic question answering capabilities, as well as the ability to form scenarios based on user input ("What if?" questions).

Section 6.1. Case 1: Typical non-epistemic questions

In this first scenario, we ask for information regarding the placement of an object, and how other agents perceive it. Suppose that the object in question is boxA. The questions we are going to ask here will demonstrate how each question receives a proper answer. We may ask some of the following questions:

Polar questions
-Is boxA leftOf boxB as seen from 180 at time 3?
-Is boxA leftOf boxB as seen from 180?
-Did the robot place boxA left of boxB at time 0?
-Did the robot pick up boxA at time 1?
-Did the human change her position at time 1?
-Did the robot pick up boxA at time 2?
-Did the human place boxA on top of boxB at time 2?

Where questions
-Where is the human located?
-Where is the robot located?

Counterfactual questions (occlusions)
-Is boxA occluded by boxB to the human?
-Is boxA occluded by boxB to the robot?

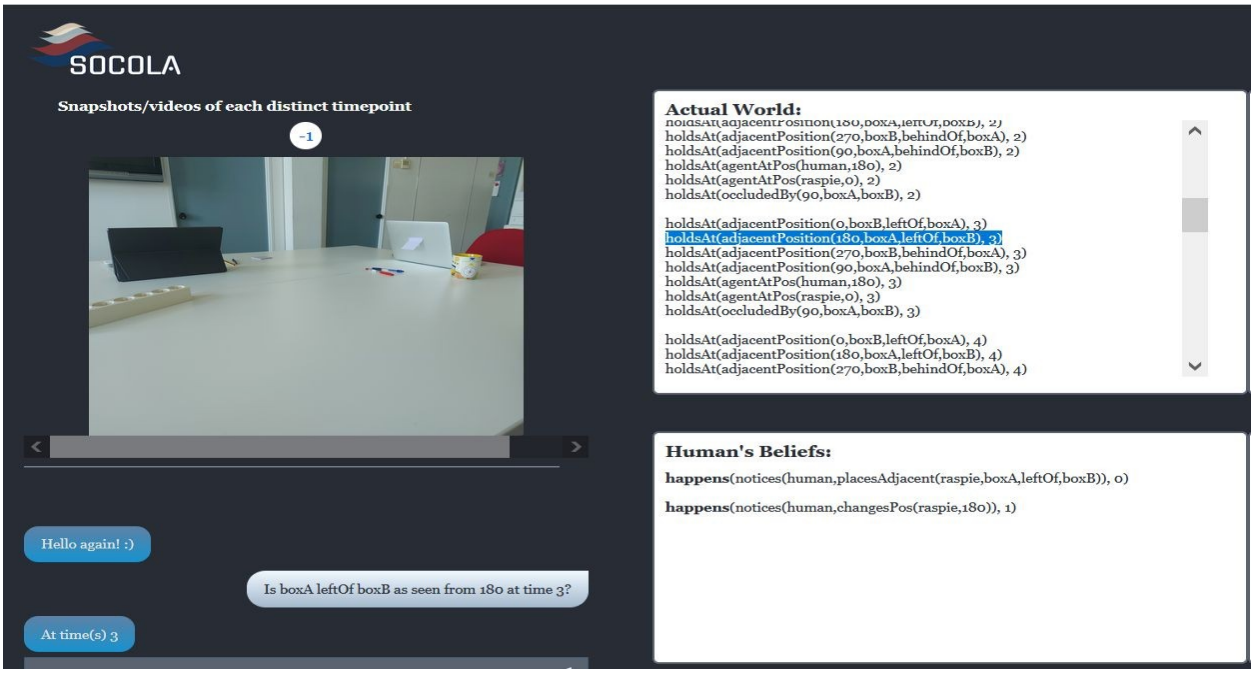
What if questions
-What if the human moves boxA behind of boxB?
-What if the robot moves boxA left of boxB?

-What if the human moves boxA on top of boxB at time 2?
-What if the human moves boxA left of boxB at time 1?

When questions
-When did the human change her position towards 0?
-When was the human located at 90?
-When was the human located at 180?
-When was the robot located at 0?

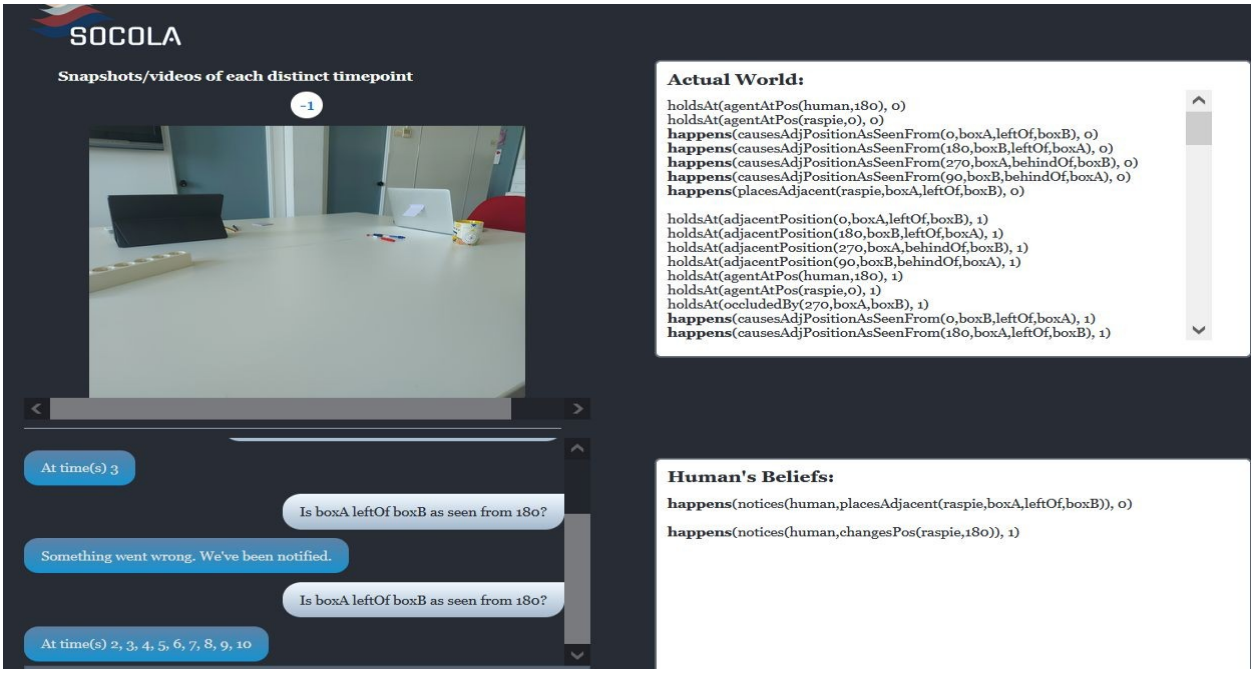
Notice that, in the aforementioned questions, regardless of their type, the user may ask either about an event occurrence or about the state of a fluent; it is the chatbot's responsibility to understand what the user is looking for.

Let's begin with the polar questions. We fire up the GUI and run the reasoner in order to load up the predicates describing our world after a given narrative of actions has taken place. In these running examples, we have run the reasoner with max time point 10. After that, we run the chatbot's controller and proceed to ask the questions. Let's start with this: *Is boxA leftOf boxB as seen from 180 at time 3?*



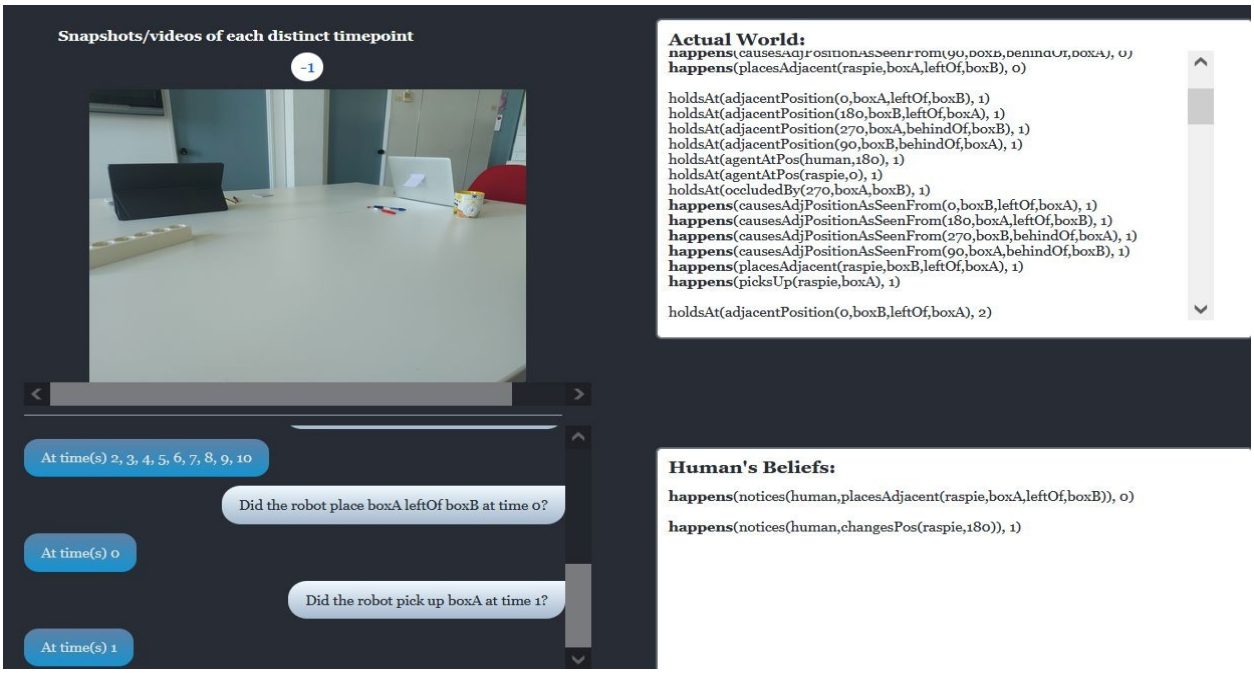
As we can see, our system locates the appropriate predicate (we manually marked it in the "Actual World" field) and returns an answer which tells us at what time point was boxA left of boxB. If it couldn't find such a predicate, it would return a "No results found!" message. If it could find more than one of the predicates that matches with the question, the answer would include multiple time points. We will see this in action later on.

We then ask the same question without specifying a time point, that is: *Is boxA leftOf boxB as seen from 180?.* Technically, this should return *any* timepoint where boxA is left of boxB as seen from angle 180:

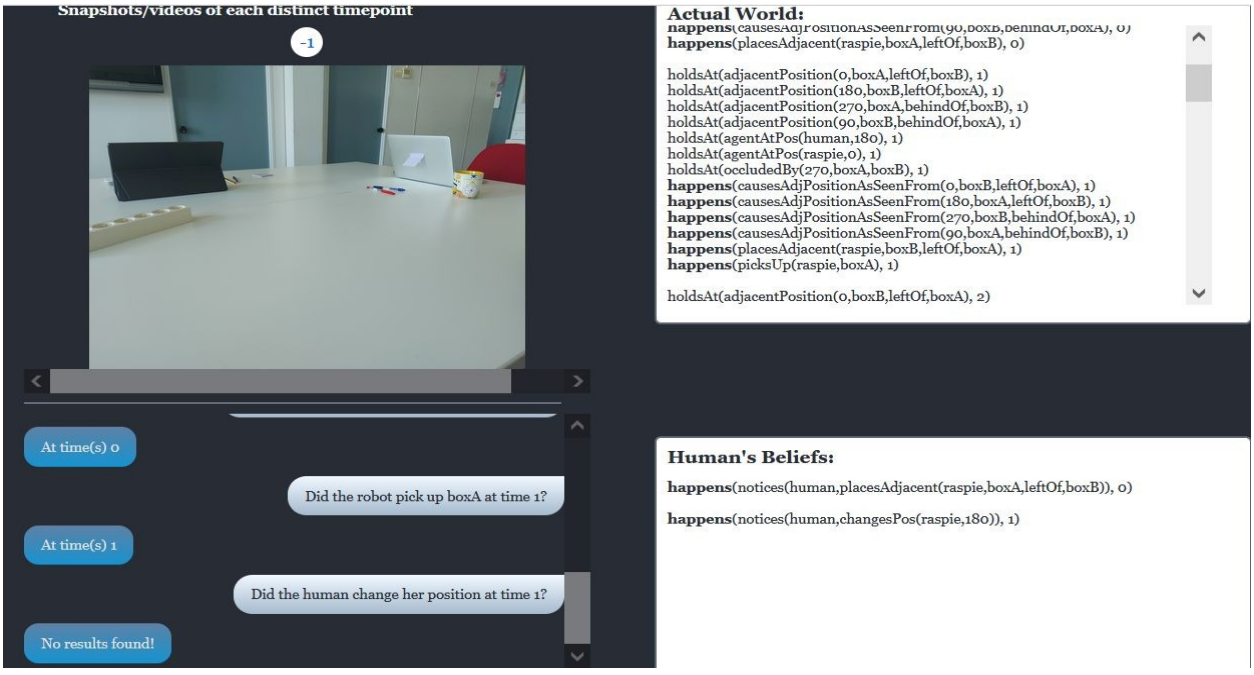


And, as we can see, the system returns *all* time points where boxA is left of boxB as seen from angle 180. Never mind about the error above, connection issues…

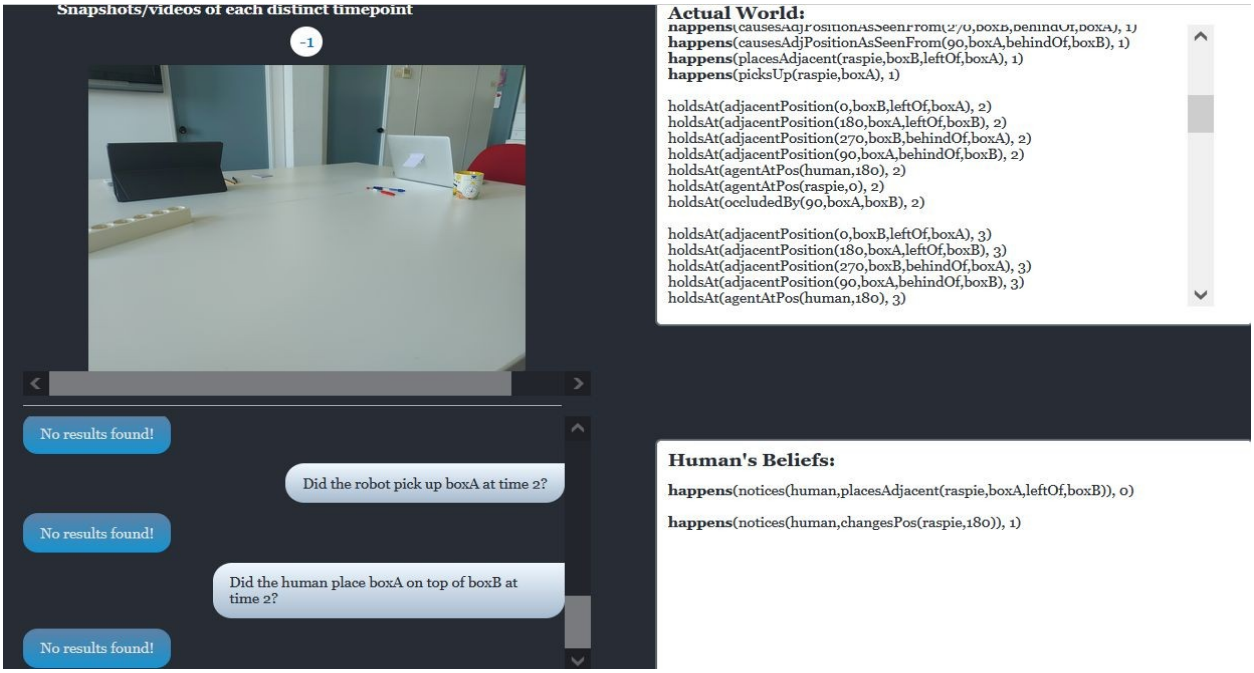
Now we are going to ask two questions about the robot: when it placed boxA left of boxB at time 0 and when it picked up boxA at time 1:



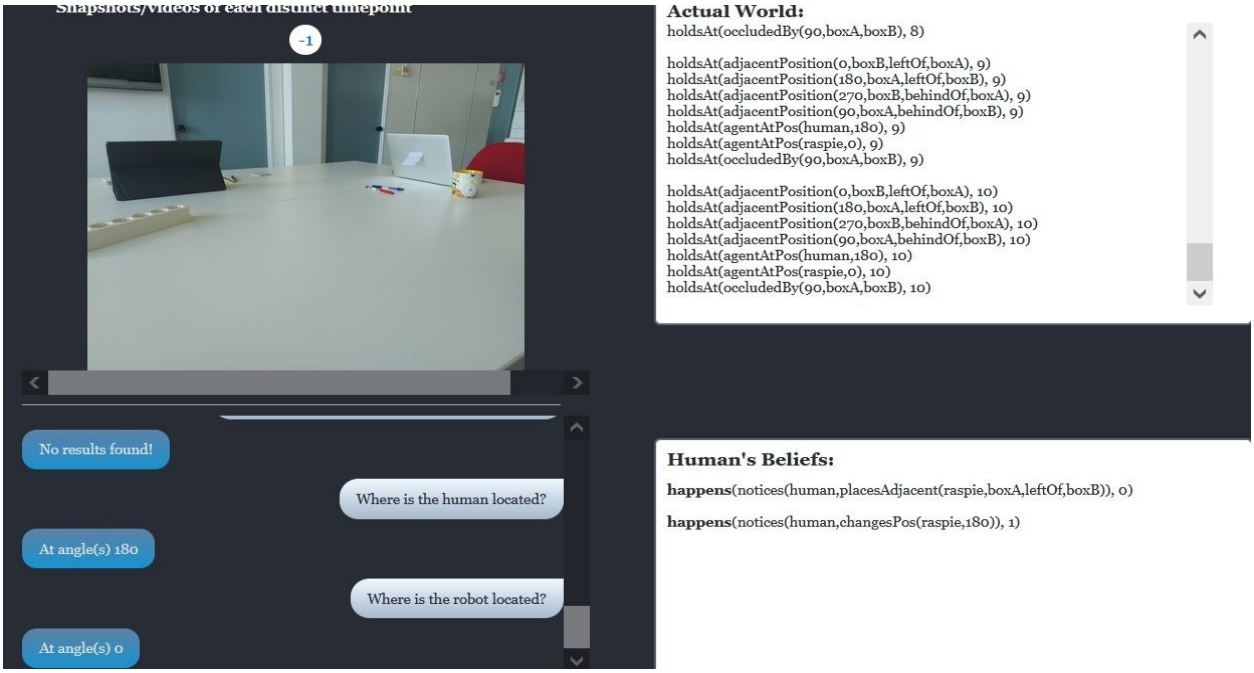
As we can see, our system is capable of handling this type of questions as well. So far, so good. Now, let's ask whenever the human changed her position at a certain time point: *Did the human change her position at time 1?* As we can see, there is no such predicate that describes such an event in the "Actual World" field, therefore, we are expecting nothing to be found. Let's see:



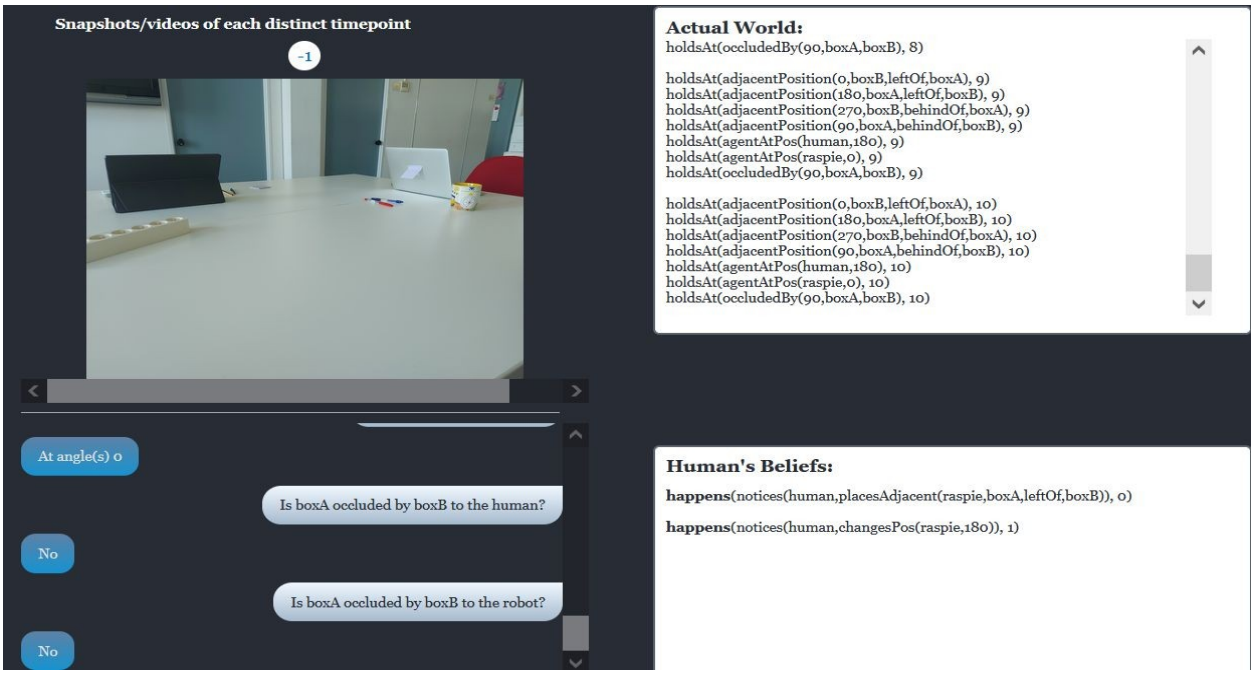
Just as we expected it, no predicate, no results. Asking "*Did the robot pick up boxA at time 2?"* and "*Did the human place boxA on top of boxB at time 2?*" should yield the same answer as well:



And that's all in this case with the simple polar questions. Now, let's check out the "where" questions. We are going to ask wherever the human and the robot are *currently* (that is, the latest time point) located, therefore, the questions are pretty much obvious:



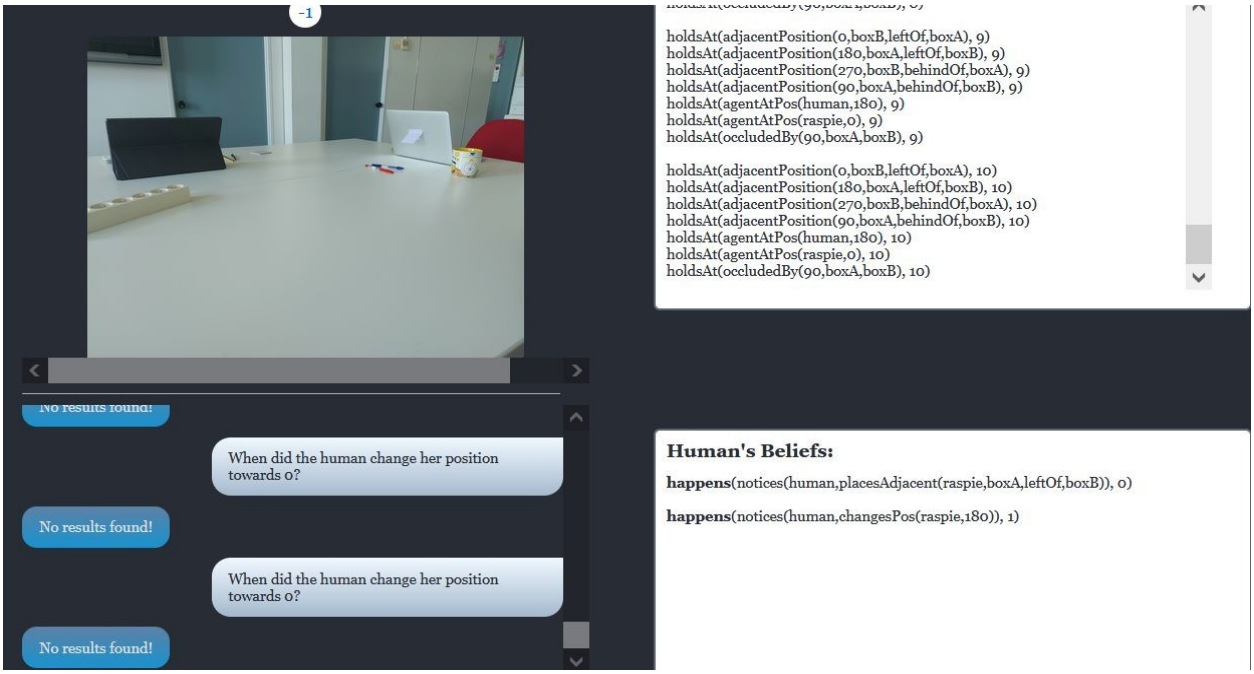
Now, two counterfactual questions: we are going to ask whenever boxA is currently occluded by boxB from the human's or the robot's point of view, that is "Is boxA occluded by boxB to the human/robot?". Let's check this out:



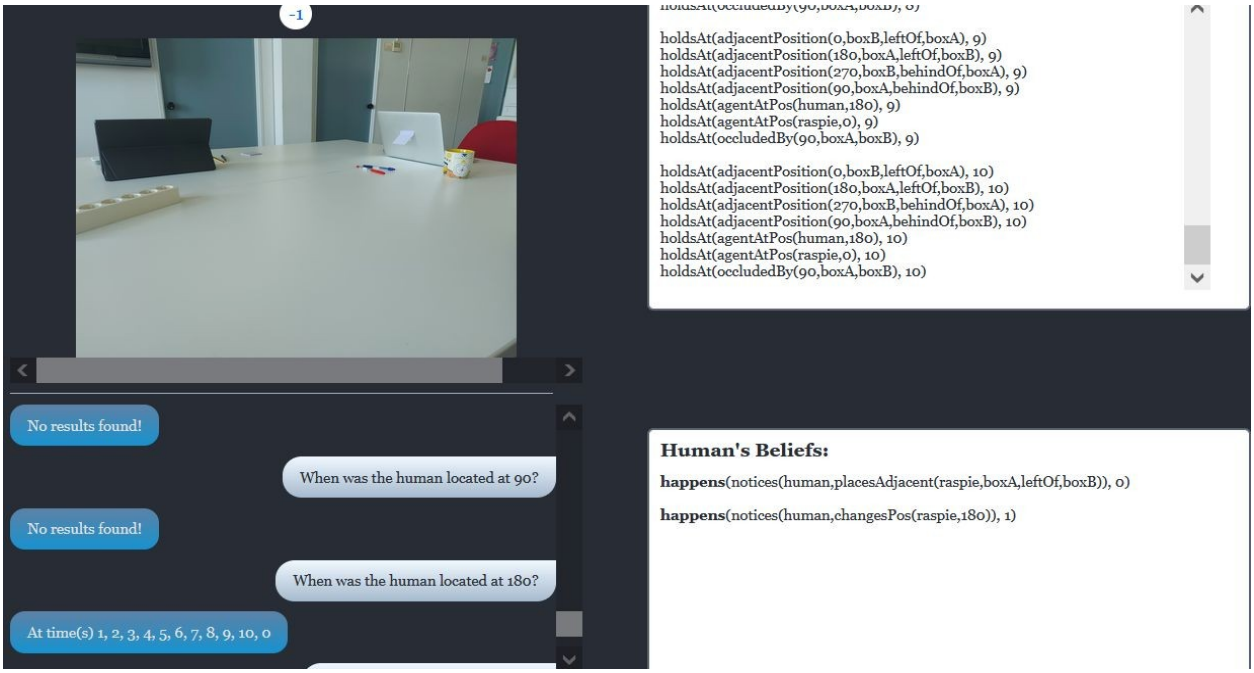
We expected "No" in both cases. BoxA is *indeed* occluded by boxB, but this is as seen from angle 90. As we saw earlier, the robot is located at angle 0 and the human at 180 - that is,

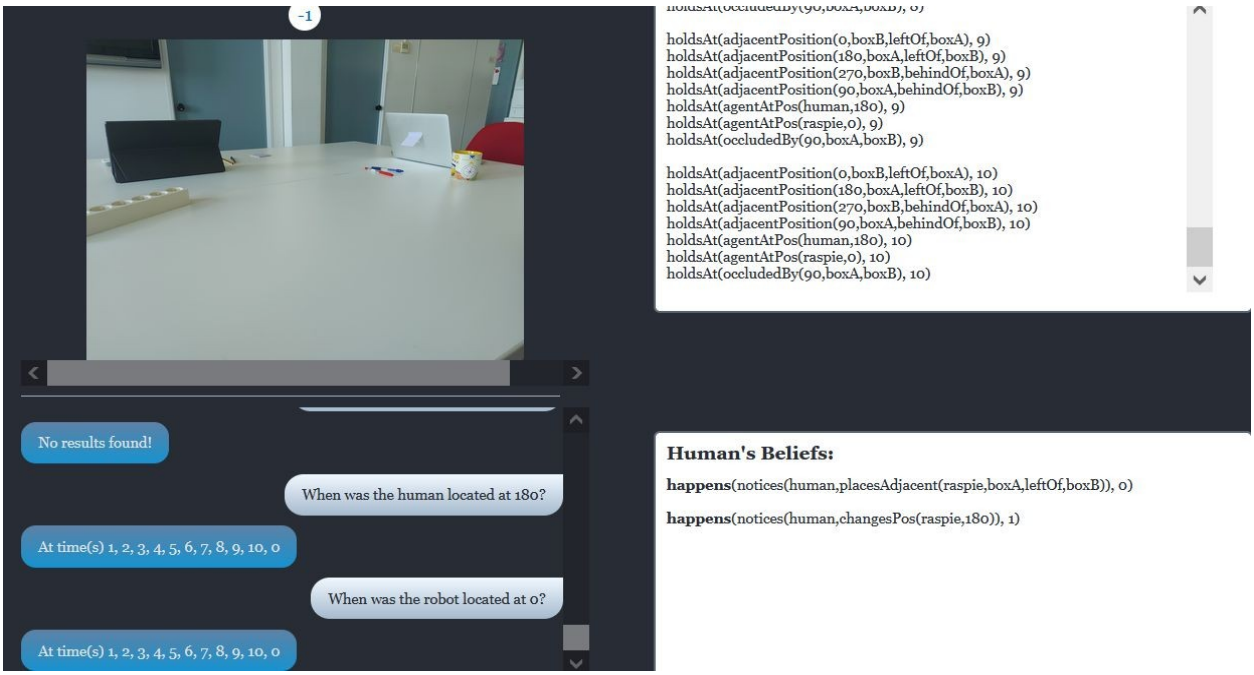
neither is located at 90. Were one of these agents located at 90, one of the answers (or both, if both agents were at 90) would be a “Yes”.

And then we have “when” questions, where we expect a time point (or a set of time points) as an answer. We’re first going to ask when did the human move to angle 0 (don’t mind that I asked this question three times, just testing). Since no such event happened, we should get no results:



Then we ask when was the human at angle 90, when was the human at angle 180 and when was the robot at angle 0. If we check the facts, the human *never* was at 90, therefore, we should get no results here. However, the human *was* at 180 at all time points, and ditto for the robot who was at 0. Therefore, the latter two should return all time points as an answer.

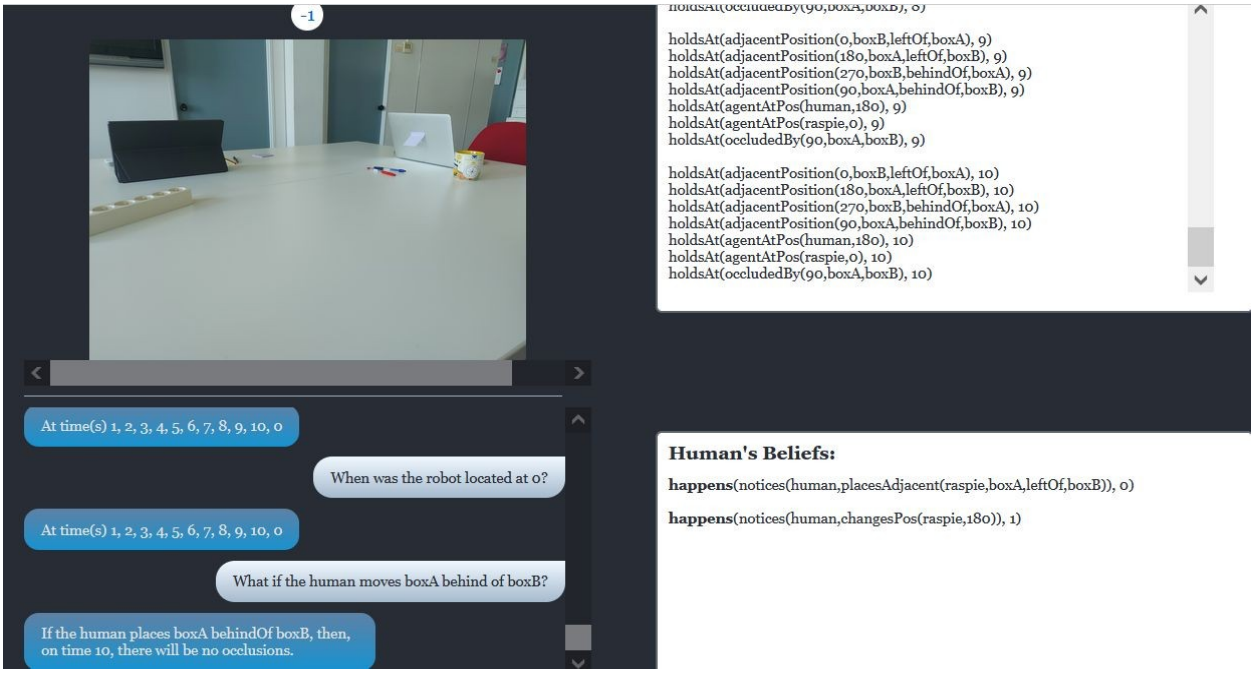




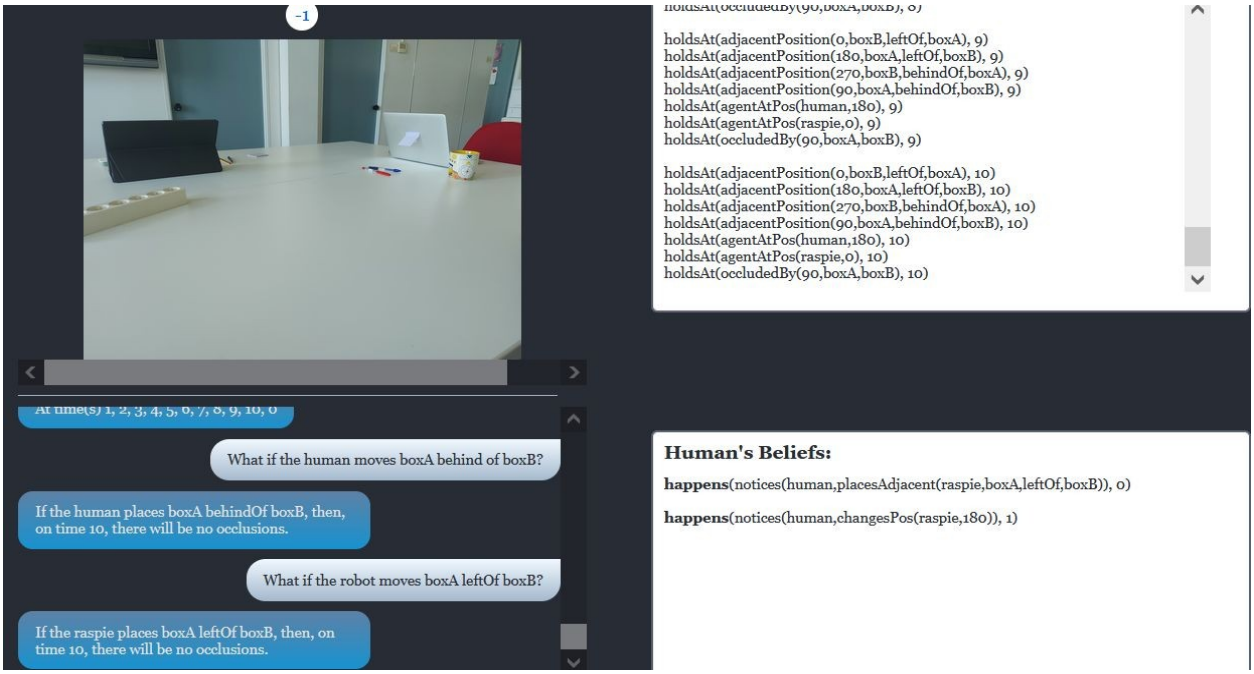
And, as we can clearly see, the “when” questions work well, too.

Finally, let’s check out “what if” questions. As we mentioned earlier, these questions append a predicate corresponding to “what would happen if…” to the initial state and then run the reasoner in order to create an “imaginary” world where the desired event has happened. In our implementation, “what if” questions ask whenever there will be occlusions between objects if an

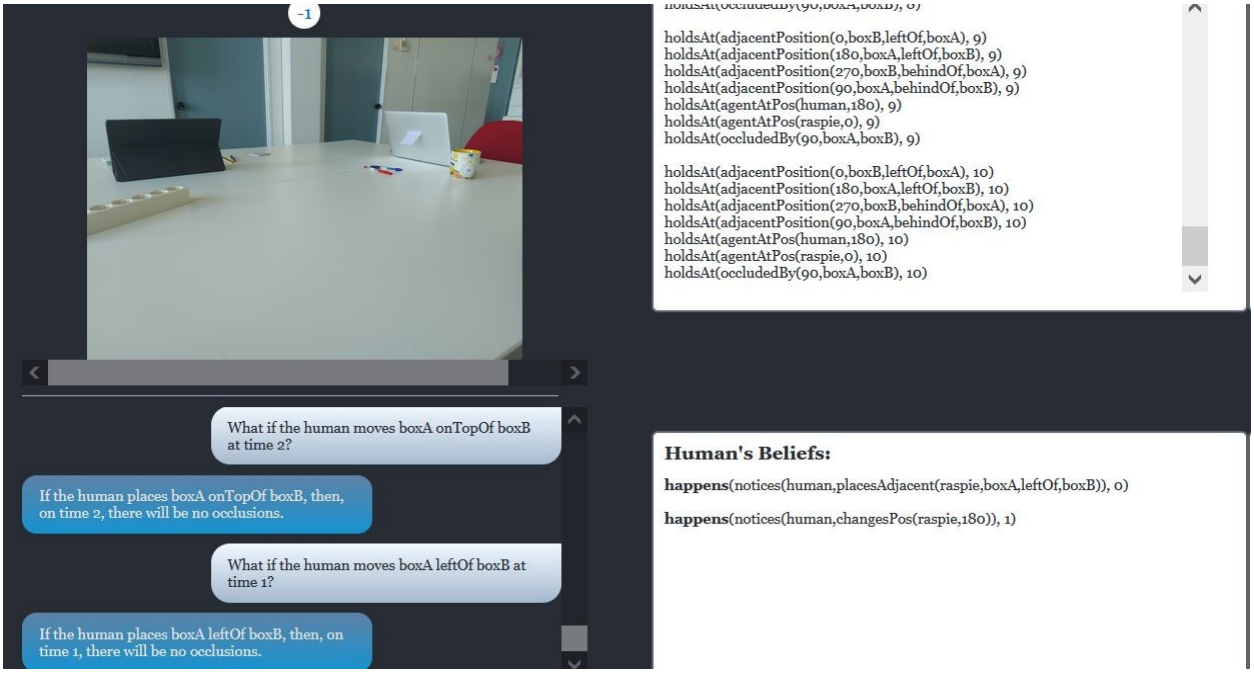
agent placed them adjacent in some spatial position (behind of, left of, etc.). Let's begin by wondering: what would happen if the human moved boxA behind of boxB at the current time point? Thus: *What if the human moves boxA behind of boxB?*



So, there will be no occlusions if the human moves boxA behind of boxB at the latest timepoint. What if the robot moves boxA left of boxB?



Still, no occlusions. OK. And what if the human moves boxA on top of boxB at time 2 and boxA left of boxB at time 1?



No occlusions as well. If there were any occlusions, the system would obviously let us know about them. The “what if” questions seem to be handled by our system appropriately, so they’re good as well.

And this concludes the first usage case of our system, where we showcase its basic question answering abilities with sets of simple questions. In the next cases, we’re going to test it on epistemic questions - both levels 1 and 2 - as well as how the system handles dynamic usage scenarios (where the reasoner updates the states in regular intervals, unlike other cases, where the state is static - that is, the initial state and just that).

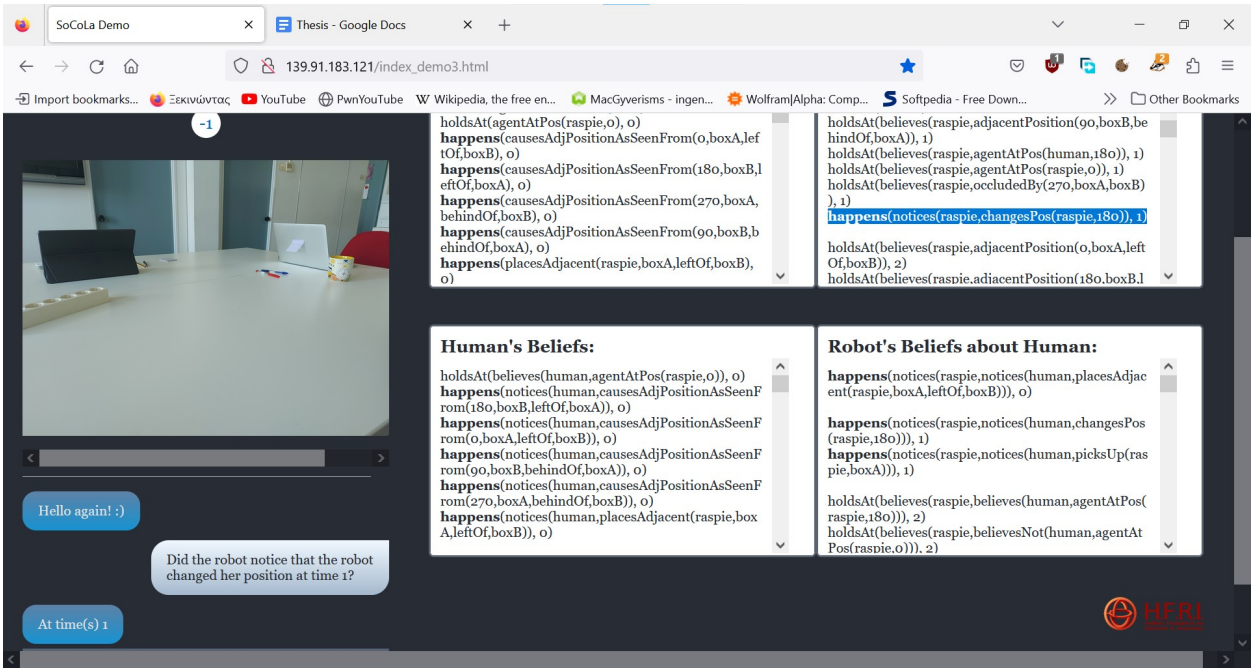
Section 6.2. Case 2: Level 1 epistemic questions

Here, we are going to ask the agents on what they noticed and what they believe about the world. We are repeating exactly the same process as with Case 1, only this time we are asking other agents what they notice and believe, not the actual world.

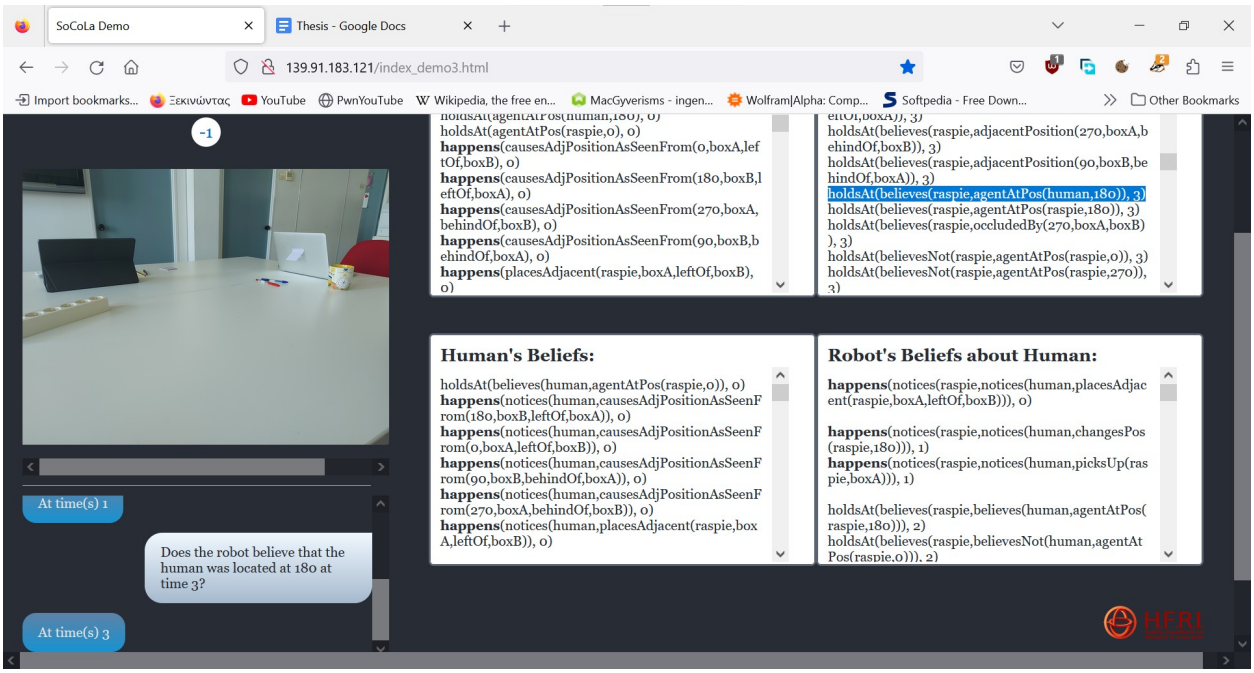
-Did the robot notice that the robot changed her position at time 1?
-Did the human notice that the robot picked up boxA at time 2?
-Does the robot believe that the human was located at 180 at time 3?

-Does the robot believe that the human is located at 90? //Prepei na dw to currtime sta parameters
-Does the robot believe that boxB was behindOf boxA as seen from 90 at time 3?
-Does the human believe that boxB was left of boxA as seen from 180 at time 3?
-Did the robot notice that the human changed her position towards 180?

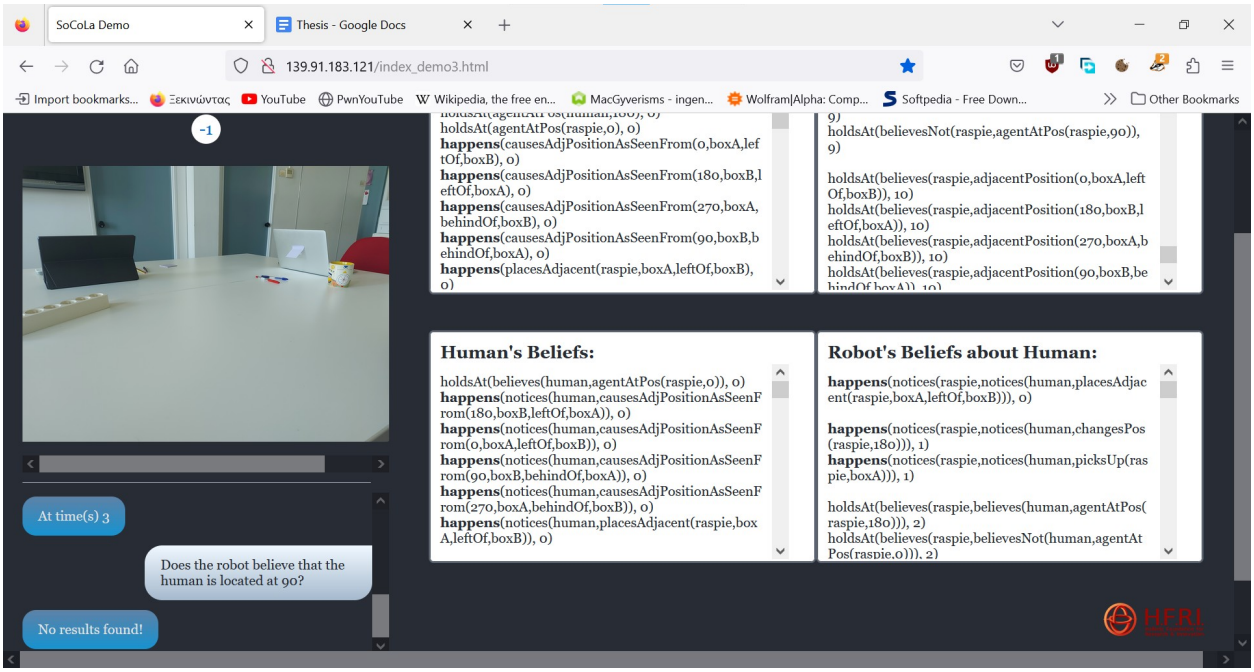
Let's start with the first question: is the raspie aware that it changed its position at a given time point (here, 1)? Let's ask away and find out:



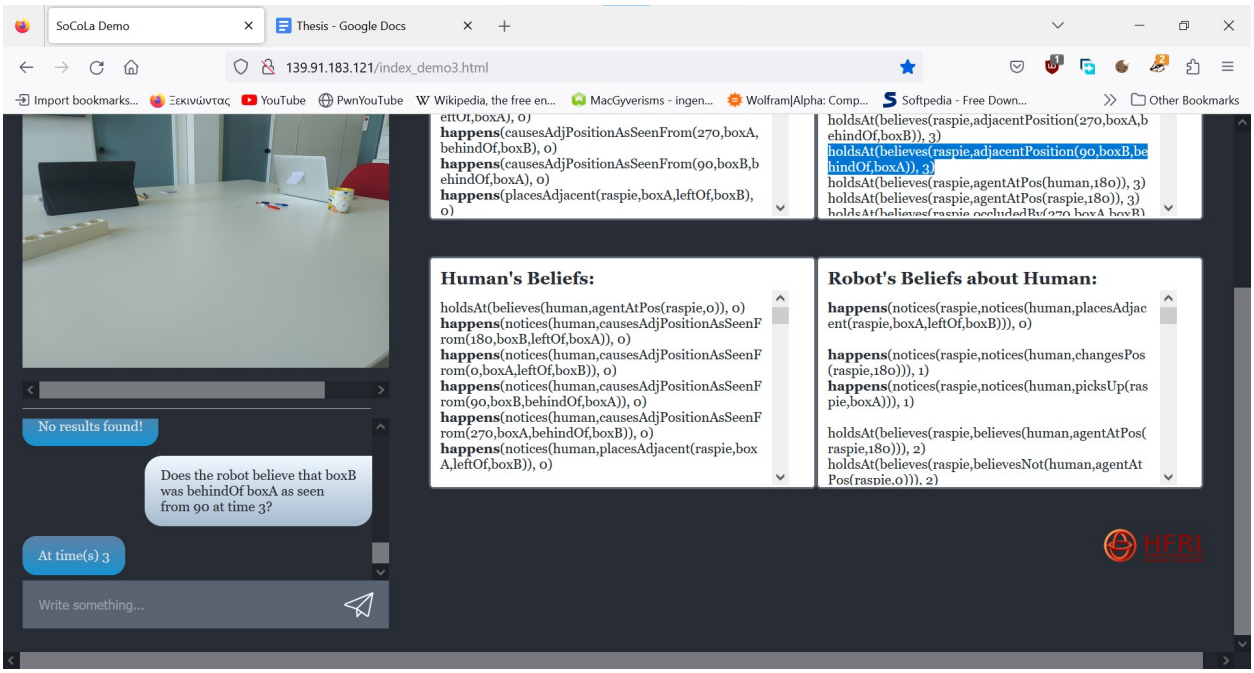
As we can see, the raspie knows that it changed its position at time 1 (there is a respective predicate in the "Robot's Beliefs" area), so the chatbot returns the appropriate answer. Good enough. Now, let's try something else. Does the raspie believe that the human was at angle 180 at time 3? Oh yes, it does, since we have a predicate for this.



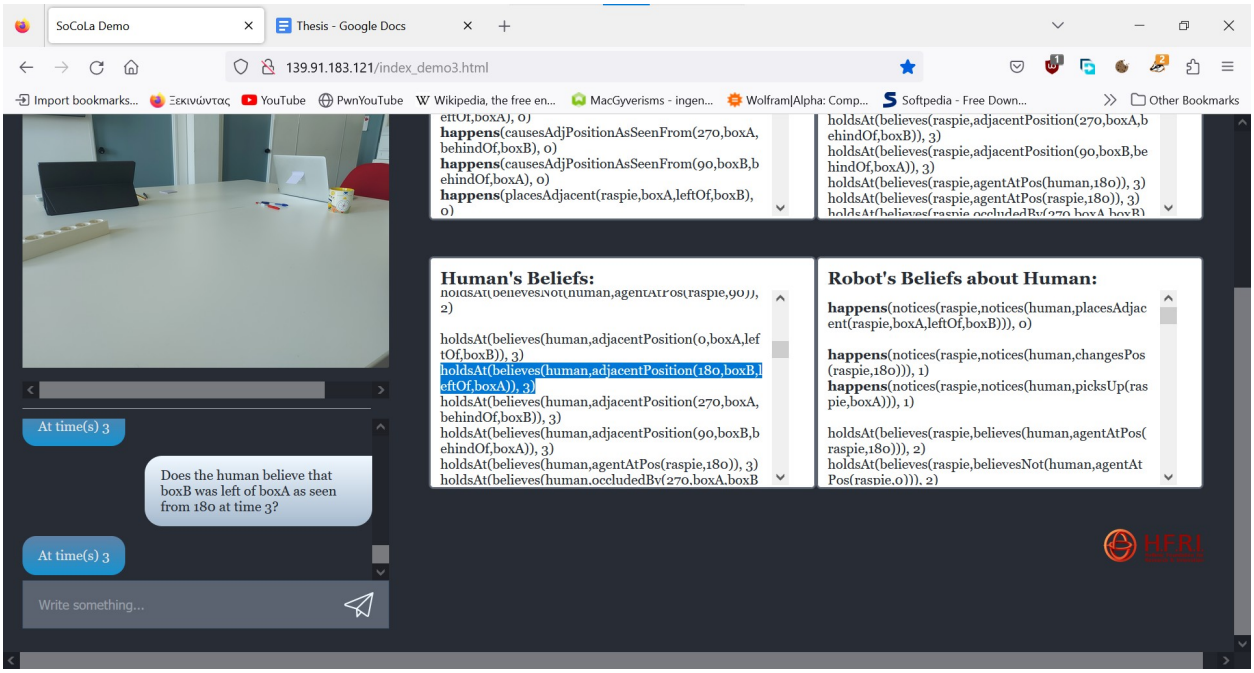
But, what about angle 90? Was the human there at some time? Nope.



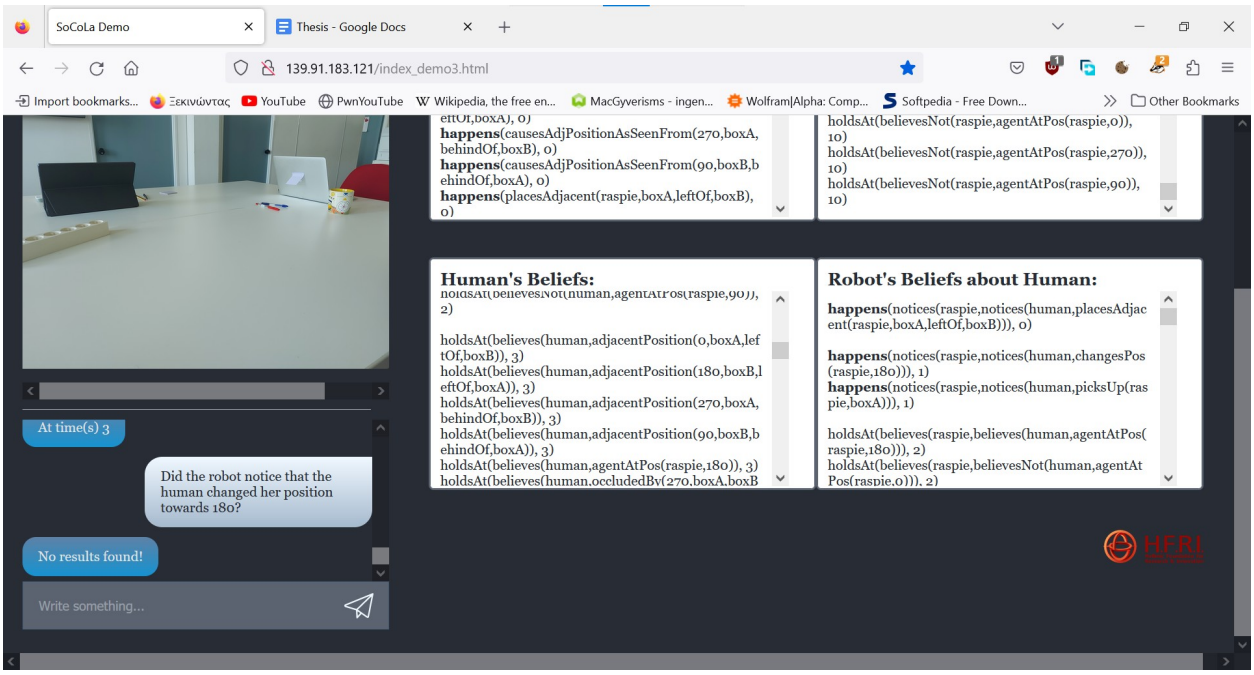
Now, let's ask the robot its opinion on the placement of boxes A and B. If the robot was at angle 90 at time 3, how would the placement of the two boxes look like to it? Simple enough…



So, the raspie believes box B would be left of box A as seen from angle 90. Does the human believe the same as seen from angle 180? Yes, and we have a predicate for that.



And finally, one last question to the robot: did it notice, at any time, that the human changed her position towards 180? There is no such predicate in the robot's "mind", so the answer we should get would be a "no". And a "no" it is.



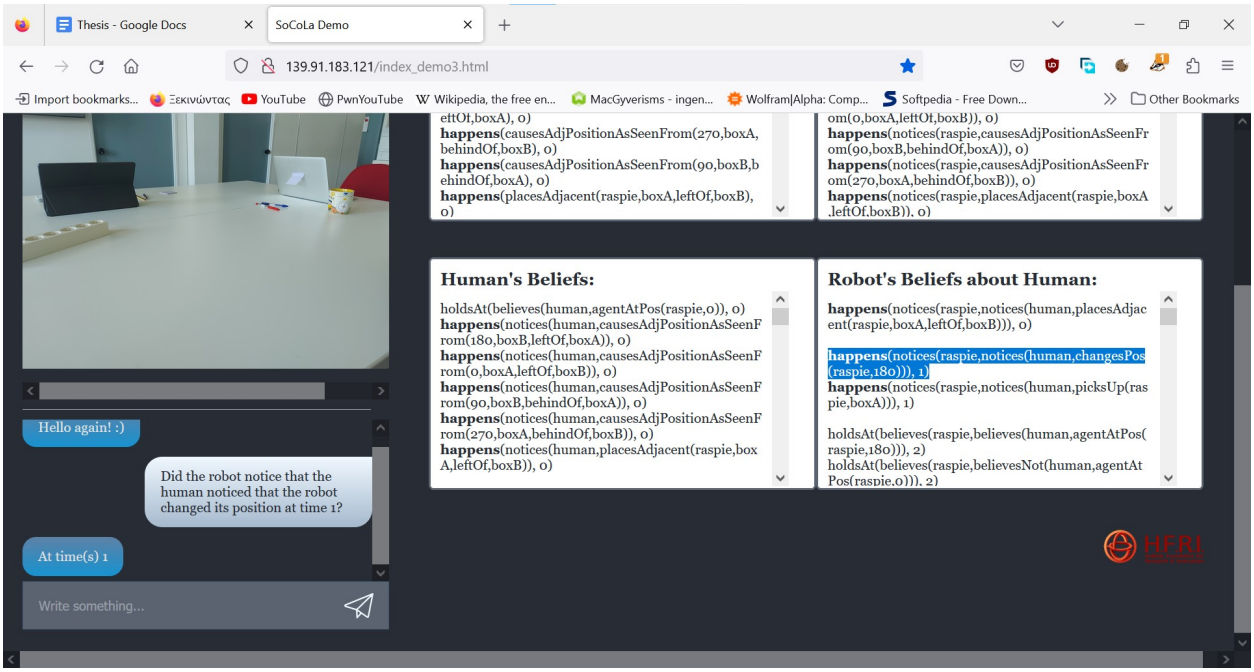
And so, this usage case showcases how easily P-Chat can handle (relatively simple) level 1 epistemic questions - that is, what does *one* observer observes or believes. The ability of P-Chat to answer level 1 epistemic questions can be further expanded in the following case, where it answers level 2 epistemic questions - that is, questions that ask an observer what they notice/believe about *another* observer about what that other observer notices or believes about the world. When we have examined that next scenario as well, we may consider P-Chat ready for action in a dynamic usage case - where the world and the agents' beliefs and observations change over time.

Section 6.3. Case 3: Level 2 epistemic questions

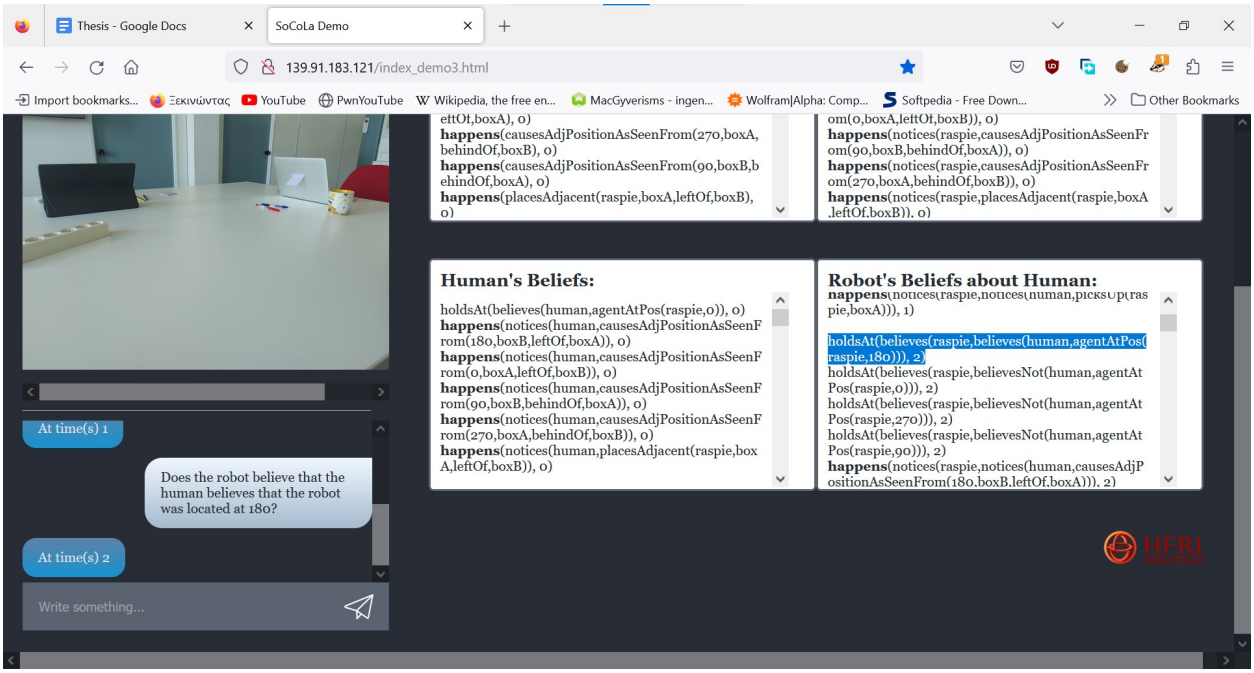
Here, we are going to ask agents questions about what other agents believe/notice. It's exactly the same as Case 2, only this time, we are asking an agent on what it notices/believes about another agent.

-Did the robot notice that the human noticed that the robot changed its position at time 1?
-Does the robot believe that the human believes that the robot was located at 180?
-Does the robot believe that the human believes that the robot was located at 90 at time 5?
-Did the robot notice that the human noticed that the robot changed its position towards 0 at time 3?
-Does the robot believe that the human believes that the robot was located at 180 at time 2?

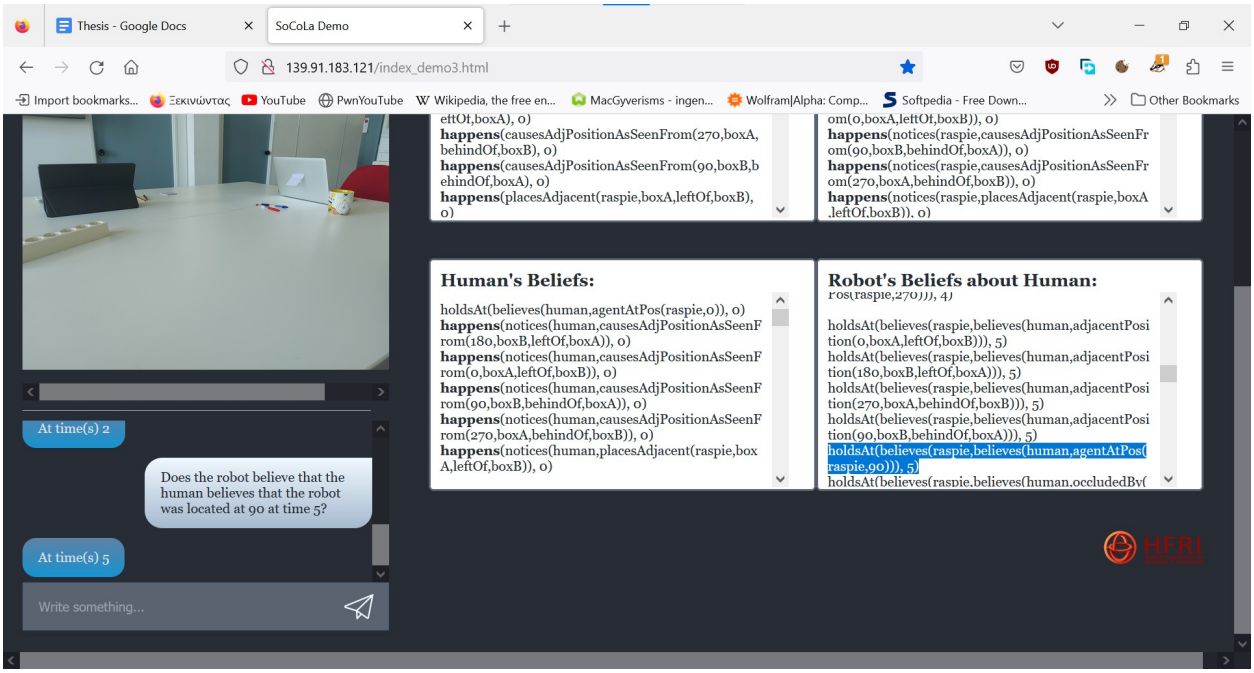
Alright, let's get started. The robot changed its position at time 1. Did it notice that the human noticed that? Let's see…



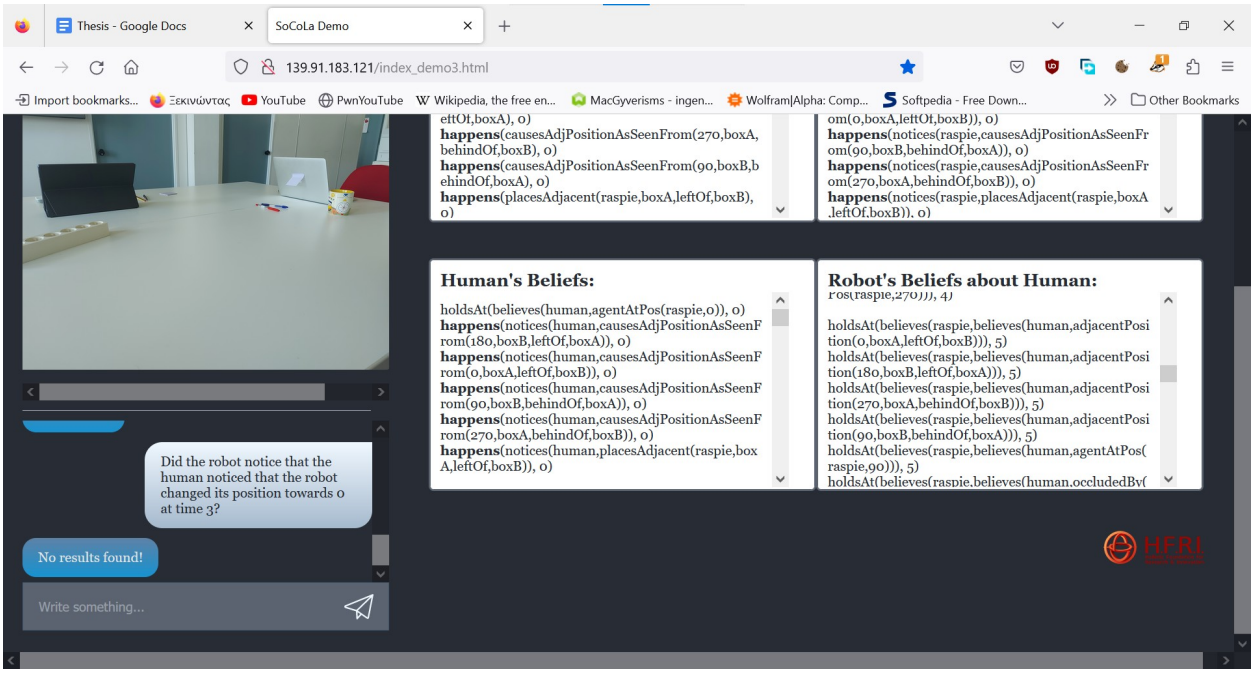
The robot was also at angle 180 at time point 2. Does it believe that the human believes that the robot was at 180 at time 2? Yes, it does.



Ditto for time point 5, where it was at angle 90…

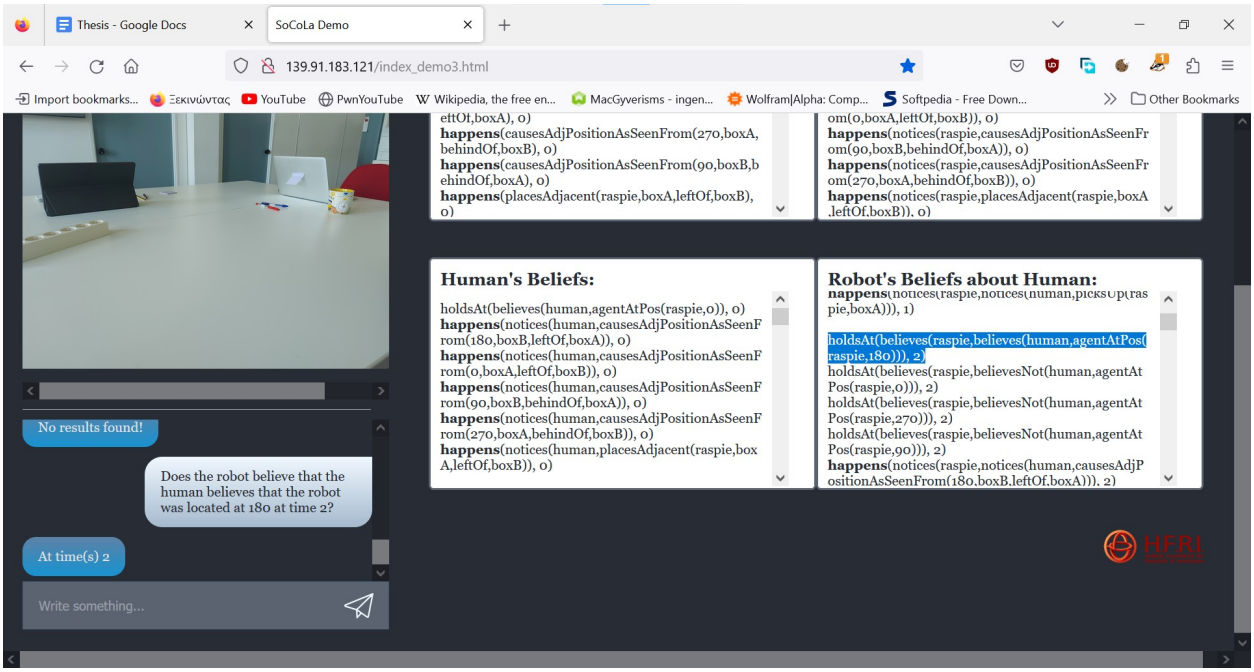


Now, the user may think that the robot changed its position towards angle 0 at time point 3. If this did happen, did the robot notice that the human noticed that change?



No. This means that the robot did not change its position towards 0 at time point 3. Had it done so, the human would have noticed it, and so the robot would have noticed it as well. But since

nobody noticed anything, nothing happened (like in the real world). And finally, we learned earlier that the robot was at angle 180 at time point 2. Was it really that way? The answer is yes.



So, we can now see how our system can handle level 2 epistemic questions as well, in addition to level 1 ones. The only limitation here is that the only subdomain that such questions are applicable is the one that concerns the robot's thoughts about the human. However, given the flexible nature of the GUI (and the system's implementation in general), it's possible that other combinations may be included in future revisions of the project as well. The matter, however, is that our system can handle both epistemic question levels easily, so, let's sum up everything in a final combined and dynamic usage scenario…

Section 6.4. Case 4: A small, dynamic scenario

In this scenario, we are going to make some observations about the world and the agents. Then, we will run the reasoner with a new event entry. After this, we will repeat the observations, after this new event has happened. What we want to do here is demonstrate how the reasoner works in an actual usage scenario, that is, how we can trust it in order to answer different kinds of user questions.

We initially have maxstep=10.

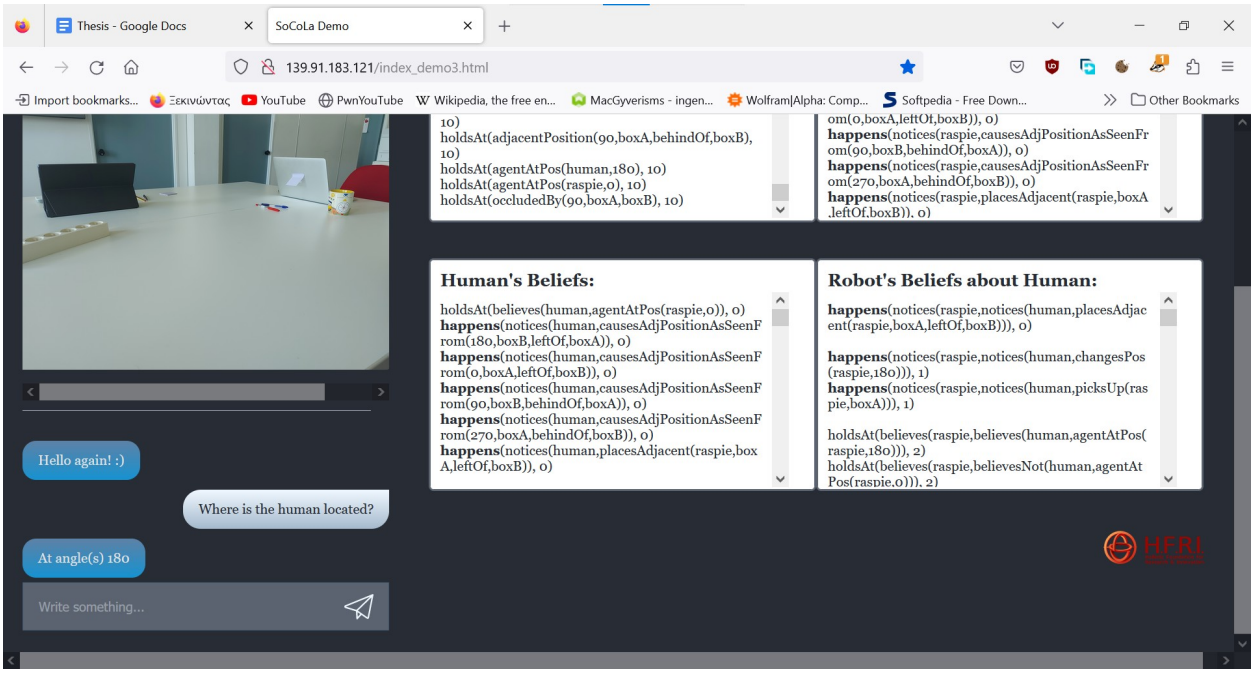
-Where is the human located?
-Is boxA occluded to the human as seen from 180?

-Where is the robot located?
-Does the robot believe that the human believes that the robot was located at 180 at time 2?
-Is boxA occluded to the robot as seen from 0?
-Did the human place boxA behind of boxB?
-Does the robot believe that boxB is behind of boxA as seen from 90?
-What if the human moves boxA behind of boxB?

Now, we add an event where the human changes her position towards 0 at time 5 and run the reasoner with maxstep=15. Then, we ask our questions again.

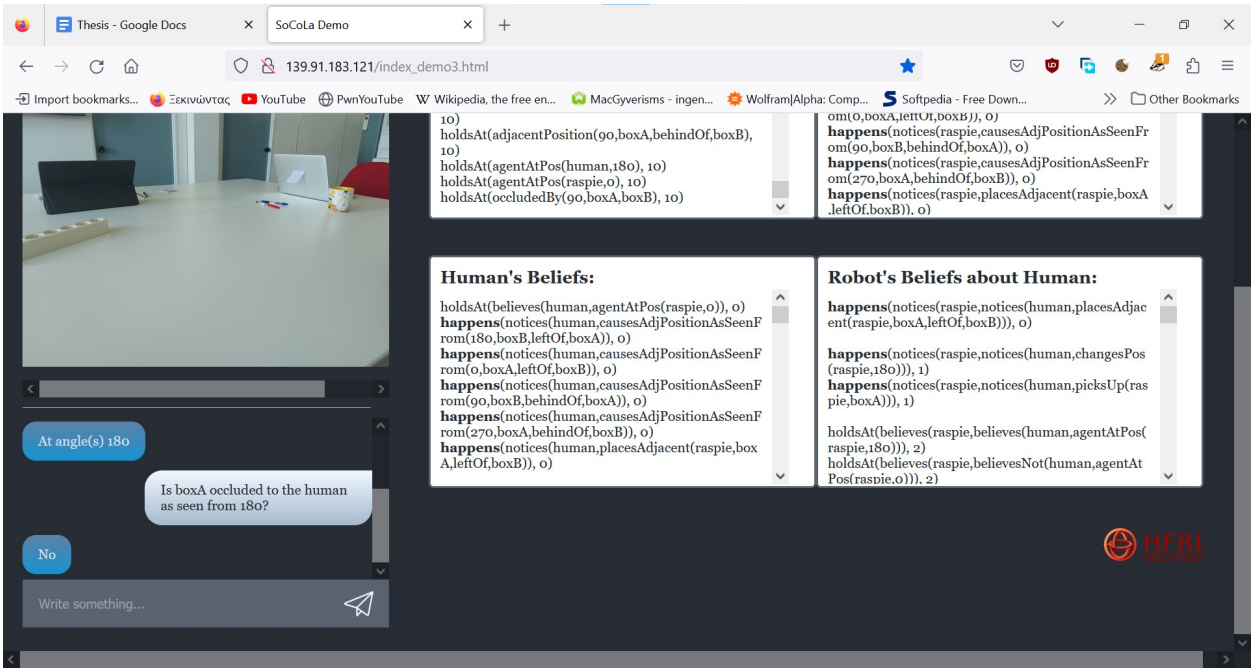
-Where is the human located?
-Is boxA occluded to the human as seen from 180?
-Where is the robot located?
-Is boxA occluded to the robot as seen from 0?
-Did the robot pick up boxA at time 1? //Mhpws na to balw nia pianei ola ta timepoints?
-Does the robot believe that boxB was behind of boxA as seen from 90 at time 4?
-Did the robot notice that the human changed her position at time 5?
-Does the robot believe that the human believes that the robot changed its position towards 180 at time 1?

Let's get this case running. We start at max timepoint=10 and we first ask a simple question: *where is the human located right now?*

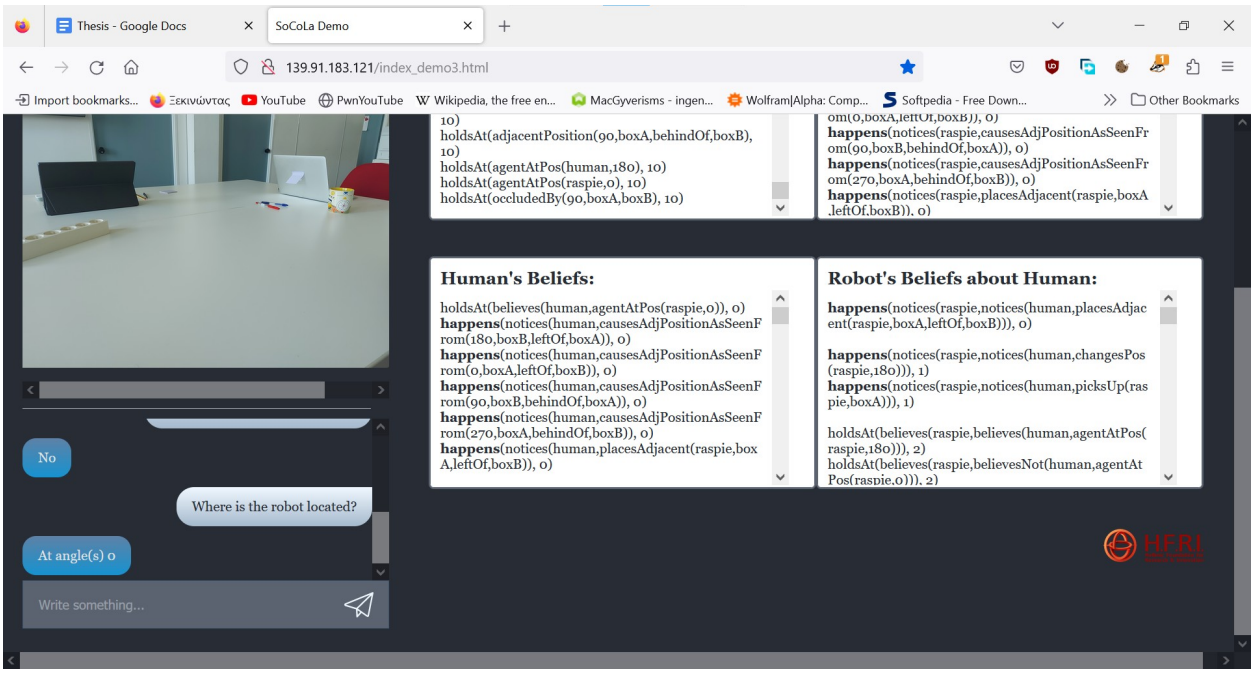


Now that we know the human's angle, let's ask another one: *is boxA occluded to the human?*

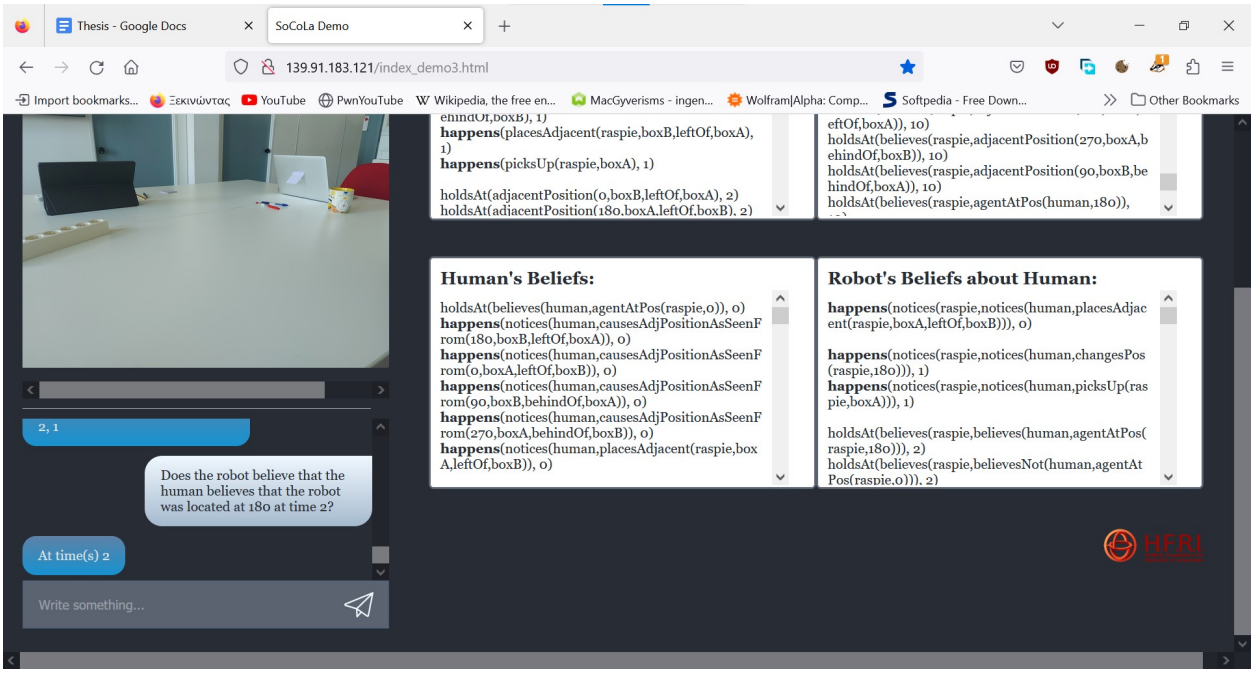
Given that at time point 10 there is no occludedBy(for angle 180 (the human's angle), the answer should be a "No".



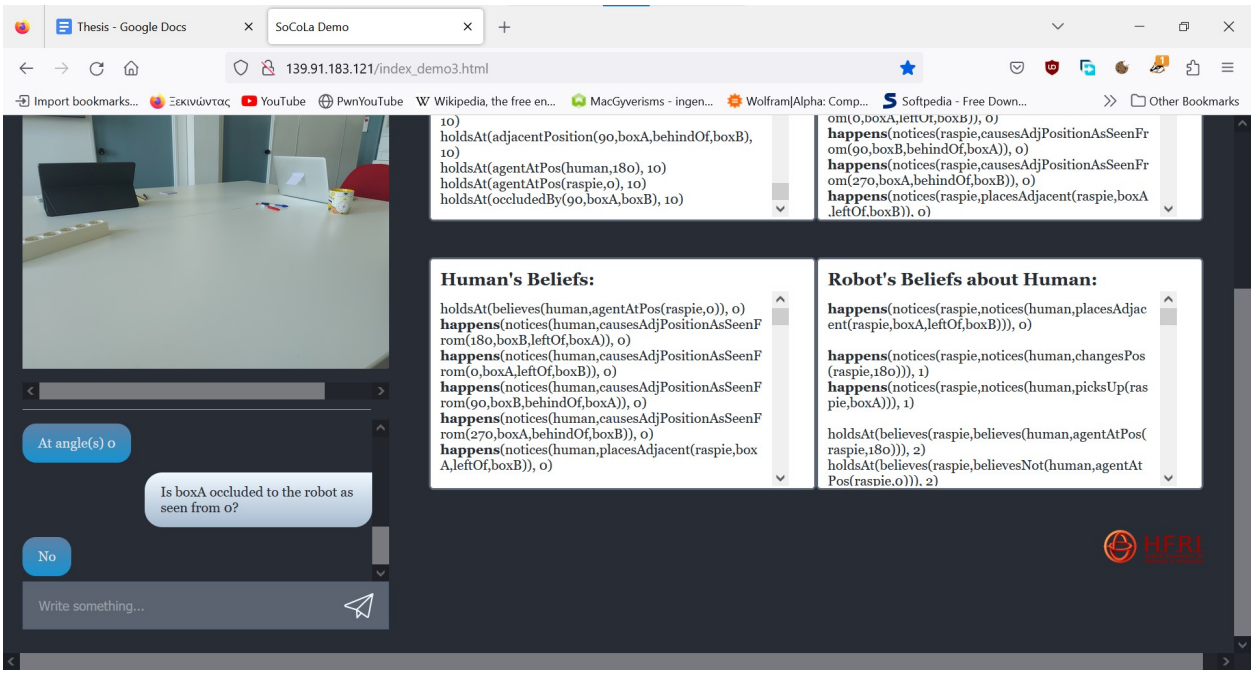
So far, so good. Now we need to learn the location of the robot...



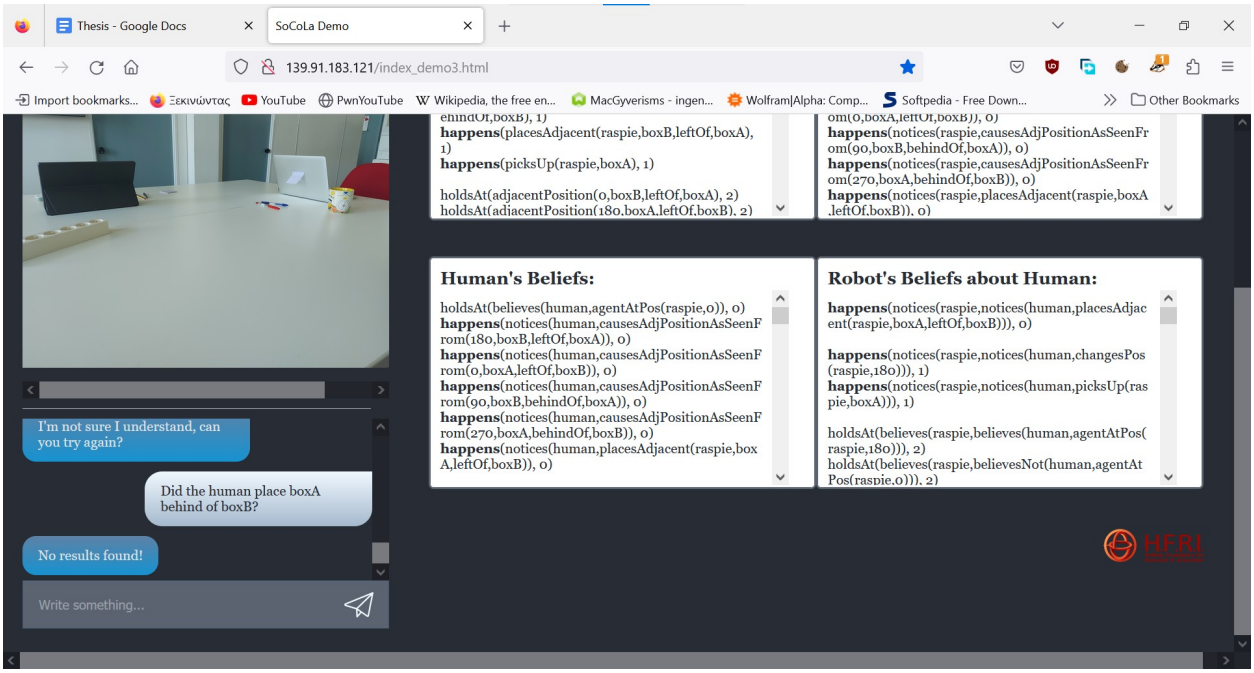
...and ask it whenever it believes that the human believes that the robot was located at angle 180 at time 2.



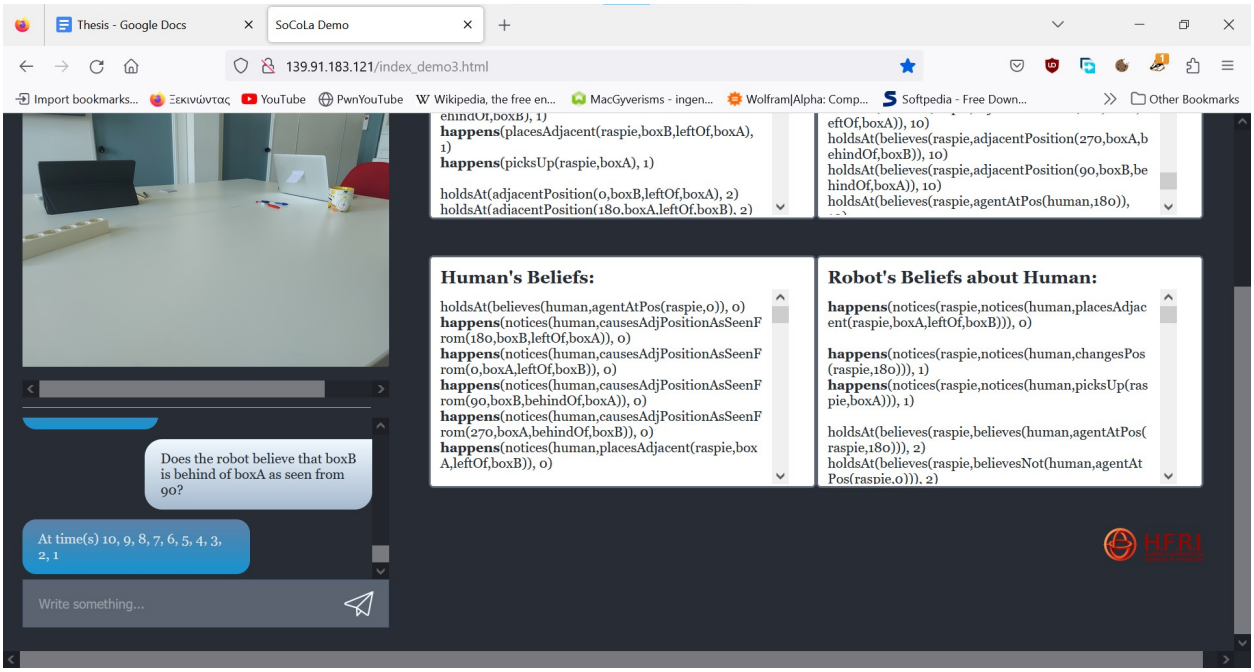
Again, a question about occlusions. Now we are asking whenever boxA is occluded to the robot as seen from angle 0. It should be not.



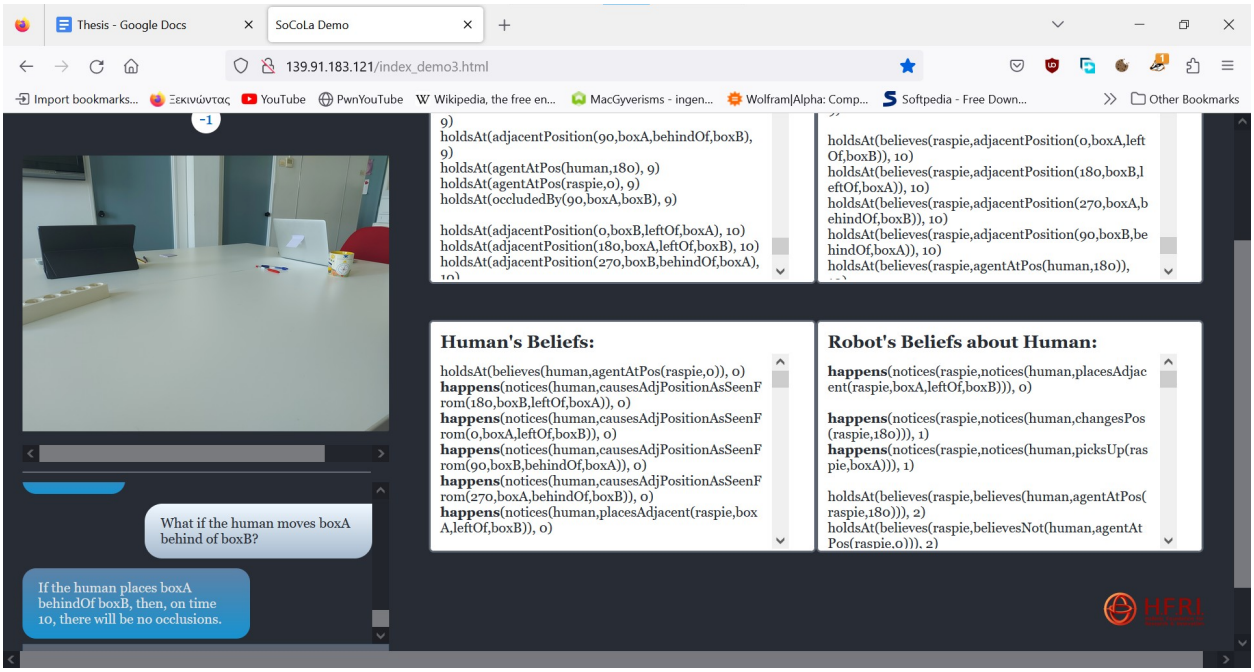
And yet another polar. *Did the human place boxA behind of boxB*. No predicates, answer is "No".



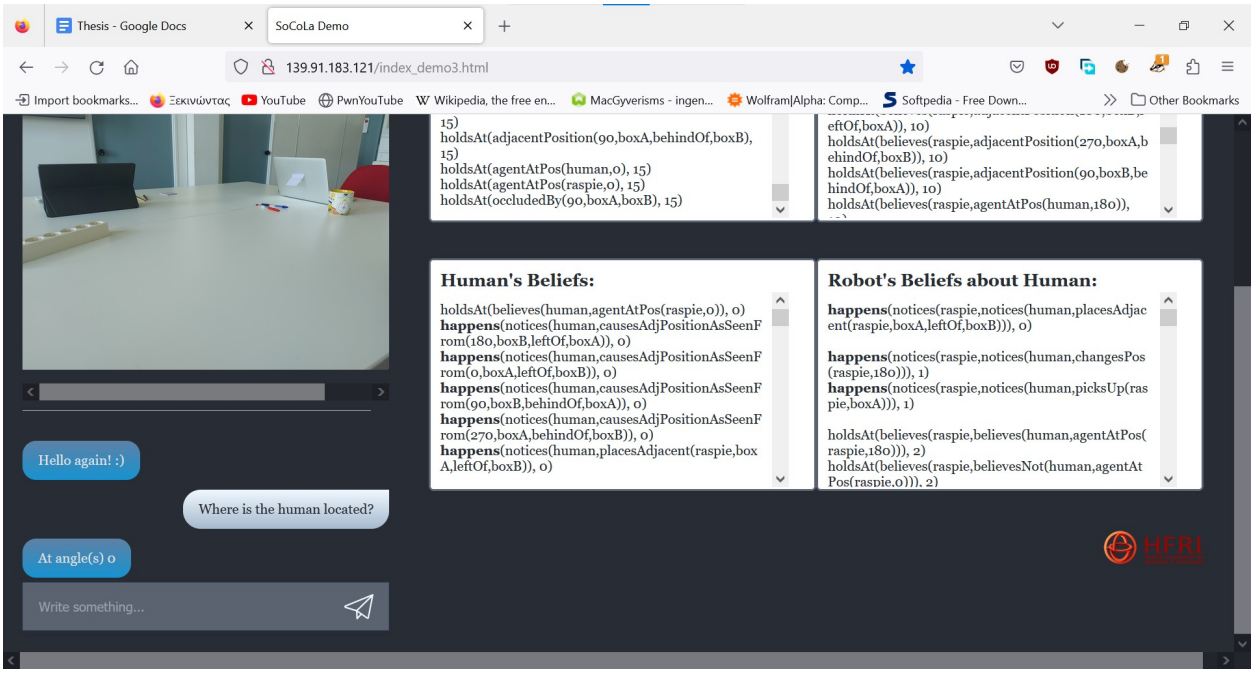
Yet another epistemic question on item positioning: *does the robot believe that boxB is behind of boxA as seen from 90 at the current moment?* Yes, it does.

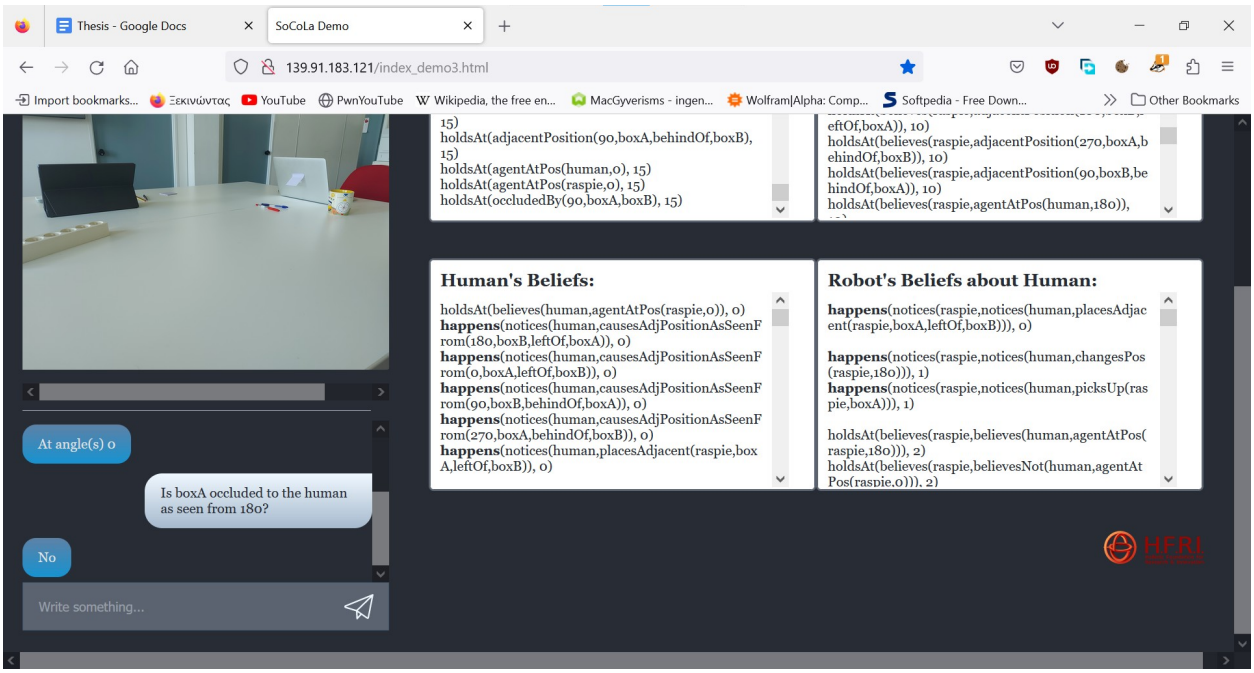


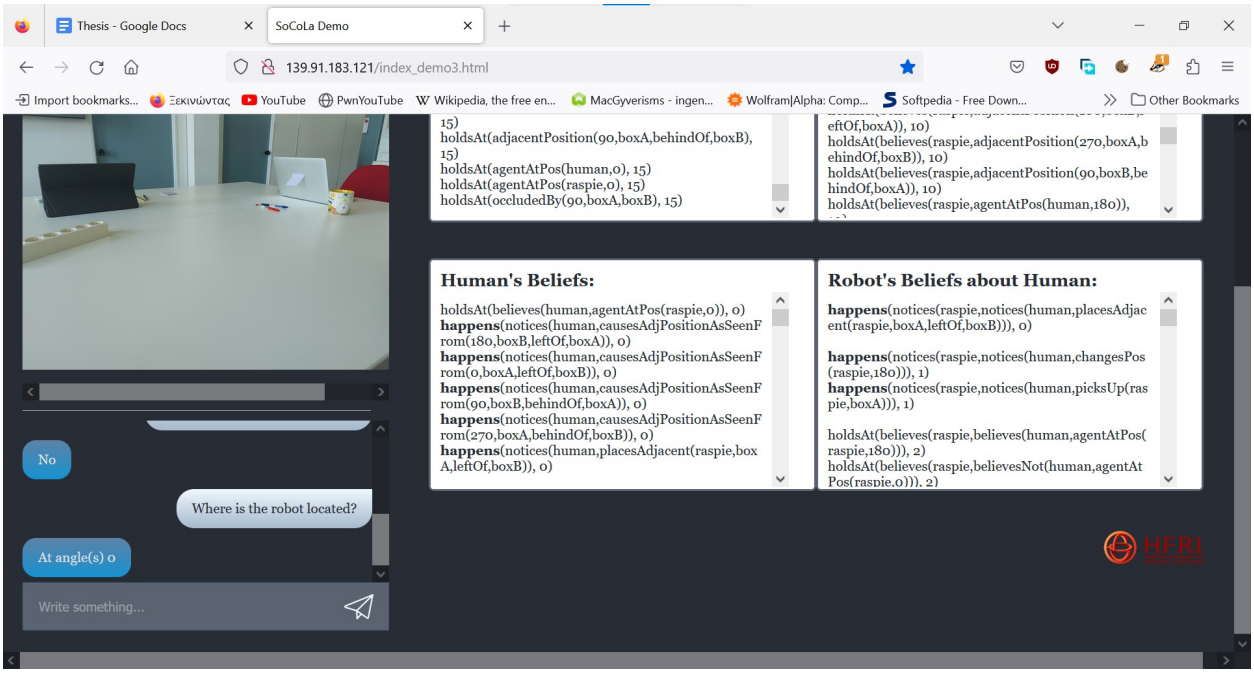
And a hypothesis: *What if the human moves boxA behind of boxB at the current time point?*

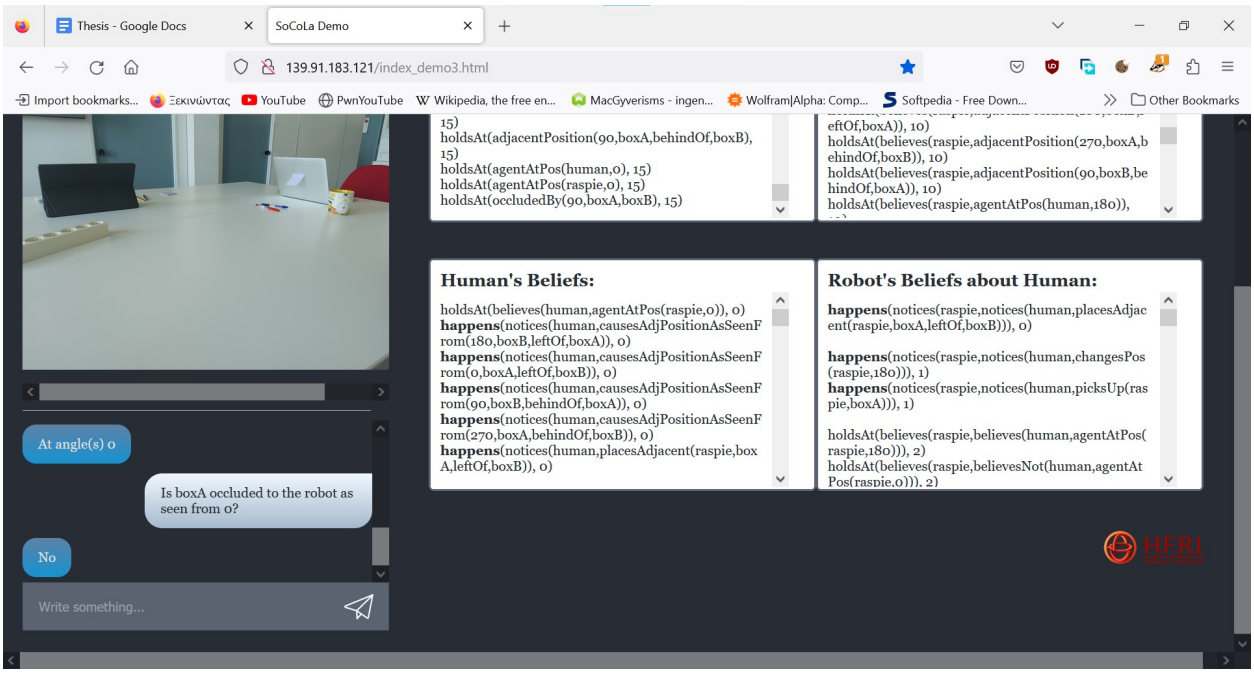


Now, we do as promised in the initial case description. We add a new predicate in the initial state and run the reasoner again so as to simulate a dynamic change in the world state. The predicate we add is: happens(changesPos(human,0),5), which means that the human changes her position towards angle 0 at time point 5. We run the reasoner again, this time with max timepoint = 15. We repeat some of the questions before:

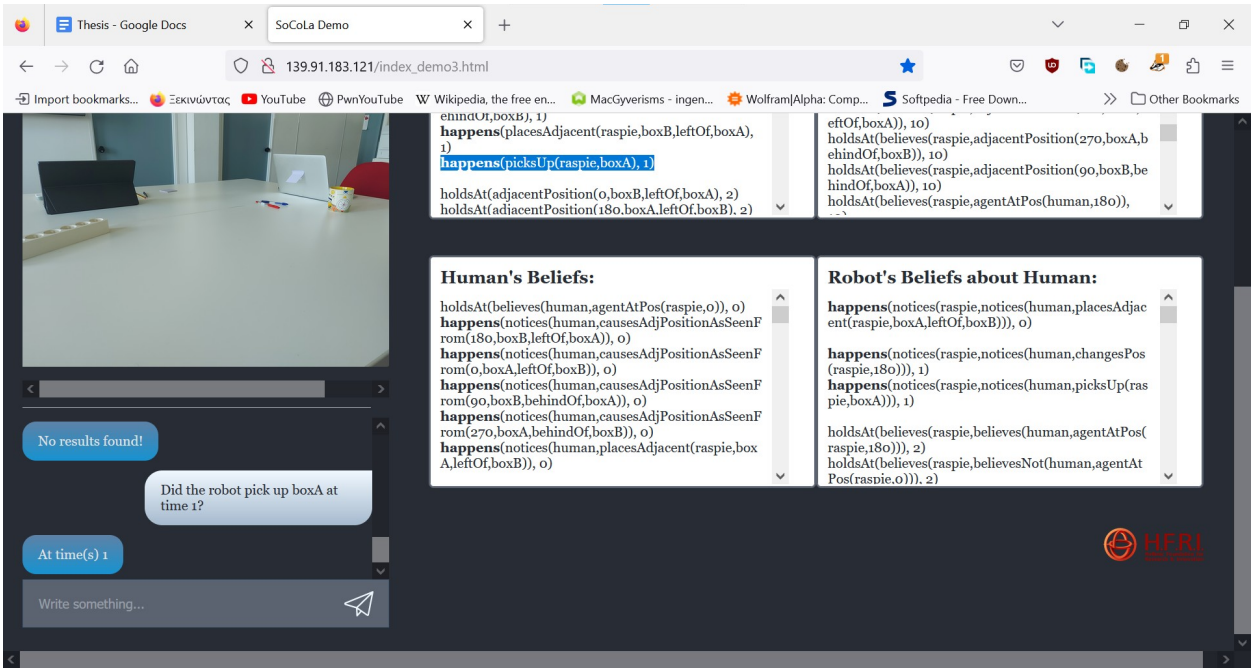



Thesis - Google Docs
SoCoLa Demo
139.91.183.121/index_demo3.html
Import bookmarks...
Ξεκινώντας
YouTube
PwnYouTube
Wikipedia, the free en...
MacGyverisms - ingen...
Wolfram|Alpha: Comp...
Softpedia - Free Down...
Other Bookmarks
15)
holdsAt(adjacentPosition(90,boxA,behindOf,boxB), 15)
holdsAt(agentAtPos(human,0), 15)
holdsAt(agentAtPos(raspie,0), 15)
holdsAt(occludedBy(90,boxA,boxB), 15)
eftOf,boxA)), 10)
holdsAt(believes(raspie,adjacentPosition(270,boxA,behindOf,boxB)), 10)
holdsAt(believes(raspie,adjacentPosition(90,boxB,behindOf,boxA)), 10)
holdsAt(believes(raspie,agentAtPos(human,180)),
Human's Beliefs:
holdsAt(believes(human,agentAtPos(raspie,0)), 0)
happens(notices(human,causesAdjPositionAsSeenFrom(180,boxB,leftOf,boxA)), 0)
happens(notices(human,causesAdjPositionAsSeenFrom(0,boxA,leftOf,boxB)), 0)
happens(notices(human,causesAdjPositionAsSeenFrom(90,boxB,behindOf,boxA)), 0)
happens(notices(human,causesAdjPositionAsSeenFrom(270,boxA,behindOf,boxB)), 0)
happens(notices(human,placesAdjacent(raspie,boxA,leftOf,boxB)), 0)
Robot's Beliefs about Human:
happens(notices(raspie,notices(human,placesAdjacent(raspie,boxA,leftOf,boxB))), 0)
happens(notices(raspie,notices(human,changesPos(raspie,180))), 1)
happens(notices(raspie,notices(human,picksUp(raspie,boxA))), 1)
holdsAt(believes(raspie,believes(human,agentAtPos(raspie,180))), 2)
holdsAt(believes(raspie,believesNot(human,agentAtPos(raspie,0))), 2)
HFRI
At angle(s) 0
Is boxA occluded to the human as seen from 180?
No
Write something...

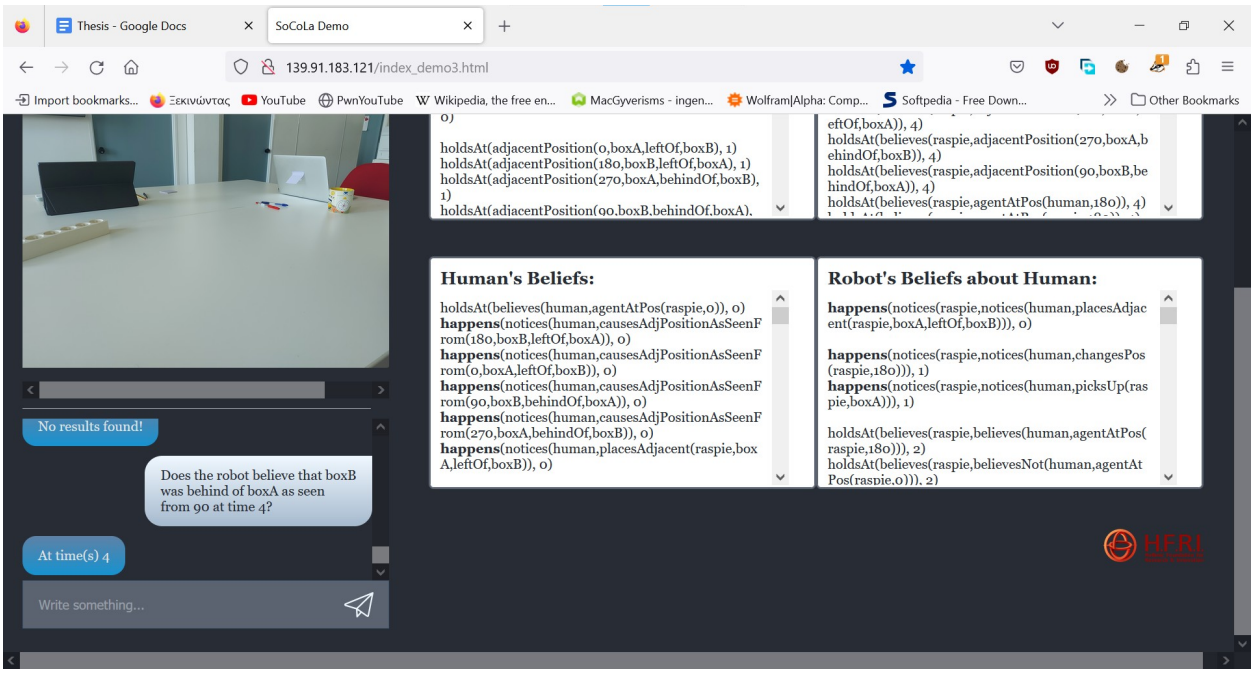

Thesis - Google Docs
SoCoLa Demo
139.91.183.121/index_demo3.html
Import bookmarks...
Ξεκινώντας
YouTube
PwnYouTube
Wikipedia, the free en...
MacGyverisms - ingen...
Wolfram|Alpha: Comp...
Softpedia - Free Down...
Other Bookmarks
15)
holdsAt(adjacentPosition(90,boxA,behindOf,boxB), 15)
holdsAt(agentAtPos(human,0), 15)
holdsAt(agentAtPos(raspie,0), 15)
holdsAt(occludedBy(90,boxA,boxB), 15)
eftOf,boxA)), 10)
holdsAt(believes(raspie,adjacentPosition(270,boxA,behindOf,boxB)), 10)
holdsAt(believes(raspie,adjacentPosition(90,boxB,behindOf,boxA)), 10)
holdsAt(believes(raspie,agentAtPos(human,180)),
Human's Beliefs:
holdsAt(believes(human,agentAtPos(raspie,0)), 0)
happens(notices(human,causesAdjPositionAsSeenFrom(180,boxB,leftOf,boxA)), 0)
happens(notices(human,causesAdjPositionAsSeenFrom(0,boxA,leftOf,boxB)), 0)
happens(notices(human,causesAdjPositionAsSeenFrom(90,boxB,behindOf,boxA)), 0)
happens(notices(human,causesAdjPositionAsSeenFrom(270,boxA,behindOf,boxB)), 0)
happens(notices(human,placesAdjacent(raspie,boxA,leftOf,boxB)), 0)
Robot's Beliefs about Human:
happens(notices(raspie,notices(human,placesAdjacent(raspie,boxA,leftOf,boxB))), 0)
happens(notices(raspie,notices(human,changesPos(raspie,180))), 1)
happens(notices(raspie,notices(human,picksUp(raspie,boxA))), 1)
holdsAt(believes(raspie,believes(human,agentAtPos(raspie,180))), 2)
holdsAt(believes(raspie,believesNot(human,agentAtPos(raspie,0))), 2)
HFRI
No
Where is the robot located?
At angle(s) 0
Write something...



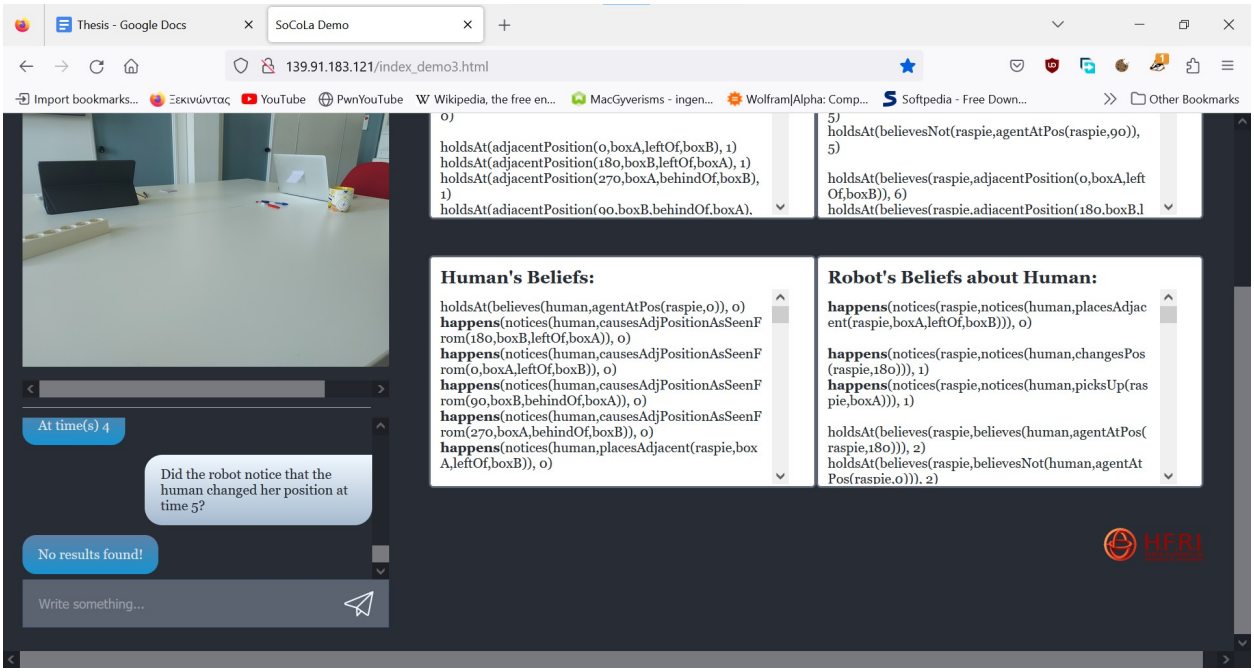
Now for something different: *did the robot pick up boxA at time 1?* It definitely did.



Yet another question on item positions...

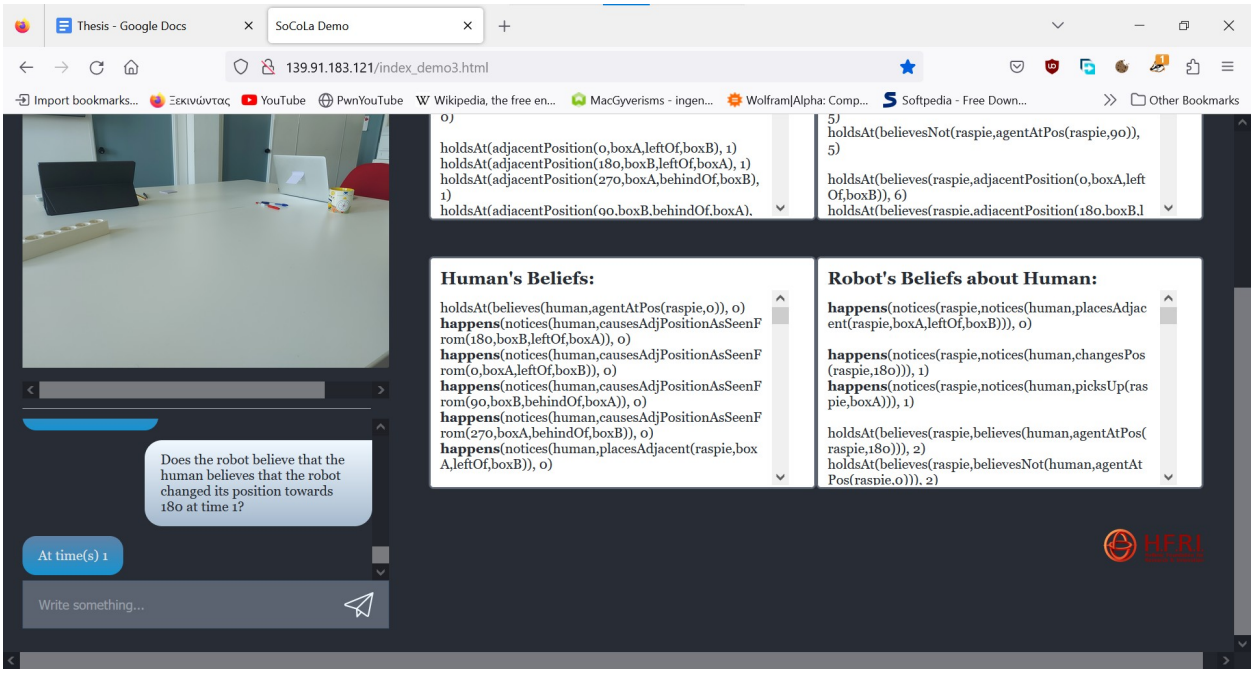


...and an epistemic question on whenever the robot notice that the human changed her position at time 5. Oddly enough, it returns a “no”. Given that the human did *indeed* change her position at time 5, as per our modification to the initial state, shouldn’t it say “Yes”, which means, should there be the appropriate “notices” predicate in the “Robot’s Beliefs” area?



It does however believe that the human believes that the robot changed its position towards 180

at time 1. “Notices” should be a more convincing keyword...



To sum up, we can now see how our system handles dynamic usage scenarios, where the state of the world and the agents’ minds changes over time. This, in fact, is closer to a real-world setting, where the state could change anytime. Sufficient to say, this demonstrates how efficiently can our architecture handle real-world usage cases that develop over time, without any hassle.

Chapter 7. Implementation

Section 7.1. Some words about the implementation

We have seen the architectural design of our system and how it works. Now, it's time to get deeper inside each component and examine how exactly it works. Our project is modular, which means that each component uses different technologies and procedures in order to accomplish its individual goal. To this end, we need to know how exactly each of these components accomplishes its functionality, so as to have a greater picture of the whole system as a pipeline – because this is what it essentially is. This examination also enables us to sugggest changes and tweaks in the components, so as to expand their functionality and solve bugs wherever they appear. Thus, examining the components is crucial, since it enables a developer to better understand the system in detail and modify it according to their needs.

Section 7.2. The controller

We shall start from the controller, which is the "heart" of the entire architecture. The controller has the following "duty": It receives processed data from wit.ai, forms Event Calculus queries based on them, performs reasoning if needed, records results and returns them to the user, if it finds any. The "heart" of the controller is a dispatcher function that calls appropriate handling functions based on what kind of user question it received from wit.ai. In pseudocode, it looks something like this:

```
function dispatcher(userData):
    if (userData.type = POLAR):
        handlePolar(userData)
    else if (userData.type = WHATIF):
        handleWhatif(userData)
    else if (userData.type = WHEN):
        handleWhen(userData)
    …
```

As we can see, the dispatcher calls different functions based on the type of the user question. Each of these functions handles the user question in a different manner. What all of them have in common is that they implement the same predicate functionality: they form an Event Calculus predicate based on the question's intent and entities, search for what matches with it (as closely as possible) among all EC predicates (either existing ones or reasoned), record everything that matches and then return the matched predicates as results to the user. The controller is fully implemented in Python, and thus, can run in any system that can support the Python language. In our experimental setup, it ran on a Windows 10 system running the Anaconda Python environment. The code for the controller can be found in https://gitlab.isl.ics.forth.gr/socola/p-

chat (registration with ISL ICS-FORTH GitLab required)

Section 7.3. The chatbot: Wit.ai[LN1]

The second most important component of our architecture is the chatbot itself. The chatbot takes the user's Natural Language query and marks its intent, as well as any entities found within it. Essentially, it provides a more "objectized" version of the user's query to the code that calls it. In our implementation, we use Wit.ai as our chatbot. Wit.ai, an open source chatbot project, was developed by Meta (formerly Facebook) as a platform for user-assisted Natural Language recognition. Apps developed using this platform utilize supervised machine learning in order to "learn" new intents, entities and utterances that suit each app's purpose. It supports both text and speech recognition, opening the door for a vast variety of chatbot applications and implementations.

Apart from its rather intuitive learning interface, Wit.ai also supplies APIs to app developers in order to call its functionality from within their code. While it has support for the traditional HTTP API (called through standard HTTP requests), it also provides client APIs for Node.js, Ruby and Python, a client API for iOS apps and many more APIs for a variety of languages and clients. Apart from sending NL questions and receiving processed answers (mostly in the form of JSON objects), the APIs also allow one to import or export app data from Wit.ai, enabling easier backup and restore procedures in the event something goes south.

Those degrees of flexibility and adaptivity were the main reasons we chose wit.ai for our system. At first, we considered other similar-function platforms, such as Deeppavlov[LN2] and RASA[LN3], however, it was wit.ai's intuitiveness that won us over in the end. It may not be as advanced as other platforms, but its ease of use and expressiveness can open doors to very powerful chatbot apps, for any kind of purpose, and whenever input is text or voice.

Section 7.4. The Graphical User Interface[LN4]

Now let's move on to the GUI, which was implemented earlier than the controller. It's built with Node.js, with the addition of Express, a Node.js framework that assists in quick server creation. Under the hood, it utilizes the Websockets API, which implements real-time functionality by allowing the unilateral exchange of small data chunks over a persistent connection. The Websockets API is built-in into browsers, but the server requires a backend app in order to utilize it for communications.

The GUI also utilizes a controller, which assists in communication between the GUI and every other component in the architecture. The controller is a simple message routing server, that receives JSON messages from all components,forwards them to the appropriate channel, and responds with ack messages. It is implemented in Python3. It communicates with all components with the use of ZeroMQ2[LN5], a high performance universal mes-saging library that

supports common messaging patterns like Publisher/Subscriber(PUB/SUB), Re-quest/Reply(REQ/REP), Pipeline and others, over a variety of transports and keeps the code modular and easy to scale. The Controller communicates with each component over a distinct connection, using the TCP message transport, and a dedicated message pattern for each component that best suits the conversation's purpose. All of the messages between the components are in the form of JSON messages.

Section 7.5. Clingo - the system's reasoner[5]

Clingo is an ASP processor that, as described in above sections, merges the grounding and solving processes into one. One of its strong points is that it offers APIs for every common programming language that utilizes ASP, those being C, C++, Lua and Python (the latter of which is the main language that is used throughout our project). All APIs offer the same functionality, and offer control over the three principal processes clingo performs: parsing, grounding and solving.

Chapter 8. Conclusions

In this paper, we saw an intuitive chatbot architecture proposal capable of modular expansion, with enhanced abilities such as the ability to ask epistemic questions, as well as reason new situations using data of already known ones. We saw in detail what our system is made out of and how every component in the architecture connects with each other and interacts with them, as well as how exactly each component is implemented. That is, what technologies are used in its implementation, how exactly it works on an algorithmic level, as well as specifications for a real-world application of our system. While the results of testing our work are rather informative and satisfactory, they only concern an in-depth application of our system. A third-party that may use our architecture will likely implement it in another context, which means that while it will work for their purpose, the results may be a little bit different than the ones showcased in this paper.

As we have stated many times before, our architecture's modular architecture makes it easy to test or modify individual components, as well as add new ones or remove existing ones. In this manner, we can discuss many ways in which our architecture may be improved in the future. An example would be a filter between the controller and wit.ai, which removes non-question-specific words from the user's input, making it easier for wit.ai to properly recognize entities and intents.

Another future improvement may be the introduction of new reasoner rule sets in order to provide more accurate reasoning when needed. It would also be a good idea to split the controller into multiple modules (programs), controlled by a central dispatcher program, for the sake of simplicity (e.g one module will be handling non-epistemic questions, another will be handling epistemic questions, another will be handling the negotiations with wit.ai, and so on). Also, while wit.ai is pretty much versatile, it may not be the best-of-the-best choices for natural language processing, which means that we may either replace it with something else, or add another NLP platform in the pipeline so that we may get more precise and better NL recognition. We may also choose to extend Event Calculus by adding new event and fluent types, such as whenever an agent is unsure of a situation or whenever it accepts a change in the environment or not.

The list of improvements can, in fact, go on forever, given our proposal's endless possibilities. The bottom line, however, is that we have proposed a chatbot architecture that is inherently powerful, expansible, and easy to maintain and modify. While these traits may not make it the-best-out-there, they definitely add to it a huge degree of flexibility, which is definitely a plus on such chatbot architectures. This flexibility enables it to be improved and upgraded easily, allowing for a vast choice of improvements and customizations - possibly more than any other chatbot architecture can allow.

Chapter 9. References

[1] Das, Abhishek, et al. "Visual dialog." *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017.

[2] Kowalski, R., Sergot, M. A logic-based calculus of events. *New Gener Comput* **4**, 67–95 (1986). https://doi.org/10.1007/BF03037383

[3] Gouidis F., Vassiliades A., Basina N. and Patkos T. (2022). Towards a Formal Framework for Social Robots with Theory of Mind. In *Proceedings of the 14th International Conference on Agents and Artificial Intelligence - Volume 3: ICAART,* ISBN 978-989-758-547-0, pages 689-696. DOI: 10.5220/0010893300003116

[4] Vladimir, Lifschitz. "What is answer set programming." *AAAI*. Vol. 8. 2008.

[5] Gebser, Martin, et al. "Multi-shot ASP solving with clingo." *Theory and Practice of Logic Programming* 19.1 (2019): 27-82.

[6] Tamosiunaite M, Aein MJ, Braun JM, et al. Cut & recombine: reuse of robot action components based on simple language instructions. The International Journal of Robotics Research. 2019;38(10-11):1179-1207. doi:10.1177/0278364919865594

[7] Hatori, Jun, et al. "Interactively picking real-world objects with unconstrained spoken language instructions." *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2018.

[8] Zhang, Shiqi, and Peter Stone. "CORPP: Commonsense reasoning and probabilistic planning, as applied to dialog with a mobile robot." *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 29. No. 1. 2015.

[9] He, Xuehai, et al. "Pathvqa: 30000+ questions for medical visual question answering." *arXiv preprint arXiv:2003.10286* (2020).

Chapter 10. Online Resources

[LN1] https://wit.ai/

[LN2] https://deeppavlov.ai/

[LN3] https://rasa.com

[LN4] https://www.dropbox.com/s/6hklgnc11hot0er/D7.4_Demos.pdf?dl=0

[LN5] https://zeromq.org/

[LN6] https://en.wikipedia.org/wiki/Turing_test