A High Level Language for Human Robot Interaction

by

Barry Thomas Lumpkin

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved July 2012 by the
Graduate Supervisory Committee:

Chitta Baral, Chair
Joohyung Lee
Georgios Fainekos

ARIZONA STATE UNIVERSITY

August 2012

ABSTRACT

While developing autonomous intelligent robots has been the goal of many research programs, a more practical application involving intelligent robots is the formation of teams consisting of both humans and robots. An example of such an application is search and rescue operations where robots commanded by humans are sent to environments too dangerous for humans. For such human-robot interaction, natural language is considered a good communication medium as it allows humans with less training about the robot's internal language to be able to command and interact with the robot. However, any natural language communication from the human needs to be translated to a formal language that the robot can understand. Similarly, before the robot can communicate (in natural language) with the human, it needs to formulate its communique in some formal language which then gets translated into natural language. In this paper, I develop a high level language for communication between humans and robots and demonstrate various aspects through a robotics simulation. These language constructs borrow some ideas from action execution languages and are grounded with respect to simulated human-robot interaction transcripts.

ACKNOWLEDGEMENTS

Without my friends, colleagues, and especially mentors, this work would not have been possible.

I would like to thank Dr. Chitta Baral for the many years of guidance he has given me. It was through him that I even began learning about research. I especially thank him for his collaboration with me on this work.

I would also like to thank Dr. Joohyung Lee and Dr. Georgios Fainekos for serving as my committee members.

I thank my colleagues within the BioAI lab who have been so much to me: mentors, co-workers, and most of all friends.

I cannot begin to express my thanks for my family. They have always been there for me, and I am so lucky to have them.

Finally, and closest to my heart, I thank Jenny Hastings. She has been my driving force when I felt too worn out to keep pushing forward and she has been my source of joy on these many long days.

Thank you all.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION AND MOTIVATION

Within the field of human-robot teamwork, there are highly varied implementations. On one end of the spectrum, the use of teleoperation allows robots to be used as tools in which a human operator has direct control over a robot's actions. Such systems are highly dependent on the operator's skill and are extremely hindered by situations with limited bandwidth. The other extreme contains highly autonomous robots that are simply given a high-level goal from a human supervisor who does not directly interfere in the robot's operation. This gives the operator the ability to handle many systems simultaneously but does not provide any flexibility in the event of unexpected events.

A more practical application would be an intelligent robot team in which humans and robots work together much in the same way a team of humans would. Each individual, human or robot, would be able to actively seek assistance from others when needed. For example, in an urban search and rescue scenario, it may be that the environment is too dangerous for humans, so a group of robots would be sent instead. These robots would be given tasks to autonomously complete, but since the environment is most likely unpredictable and possibly still changing, the robots would need to seek guidance throughout the operation. Another example is in face-to-face teamwork such as a construction project in which humans and robots are working together to raise the framework of a building. It is highly unlikely that each member of the team would be able to complete their work alone, rather they would need to ask one another for help affixing a new segment or seek guidance on which area should be completed next.

For such human-robot interaction, natural language is considered to be a good communication medium as it allows humans with little training on the robot's internal language to be able to command and interact with the robot. Apart from requiring less training, utilizing natural language would allow for a faster dialogue as the human would not need to translate their thoughts into a structured format that the robot would understand. However, robots still require commands to be given in a structured format in

order to be processed. As such, the natural language communication must be translated into a formal language which the robot can understand. Additionally, when the robot is forming a communique back to the human, it must first formulate the message in this formal language which is then converted into natural language the human can easily understand.

1.1 Related Work

Current work in the field of teamwork based human-robot interaction includes:

The Human-Robot Interaction Operating System (HRI/OS) from the Peer-to-Peer Human-Robot Interaction project provides a focus on allowing agents to submit requests for help which will be processed once the resources necessary become available, such as other agents (Scholtz 2002; Fong et al. 2005; 2006). Another key aspect of the HRI/OS software is that it is designed to utilize spatial reasoning and perspective-taking to enable dialogue using relative locations.

The Jidowanki and Biron robots utilize a task negotiation dialogue in which the robots will prompt the user with queries until a clear goal is assigned based on the current known environment (Clodic et al. 2007). Additionally, this system allows for a robot to submit a request for a plan modification in the event that it determines another, potentially better, plan is now available due to changes in the environment. The user is able to accept or reject this new plan, or even initiate their own plan modification.

A robotic wheelchair was recently used to study the effect of interactive dialogue on how a user interacts with the system (Fischer 2011). The first study showed that an interactive dialogue allowed the user to better understand the capabilities of the system and become much more proficient in its use. The second study showed that slight changes in the wording used by the robot had a significant effect on users' engagement in human-robot interactions.

The Generalized Grounding Graphs is a framework to find and execute plans generated from natural language commands (Tellex et al. 2011). This work involves automatically generating a probabilistic graphical model from natural language commands. Using an annotated corpus of natural language commands paired with their correspond-

ing actions, the system automatically learns the meanings of manipulation verbs. This is a limitation of the system since the supervised learning requires a significant annotated training set. Additionally, the system is limited by the size of their search space since increasingly complex tasks require deeper searches and more sophisticated algorithms. An interesting note is that this framework returns not only the plan for execution, but also groundings for everything in the command with corresponding confidence scores. Such groundings can be used to identify portions of the command that need further clarification.

A system for learning navigational commands is presented by David Chen and Raymond Mooney (Chen and Mooney 2011). This system learns navigation instructions by observing communication and corresponding actions without needing either linguistic knowledge or direct supervision. The lack of supervision is a key aspect of this system as it means the expense of generating an annotated corpus is avoided. The system was also designed to use landmarks in the environment to assist with error recovery on long and complex instructions. Due to the length of the commands, without landmarks, a small error at the beginning could ruin the entire process. Additionally, their system refines the generated plans by using a learned semantic lexicon to remove extraneous information to provide better plan generation.

There has also been similar work in describing temporal logic for motion planning (Fainekos et al. 2009, Kress-Gazit et al. 2009). These systems enable a robot to react to the environment and not simply follow a predetermined behavior. They utilize temporal logic based motion planning through the representation of a variety of tasks using Linear Temporal Logic (Pnueli 1977). This approach has a downside in that the LTL formulas, especially the LTL variant used by this system, are not always intuitive and may require some experience for writing.

The distributed, integrated, affective, reflective, cognitive (DIARC) architecture (Scheutz et al. 2007; 2011) includes natural language processing capabilities combined with backchannel feedback, such as nodding. This system contains algorithms for processing natural language instructions into logical forms that allow the system to

compute a course of actions to complete the goals. DIARC uses the Agent Development Environment to provide the infrastructure support for a variety of hosts and services. A note of interest is that the DIARC architecture also utilizes affect to modify its natural language interactions with other agents (Scheutz et al. 2005; 2006).

1.2 Motivation
The CReST Corpus

This work is based on data from the multi-modal CReST corpus which consists of human-human dialogues in an "instruction-following" task (Eberhard et al. 2010). This corpus is unique in that it involves remote collaborations between two interlocutors who each have to perform tasks that require the other's assistance. In addition, one interlocutor's tasks require physical movement through an indoor environment as well as interactions with physical objects within the environment. Thus, the dialogue is particularly useful for developing human-robot interaction systems where robots have to perform physical tasks based on instructions given by a human supervisor (e.g., search and rescue missions in disaster areas). The following are sample dialogue excerpts taken from the corpus:

Director: um you should i- [pause] straight in front of you should be: a: door

Director: go through the door

Searcher: okay

...

Director: um there will be a room on your right don't go in that room

Director: continue all the way to the end [pause] turn right

Searcher: okay

...

Director: okay [pause] and then you should [pause] there should have been two blue boxes one on the chair one behind the [pause] door

Searcher: um there's just one behind the door

...

Director: straight in front of you there should be a chair

Searcher: yes

Director: at a table there's a blue box there

Searcher: yes

Director: okay [pause] get that

Director: and then [pause] um keep - if you keep going straight from that chair there should be a pink box on a table or something

Searcher: oh yeah okay

Director: okay

Searcher: there's a pink box and a green box number five on that shelf

Director: okay

...

Director: okay um and we're still looking for green [pause] boxes number six and one

Searcher: okay

Searcher: so should I head back to the other room then?

Director: um sure head back to the other rooms

Searcher: okay

Director: maybe look around see: [pause] if - if you can find any green - green boxes

Motivating Example

The following dialogue is a hypothetical conversation between a human director and a robotic searcher to simulate the types of communication that would be expected within human-robot interactions. This example was created based on simplified versions of dialogues found in the CReST corpus. For this example, the director can only see a map and communicate verbally with the searcher which is physically in the mapped environment.

In this scenario, the robot, carrying a yellow block, is situated in a long east to west hallway with a room to the north as seen in Figure 1.1. Inside the room is a green box with the number 7 on the side. On the north end of the room is another hallway stretching east to west with a pink box on either end. Within this environment, the following is a possible conversation between the human director and the robot in

order to find the green box and its number, then place a yellow block within one of the pink boxes.

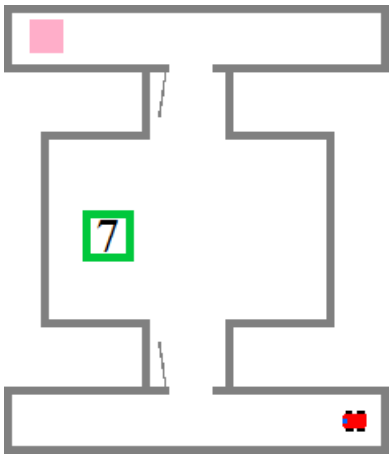

Figure 1.1: Motivating Example Environment

Director: While going down the hallway, find the door on your right and go through it.

Searcher: Done.

Director: Is there a green box in the room?

Searcher: Yes.

Director: Go to the box.

Searcher: Done.

Director: What is the number on the box?

Searcher: 7.

Director: Report the location of this box.

Searcher: Go down hallway. Enter first door on the right. Box is ahead on the left.

Director: Go through the door.

Searcher: North door or south door?

Director: North.

Searcher: Done.

Director: Either go left or go right.

Searcher: Choosing left. done.

Director: Go to the end of the hall. You should see a pink box.

Searcher: Done. There is a pink box.

Director: Place a yellow block in the pink box.

Searcher: Done.

1.3 Approach

The goal is to develop methodologies for natural language communication between a robot and its human controller in the context of the human controller directing the robot to perform certain tasks. This communication involves two parts:

1. *The human communicating with the robot and the robot receiving that input and processing it:* The human to robot communication is mainly of four types. (a) The human may direct the robot to do a certain task. (b) The human may provide knowledge for the robot to learn in the form of facts or new actions. (c) The human may ask a question to the robot. (d) The human may respond to a query by the robot.

2. *The robot communicating with the human:* The robot to human communication involves the following types. (a) It answers the human's questions, often involving what it senses. (b) It reports what it has done and could not do. (c) It asks a question to the human regarding what it should do. This could be about stating multiple actions that it could do and asking which one the human would prefer. This could be about remembering some previously assigned goal and sensing an opportunity to achieve that goal, even though the human director has moved on with respect to the goals. This could be about making sure that the plan it has made is acceptable to the human. In that case, the robot can state what it is planning to do and ask for confirmation from the human.

To be able to achieve the above communication, several formal languages need to be developed and linked with natural language. In particular, a Robot Input Language (RIL) and a Robot Output Language (ROL) are needed along with the ability to translate

statements uttered by humans in a natural language (say English) to the RIL and then translate statements in ROL to a natural language to be communicated to the human.

1.4 Main Contributions

In this research, I introduce two main types of robot interface languages. The Robot Input Language provides a means for the human to provide input to the robot. This language is divided into four distinct sections with each defined in terms relevant to their usage such as an extension of Golog with temporal logic or a database query language. These four sections cover the essential dialogue types of directives, knowledge for the robot, queries over the robot's knowledge, and answers to questions posed by the robot. The Robot Output Language allows the robot to communicate back with the human director. It is divided into three sections defined using a combination of simple statements, predefined predicates, and the database query language used within RIL. These three sections are reports to respond to prior directives, answers to respond to queries, and questions to resolve problems or choices encountered by the robot.

The main contribution of this work is that it provides the languages to form a bridge between natural language and robot architectures. This bridge will greatly simplify the process of converting natural language dialogue to a wide variety of platforms since language translators will only need to be designed to work with this middle language. Rather than training a new language translator for each type of robot, the robots will only need to implement a translation system for RIL and ROL.

To demonstrate the use of the RIL and ROL languages, I create a simulation of an urban search and rescue environment using Answer Set Programming logic to handle significant portions of the RIL-D language and the java based Agent Development Environment to handle the robot simulation and necessary ROL features. Finally, I describe a separate work that demonstrates a potential approach to translating natural language into RIL syntax.

1.5 Thesis Outline

The rest of this work is organized as follows: chapter 2 presents the syntax and semantics of the languages developed to serve as the interface between natural language and robot architectures. Chapter 3 describes the partial implementation of the languages using Answer Set Programming and the ADE robotics simulator as well as a method of translating natural language into RIL. Finally Chapter 4 finishes this work with a conclusion and discussion of potential future work.

Chapter 2

The Robot Interface Languages

As previously mentioned, RIL has four parts: Directives, Learning, Queries, and Answers. I propose an extension of Golog (Levesque et al. 1997) with temporal logic and "goal" statements as the language for Directives (RIL-D) and a database query language as the language for Queries (RIL-Q). The language for Learning (RIL-L) is composed of a logical syntax for learning about the world as well as language constructs for learning actions and goals for other agents. The syntax for the RIL Answer to a question (RIL-A) is determined by the specific question asked. ROL has three parts: Reports, Answers, Questions. The Reports language (ROL-R) consists of simple statements to respond to an RIL-D command. The Answers section (ROL-A) also contains simple statements of fact as well as the requested values from RIL-Q. Finally, the language for Questions (ROL-Q) consists of various predicates to enable questions on action plans and goal statements as well as the database query language as in RIL-Q. The following sections elaborate on each of these languages beginning with a background information.

2.1 Background

Propositional Formula

For the purpose of this work, a formula is defined syntactically as follows where propositional variables can either hold the value $true$ or $false$

$$\begin{aligned} < prop\_formula > \ ::= \ & < prop\_variable > \ | \ \neg < prop\_formula > \ | \\ & (< prop\_formula > \wedge < prop\_formula >) \ | \\ & (< prop\_formula > \vee < prop\_formula >) \end{aligned}$$

A formula is satisfied by an interpretation if it makes the formula true according to Table 2.1

Linear Temporal Logic

For the purpose of this work, a temporal formula is defined using Linear Temporal Logic (Pnueli 1977) as follows

$F1$	$F2$	$\neg F1$	$(F1 \wedge F2)$	$(F1 \vee F2)$
T	T	F	T	T
T	F	F	F	T
F	T	T	F	T
F	F	T	F	F

Table 2.1: Formula Interpretations

$$
\begin{aligned}
< LTL\_formula > \ ::= \ & < prop\_formula > \mid \neg < LTL\_formula > \mid \\
& < LTL\_formula > \wedge < LTL\_formula > \mid \\
& < LTL\_formula > \vee < LTL\_formula > \mid \\
& \bigcirc < LTL\_formula > \mid \Box < LTL\_formula > \mid \\
& \Diamond < LTL\_formula > \mid \\
& < LTL\_formula > U < LTL\_formula >
\end{aligned}
$$

The truth of an LTL formula is defined with respect to a trajectory and an initial state. In the following, p denotes a propositional formula, s_i's are states, σ is the trajectory $s_0, s_1, \ldots$, and f_i's denote LTL formulas as defined above.

- $(s_j, \sigma) \models p$ iff p is true in s_j.
- $(s_j, \sigma) \models \neg f$ iff $(s_j, \sigma) \not\models f$.
- $(s_j, \sigma) \models f_1 \wedge f_2$ iff $(s_j, \sigma) \models f_1$ and $(s_j, \sigma) \models f_2$.
- $(s_j, \sigma) \models f_1 \vee f_2$ iff $(s_j, \sigma) \models f_1$ or $(s_j, \sigma) \models f_2$.
- $(s_j, \sigma) \models \bigcirc f$ iff $(s_{j+1}, \sigma) \models f$.
- $(s_j, \sigma) \models \Box f$ iff $(s_k, \sigma) \models f$, for all $k \geq j$.
- $(s_j, \sigma) \models \Diamond f$ iff $(s_k, \sigma) \models f$, for some $k \geq j$.
- $(s_j, \sigma) \models f_1$ U f_2 iff there exists $k \geq j$ such that $(s_k, \sigma) \models f_2$ and for all i, $j \leq i < k, (s_i, \sigma) \models f_1$.

A variation of Linear Temporal Logic used in this work is Past Time Linear Temporal Logic[1] which is defined with the following syntax

$$
\begin{aligned}
< PTLTL\_formula > \ ::= \ & < prop\_formula > \ | \neg < PTLTL\_formula > \ | \\
& < PTLTL\_formula > \wedge < PTLTL\_formula > \ | \\
& < PTLTL\_formula > \vee < PTLTL\_formula > \ | \\
& [*] < PTLTL\_formula > \ | \\
& < * >< PTLTL\_formula > \ | \\
& (*) < PTLTL\_formula > \ | \\
& < PTLTL\_formula > Ss < PTLTL\_formula > \ | \\
& < PTLTL\_formula > Sw < PTLTL\_formula > \ | \\
& [< PTLTL\_formula >, < PTLTL\_formula >)s \ | \\
& [< PTLTL\_formula >, < PTLTL\_formula >)w \ | \\
& start(< PTLTL\_formula >) \ | \\
& end(< PTLTL\_formula >)
\end{aligned}
$$

The truth of a PTLTL formula is defined with respect to a trajectory and a current state. In the following, p denotes a propositional formula, s_i's are states, σ is the trajectory $s_0, s_{-1}, \ldots$, and f_i's denote PTLTL formulas as defined above.

- $(s_j, \sigma) \models p$ iff p is true in s_j.
- $(s_j, \sigma) \models \neg f$ iff $(s_j, \sigma) \not\models f$.
- $(s_j, \sigma) \models f_1 \wedge f_2$ iff $(s_j, \sigma) \models f_1$ and $(s_j, \sigma) \models f_2$.
- $(s_j, \sigma) \models f_1 \vee f_2$ iff $(s_j, \sigma) \models f_1$ or $(s_j, \sigma) \models f_2$.
- $(s_j, \sigma) \models (*)f$ iff $(s_{j-1}, \sigma) \models f$.
- $(s_j, \sigma) \models [*]f$ iff $(s_k, \sigma) \models f$, for all $k \leq j$.
- $(s_j, \sigma) \models < * > f$ iff $(s_k, \sigma) \models f$, for some $k \leq j$.

[1] http://fsl.cs.uiuc.edu/index.php/Past_Time_Linear_Temporal_Logic

- $(s_j, \sigma) \models f_1 S s f_2$ iff there exists $k \leq j$ such that $(s_k, \sigma) \models f_2$ and for all i, $k < i \leq j, (s_i, \sigma) \models f_1$.

- $(s_j, \sigma) \models f_1 S w f_2$ iff $(s_j, \sigma) \models [*] f_1$ or $(s_j, \sigma) \models f_1 S s f_2$.

- $(s_j, \sigma) \models start(f)$ iff $(s_{j-1}, \sigma) \models \neg f$ and $(s_j, \sigma) \models f$.

- $(s_j, \sigma) \models end(f)$ iff $(s_{j-1}, \sigma) \models f$ and $(s_j, \sigma) \models \neg f$.

- $(s_j, \sigma) \models [f_1, f_2) s$ iff there exists $k \leq j$ such that $(s_k, \sigma) \models f_1$ and for all i, $k \leq i \leq j, (s_i, \sigma) \models \neg f_2$.

- $(s_j, \sigma) \models [f_1, f_2) w$ iff $(s_j, \sigma) \models [*] \neg f_1$ or $(s_j, \sigma) \models [f_1, f_2) s$.

2.2 RIL-D

The RIL-D language is for the human to specify a directive to the robot, and since they are in an interactive setting, the human expects not only the robot to act on that directive, but also to give a verbal response to the human. The directive given to the robot may specify exact actions that need to be executed, may have a sequence of steps to be taken, may have iterative statements, may have conditional actions, may specify non-deterministic choices, may specify certain goals that needs to be achieved, and may include observational commands.

The responses expected from the robot include:

1. confirmation that a directed action was executed or a goal was achieved;

2. refusal that an action could not be executed or a goal could not be achieved;

3. a result of an observational command;

4. a question back to the human, when the human interrupts with a contradictory or confusing request while the robot is executing a previous directive; and

5. a question to the human, when the robot faces multiple choices and cannot decide which one to take.

To illustrate these responses, consider the following examples:

1. a) Given the scenario in which the robot is at the start of a hallway, the directive "Continue all the way to the end and then turn right" will result in the robot returning the confirmation "done" after reaching the end of the hallway and completing the action to turn to the right.

   b) If there is a reachable door in the vicinity of the robot, the achievement command "When you get to the door let me know" will result in a confirmation response "done" once the door is reached.

2. a) Given a variant of the scenario above in which the robot is at the start of an impassable hallway, possibly filled with debris, the directive "Continue all the way to the end and then turn right" will result in the robot returning the refusal "failed" since it cannot complete the directive.

   b) If there is not a reachable door in the vicinity of the robot, the achievement command "When you get to the door let me know" will result in the refusal "failed" since the robot cannot achieve being at a door without a door being present.

3. a) For the situation in which the robot has just entered a room with a blue box to the right of the entrance in a corner, an observational command such as "Is there a blue box in the corner to your right?" will result in the confirmation "yes" once the robot has successfully observed the positioning of the blue box.

   b) If there is a green box numbered with a 7, the observational command "What is the number on the green box?" will return the value "7" after the robot has observed the number on the box.

4. a) Given that the robot knows that there only exist pink boxes and yellow blocks, a directive to "Pick up the pink block" does not make sense and would result

in the robot responding with the request “clarify” to indicate the unintelligible directive.

b) If the robot is given contradictory commands such as the goal “Go to the door ahead” followed by “Never go near a door” the robot would recognize that the second command prevents the completion of the first and would request the human to “clarify” and provide feedback on what commands to obey.

5. a) For the situation with the robot in a hallway in which the path ahead of it forks to the left and right, the directive “Continue down the hallway” would result in the robot returning the question “left or right” then awaiting the human’s reply.

b) In this situation, the robot has already visited several hallways, previously observed several blue boxes, and is currently in the middle of another hallway that continues forward into an area that has not yet been explored. If the robot is given the command “Pick up all of the blue boxes” it will respond by asking “forward or backward” to determine whether to go back and pick up the known boxes first versus moving forward to find new blue boxes in the unexplored areas before returning to the original hallways.

The Syntax of RIL-D

The syntax of RIL-D is specified in Table 2.2 along with brief descriptions.

Syntactically, RIL-D takes Golog, removes test actions and adds temporal formulas and $goal(self, \phi)$ constructs. In Golog a test action forces one to chose the trajectory corresponding to the program before the test action in such a way that the test action holds true. This language does not allow such planning via test actions. Planning is directly specified via $goal(self, \phi)$. However, the language does have observational commands which are similar to test action but the purpose is that the human director may command the robot to make an observation which is then returned as the value observed or a failure to make the requested observation. The value to be returned by the observation is not known until the observation action is actually performed.

	Syntax	Intuitive Meaning
1	Simple Action: An action a is an RIL-D program.	Execute action a, such as *turn_right*
2	Parameterized Action: If $f(X_1 \dots X_n)$ is a formula and $a(X_1 \dots X_n)$ is an action, $a(X_1 \dots X_n) : f(X_1 \dots X_n)$ is an RIL-D program.	Execute action $a(X_1 \dots X_n)$ where $X_1 \dots X_n$ satisfy the formula $f(X_1 \dots X_n)$
3	Parallel Action: If a and b are actions (simple, parameterized, or parallel), then $a \| \| b$ is an RIL-D program.	Execute actions a and b in parallel.
4	Sensing: If $X_1 \dots X_n$ are variables of sorts $s_1 \dots s_n$ and $f(X_1 \dots X_n)$ is a formula, then $sense(X_1 \dots X_n) : f(X_1 \dots X_n)$ is an RIL-D program.	Sense the values of $X_1 \dots X_n$ where $X_1 \dots X_n$ satisfy the formula $f(X_1 \dots X_n)$.
5	Observational Command: If ψ is a formula, then $sense() : \psi$ is an RIL-D program.	Sense whether or not the suggested observation ψ holds true in the current state of the world.
6	Self Goal: If $self$ is the robot agent, $\phi(X_1 \dots X_n)$ is a temporal formula, and $f(X_1 \dots X_n)$ is a formula, then $goal(self, \phi(X_1 \dots X_n)) : f(X_1 \dots X_n)$ is an RIL-D program.	Create and execute a plan for $self$ to satisfy $\phi(X_1 \dots X_n)$ where $X_1 \dots X_n$ satisfy the formula $f(X_1 \dots X_n)$.
7	Sequence: If a and b are RIL-D programs, then $a; b$ is an RIL-D program.	Execute the RIL-D program a immediately followed by the second program. b
8	Choice: If a and b are RIL-D programs, then $a \mid b$ is an RIL-D program.	Execute either RIL-D program a or b but not both.
9	Parametric Choice: If X_1 is a variable of sort s, $p(X_1 \dots X_n)$ is a program, and $f(X_1 \dots X_n)$ is a formula, then $pick(X_1, p(X_1 \dots X_n)) : f(X_1 \dots X_n)$ is an RIL-D program.	Choose one object X_1 matching the conditions specified in $f(X_1 \dots X_n)$ and then execute program $p(X_1 \dots X_n)$ given the chosen object.
10	Condition: If a and b are RIL-D programs and ϕ is a temporal formula, then $if\ \phi\ then\ a\ else\ b$ is an RIL-D program.	If the conditions specified in ϕ hold true, execute program a, otherwise execute program b.
11	While: If a is an RIL-D program and ϕ is a past time linear temporal logic formula, then $while\ \phi\ do\ a$ is an RIL-D program.	Check if the conditions specified in ϕ hold true, and if so, execute program a then repeat the process.

Table 2.2: The Syntax of RIL-D

Syntax Examples

I now illustrate each of these constructs through the following examples and translations, with the example numbers corresponding to Table 2.2:

1. a) "Turn 90 degrees to the right":

   $turn\_right.$

   b) "Move forward a single step":

   $go\_straight\_one\_step.$

2. a) "Proceed through the doorway":

   $go\_through(X) : is(X, door).$

   b) "Pick up a blue box":

   $pick\_up(X) : is(X, box) \wedge has(X, color, blue).$

3. a) "Raise your right arm while turning 90 degrees to the right":

   $raise\_right\_arm||turn\_right.$

   b) "Push the door while turning to the left":

   $push(X) : is(X, door)||turn\_left.$

4. a) "What is the number on the green box":

   $sense(X) : has(Y, number\_on, X) \wedge has(Y, color, green) \wedge is(Y, box).$

   b) "Look for a green box":

   $sense(X) : is(Y, box) \wedge has(Y, location, X) \wedge has(Y color, green).$

5. a) "Is there a chair in front of you":

   $sense() : is(X, chair) \wedge has(X, location, front).$

   b) "Is there a blue box on the table":

   $sense() : has(X, on, Y) \wedge has(X, color, blue) \wedge is(X, box) \wedge is(Y, table).$

6. a) "When you get to the door, let me know":

   $goal(self, \diamond has(self, at, X)) : is(X, door).$

b) “Go out of the room”:

$goal(self, \Diamond\neg has(self, at, X)) : has(self, at, X) \wedge is(X, room).$

7. a) “Go through the door and then turn right”:

$go\_through(X) : is(X, door); turn\_right.$

b) “After you get to the door, push forward”:

$goal(self, \Diamond has(self, at, X)) : is(X, door); push.$

8. a) “Either go to the door or go to the blue box”:

$(goal(self, \Diamond has(self, at, X)) : is(X, door))|$

$(goal(self, \Diamond has(self, at, Y)) : is(Y, box) \wedge has(Y, color, blue)).$

b) “You can either turn right and pick up the blue box or turn left and pick up the pink box”:

$(turn\_right; (pick\_up(X) : has(X, color, blue) \wedge is(X, box))) |$

$(turn\_left; (pick\_up(Y) : has(Y, color, pink) \wedge is(Y, box))).$

9. a) “Select one yellow block and place it in a box”:

$pick(Y, (put\_in(Y, X) : is(X, box))) : is(Y, block) \wedge has(Y, color, yellow).$

b) “Select a hallway, excluding the hallway you are in, and go to it”:

$pick(X, goal(self, \Diamond has(self, at, X))) : is(X, hall) \wedge \neg has(self, at, X).$

10. a) “If there is a door on your right, go through it, otherwise turn around”:

$if\ sense() : has(X, location, right) \wedge is(X, door)$

$then\ go\_through(X).$

$else\ turn\_around.$

b) “If the box is not in front of you, find the direction of the box, otherwise pick up the box”:

$if\ \neg sense() : has(X, location, front) \wedge is(X, box)$

$then\ sense(Y) : has(X, location, Y).$

$else\ pick\_up(X).$

11. a) "Continue all the way to the end of the hallway":

    $while\ \neg has(self, at\_end, X) \wedge is(X, hall)$

    $do\ go\_straight\_one\_step.$

    b) "Step forward until the next command is given":

    $while\ \neg(*)has(self, new\_instruction, X) \wedge is(X, instruction)$

    $do\ go\_straight\_one\_step.$

Returning to the more complex example from the motivation, a close translation of the Director's commands can be written as:

1. $while\ \neg sense() : has(X, location, right) \wedge is(X, door)$

   $do\ go\_straight\_one\_step.$

2. $go\_through(X) : is(X, door) \wedge has(X, location, right).$

3. $sense() : has(self, at, Y) \wedge is(Y, room) \wedge is(X, box) \wedge has(X, color, green)$

   $\wedge has(X, at, Y).$

4. $goal(self, \Diamond has(self, at, X)) : is(X, box) \wedge has(X, color, green).$

5. $sense(X) : has(Y, number\_on, X) \wedge has(Y, color, green) \wedge is(Y, box).$

6. $go\_through(X) : is(X, door) \wedge has(X, location, right).$

7. $turn\_left \mid turn\_right.$

8. $while\ \neg has(self, at\_end, X) \wedge is(X, hall)$

   $do\ go\_straight\_one\_step.$

9. $sense() : is(X, box) \wedge has(X, color, pink) \wedge has(X, location, front).$

10. $pick(Y, (put\_in(Y, X) : is(X, box) \wedge has(X, color, pink))) : is(Y, block) \wedge$

    $has(Y, color, yellow).$

The Semantics of RIL-D

The semantics of an RIL-D program in essence say what are the valid ways in which the world will progress. It provides the information about the action execution and the responses given by the robot. The semantics would consider an initial state s_0[2] and generate a set of possible trajectories for a given RIL-D program, each consisting of $t_1, \ldots t_n$, where t_i is an action or a response-set.

For example, suppose that in s_0 the fluents $is(h1, hall)$, $has(self, at\_end, h1)$, $has(self, at, h1)$, $is(d1, door)$ hold true, however $has(d1, location, right)$ has not yet been sensed and the following RIL-D program is given:

$if\ sense() : has(X, location, right) \wedge is(X, door)$

$then\ go\_through(X).$

$else\ turn\_around.$

The two possible trajectories for $t_1, \ldots, t_n$, both with $n = 2$ and ROL-R response $R(X)$, are: $t_1 = go\_through(X)$, $t_2 = R(done)$ OR $t_1 = turn\_around$, $t_2 = R(done)$.

Within the set of possible trajectories, each trajectory $t_1, \ldots t_n$, denoted by α, is a trace of a program p and is defined as follows with formula ψ, temporal formula ϕ, ROL-R response $R(X)$:

1. for $p = a$ where a is an action, if the executability conditions for a are satisfied in s_0, then $n = 2$, $t_1 = a$, and $t_2 = R(done)$, otherwise $n = 1$ and $t_1 = R(failed)$.

2. for $p = a(X_1 \ldots X_m) : f(X_1 \ldots X_m)$ where $a(X_1 \ldots X_m)$ is an action, if both $f(X_1 \ldots X_m)$ and the executability conditions for $a(X_1 \ldots X_m)$ are satisfied in s_0, then $n = 2$, $t_1 = a$, and $t_2 = R(done)$, otherwise $n = 1$ and $t_1 = R(failed)$.

3. for $p = a||b$ where a and b are actions of the types a, $a(X_1 \ldots X_m)$, or $a||b$, if the executability conditions for a and b are satisfied in s_0, then $n = 2$, $t_1 = \{a, b\}$, and $t_2 = R(done)$, otherwise $n = 1$ and $t_1 = R(failed)$.

[2] This can be generalized to a history of states and actions of the form $s_0, a_1, s_1, a_2, \ldots, s_m$, if future commands may need to look back to history.

4. for $p = sense(X_1 \ldots X_m) : f(X_1 \ldots X_m)$, $n = 1$ and if there exists values $v_1 \ldots v_m$ of the sorts $X_1 \ldots X_m$ such that $f(v_1 \ldots v_m)$ holds in s_0, then $t_1 = R(v_1 \ldots v_m)$, otherwise $t_1 = R(failed)$.

5. for $p = sense() : \psi$, $n = 1$ and if ψ holds in s_0, then $t_1 = R(yes)$, otherwise $t_1 = R(no)$.

6. for $p = goal(self, \phi(X_1 \ldots X_m)) : f(X_1 \ldots X_m)$, α is a trace of p such that $f(X_1 \ldots X_m)$ holds in s_0, α satisfies $\phi(X_1 \ldots X_m)$, $t_1 = R(acknowledged)$, and $t_n = R(done)$. If no α exists that satisfies ϕ then $t_1 = R(failed)$.

7. for $p = p_1; p_2$, if there exists an i such that $s_0, t_1, \ldots, t_i$ is a trace of p_1, and $t_i, \ldots, t_{n-1}$ is a trace of p_2, then $t_n = R(done)$, otherwise $t_1 = R(failed)$.

8. for $p = p_1 \mid p_2$, if α is a trace of p_1 or if α is a trace of p_2 then $t_n = R(done)$, otherwise $t_1 = R(failed)$.

9. for $p = pick(X_1, q(X_1 \ldots X_m)) : f(X_1 \ldots X_m)$, if there exists values $v_1 \ldots v_m$ of the sorts $X_1 \ldots X_m$ such that $f(v_1 \ldots v_m)$ holds in s_0 and $t_1, \ldots, t_{n-1}$ is a trace of $q(v_1 \ldots v_m)$, then $t_n = R(done)$, otherwise if there does not exist such $v_1 \ldots v_m$, then $n = 1$ and $t_1 = R(failed)$.

10. for $p = if\ \phi\ then\ p_1\ else\ p_2$, either α is a trace of p_1 with $t_n = R(done)$ if ϕ is satisfied by the history $s_0, a_1, \ldots, s_m$ or α is a trace of p_2 with $t_n = R(done)$ if ϕ is not satisfied by the history $s_0, a_1, \ldots, s_m$.

11. for $p = while\ \phi\ do\ p_1$, either $n = 1$ with $t_1 = R(done)$ and ϕ is not satisfied by the history trace $s_0, a_1, \ldots, s_m$ or ϕ is satisfied by the trace of the history $s_0, a_1, \ldots, s_m$ and there exists some $i <= n$ such that $s_m, t_1, \ldots, t_i$ is a trace of p_1 and $s_0, a_1, \ldots, s_m$, $seq\_act\_state(t_1, \ldots, t_i)$ is a trace of the new history of p and α is a trace of p with $t_n = R(done)$.

$seq\_act\_state(t_1, \ldots, t_n)$ results in a sequence of alternating actions and states $a_{m+1}, s_{m+1}, \ldots, s_i$ that corresponds to the trace, $t_1, \ldots, t_n$ such that after completing

$t_1, \ldots, t_n$, the history would be of the form $s_0, a_1, \ldots, s_m, a_{m+1}, s_{m+1}, \ldots, s_i$ where s_i is the present state. When computing $seq\_act\_state(t_1, \ldots, t_n)$ only the actions within $t_1, \ldots, t_n$ are considered as the response-sets do not alter the state of the robot.

Semantic Examples

The following extend some of the syntax examples to demonstrate the progression of the world in which $R(X)$ is an ROL-R response:

1. a) Suppose that in s_0 the fluents $has(self, at, r1)$, $is(r1, room)$ hold true and the following RIL-D program is given:
   $turn\_right.$
   The trajectory $t_1, \ldots, t_n$ will be $t_1 = turn\_right$, $t_2 = R(done)$ with $n = 2$.

   b) Suppose that in s_0 the fluents $has(self, at, r1)$, $is(r1, room)$, $is(d1, door)$, $has(d1, location, front)$ hold true and the following RIL-D program is given:
   $go\_through(X) : is(X, door).$
   The sequence $t_1, \ldots, t_n$ will be $t_1 = go\_through(d1)$, $t_2 = R(done)$ with $n = 2$.

   c) Suppose that in s_0 the fluents $has(self, at\_end, h1)$, $has(self, at, h1)$, $is(h1, hall)$ hold true and the following RIL-D program is given:
   $go\_straight\_one\_step.$
   The sequence $t_1, \ldots, t_n$ will be $t_1 = R(failed)$ with $n = 1$. The action was not executed because the robot could not move forward when already at the end of the hall.

2. a) Suppose that in s_0 the fluents $is(b1, box)$, $has(b1, number\_on, 7)$, $has(b1, color, green)$, $has(self, at, r1)$, $is(r1, room)$ hold true and the following RIL-D program is given:
   $sense(X) : has(Y, number\_on, X) \wedge has(Y, color, green) \wedge is(Y, box).$
   The trajectory $t_1, \ldots, t_n$ will be $t_1 = R(7)$ with $n = 1$.

   b) Suppose that in s_0 the fluents $has(self, at, r1)$, $is(r1, room)$, $is(b1, box)$, $has(b1, color, blue)$, $has(b1, location, right)$ hold true and the following RIL-

D program is given:

$sense(X) : has(Y, location, X) \wedge has(Y, color, green) \wedge is(Y, box)$.

The sequence $t_1, \ldots, t_n$ will be $t_1 = R(failed)$ with $n = 1$. The sensing failed because there is no green box for which to return a location.

3. a) Suppose that in s_0 the fluents $has(self, at, r1)$, $is(r1, room)$, $is(c1, chair)$, $has(c1, location, front)$ hold true and the following RIL-D program is given:

$sense() : is(X, chair) \wedge has(X, location, front)$.

The sequence $t_1, \ldots, t_n$ will be $t_1 = R(yes)$ with $n = 1$.

b) Suppose that in s_0 the fluents $has(self, at, r1)$, $is(r1, room)$, $is(b1, box)$, $has(b1, color, blue)$ hold true and the following RIL-D program is given:

$sense() : has(X, on, Y) \wedge has(X, color, blue) \wedge is(X, box) \wedge is(Y, table)$.

The trajectory $t_1, \ldots, t_n$ will be $t_1 = R(no)$ with $n = 1$. The sensing returned a negative since, while there is a blue box, there is no blue box on a table.

4. a) Suppose that in s_0 the fluents $has(self, at, r1)$, $is(r1, room)$, $is(d1, door)$, $has(d1, location, front)$ hold true and the following RIL-D program is given:

$goal(self, \Diamond has(self, at, X)) : is(X, door)$.

A possible sequence $t_1, \ldots, t_n$ can be $t_1 = go\_straight\_one\_step$, ..., $t_8 = go\_straight\_one\_step$, $t_9 = R(done)$ with $n = 9$.

b) Suppose that in s_0 the fluents $has(self, at, r1)$, $is(r1, room)$ hold true and the following RIL-D program is given:

$goal(self, \Diamond \neg has(self, at, X)) : has(self, at, X) \wedge is(X, room)$.

The trajectory $t_1, \ldots, t_n$ will be $t_1 = R(failed)$ with $n = 1$. The achievement action failed because there is no other room to enter.

5. a) Suppose that in s_0 the fluents $has(self, at, r1)$, $is(r1, room)$, $is(d1, door)$, $is(h1, hall)$ hold true and the following RIL-D program is given:

$go\_through(X) : is(X, door); turn\_right$.

The trajectory $t_1, \ldots, t_n$ will be $t_1 = go\_through(d1)$, $t_2 = turn\_right$, $t_3 = R(done)$ with $n = 3$.

b) Suppose that in s_0 the fluents $has(self, at, r1)$, $is(r1, room)$ hold true and the following RIL-D program is given:

$goal(self, \Diamond has(self, at, X)) : is(X, door); go\_through(X).$

The sequence $t_1, \ldots, t_n$ will be $t_1 = R(failed)$ with $n = 1$. The sequence failed since there was no door to complete the first action.

6. a) Suppose that in s_0 the fluents $has(self, at, r1)$, $is(r1, room)$, $is(d1, door)$, $has(d1, location, front)$, $is(b1, box)$, $has(b1, location, right)$ hold true and the following RIL-D program is given:

$(goal(self, \Diamond has(self, at, X)) : is(X, door))|$

$(goal(self, \Diamond has(self, at, Y)) : is(Y, box)).$

A possible sequence $t_1, \ldots, t_n$ can be $t_1 = turn\_right$, $t_2 =$ $go\_straight\_one\_step$, ..., $t_6 = go\_straight\_one\_step$, $t_7 = R(done)$ with $n = 7$.

b) Suppose that in s_0 the fluents $has(self, at, r1)$, $is(r1, room)$, $is(b1, box)$, $has(b1, location, right)$, $has(b1, color, pink)$, $is(b2, box)$, $has(b2, location, left)$, $has(b2, color, green)$ hold true and the following RIL-D program is given:

$(turn\_right; (pick\_up(X) : has(X, color, blue) \wedge is(X, box)))$

$|(turn\_left; (pick\_up(Y) : has(Y, color, pink) \wedge is(Y, box))).$

The sequence $t_1, \ldots, t_n$ will be $R(failed)$ with $n = 1$. The choice failed since neither sequence was possible due to the wrong colored boxes.

c) Suppose that in s_0 the fluents $has(self, at, r1)$, $is(r1, room)$, $is(b1, box)$, $has(b1, location, right)$, $has(b1, color, blue)$, $is(b2, box)$, $has(b2, color, pink)$, $has(b2, location, left)$ hold true and the following RIL-D program is given:

$(turn\_right; (pick\_up(X) : has(X, color, blue) \wedge is(X, box))) |$

$(turn\_left; (pick\_up(Y) : has(Y, color, pink) \wedge is(Y, box))).$

There are two possible trajectories for $t_1, \ldots, t_n$ both with $n = 3$: $t_1 =$

$turn\_right$, $t_2 = pick\_up(b1)$, $t_3 = R(done)$ OR $t_1 = turn\_left$, $t_2 = pick\_up(b2)$, $t_3 = R(done)$.

7. a) Suppose that in s_0 the fluents $has(self, at, r1)$, $is(r1, room)$, $is(b1, box)$, $is(bl1, block)$, $has(bl1, color, yellow)$, $is(bl2, block)$, $has(bl2, color, yellow)$, $has(self, picked\_up, bl1)$, $has(self, picked\_up, bl2)$ hold true and the following RIL-D program is given:
$pick(Y, (put\_in(Y, X) : is(X, box))) : (is(Y, block) \wedge has(Y, color, yellow))$.
There are two possible trajectories for $t_1, \ldots, t_n$ both with $n = 2$: $t_1 = put\_in(bl1, b1)$, $t_2 = R(done)$ OR $t_1 = put\_in(bl2, b1)$, $t_2 = R(done)$.

   b) Suppose that in s_0 the fluents $has(self, at, h1)$, $is(h1, hall)$, $is(r1, room)$ hold true and the following RIL-D program is given:
$pick(X, goal(self, \Diamond has(self, at, X))) : (is(X, hall) \wedge \neg has(self, at, X))$.
The sequence $t_1, \ldots, t_n$ will be $t_1 = R(failed)$ with $n = 1$. The parametric choice failed since there is no hall matching the criteria of not including the current hall.

8. a) Suppose that in s_0 the fluents $has(self, at, h1)$, $is(h1, hall)$, $has(self, at\_end, h1)$, $is(d1, door)$ hold true, however $has(d1, location, right)$ is not known to be true or false and the following RIL-D program is given:
$if\ sense() : has(X, location, right) \wedge is(X, door)$
$then\ go\_through(X)$.
$else\ turn\_around$.
The two possible trajectories for $t_1, \ldots, t_n$, both with $n = 2$, are: $t_1 = go\_through(d1)$, $t_2 = R(done)$ OR $t_1 = turn\_around$, $t_2 = R(done)$.

   b) Suppose that in s_0 the fluents $has(self, at, r1)$, $is(r1, room)$, $is(b1, box)$, $has(b1, location, front)$ hold true and the following RIL-D program is given:

$if\ \neg sense() : has(X, location, front) \wedge is(X, box)$

$then\ sense(Y) : has(X, location, Y).$

$else\ pick\_up(X).$

The sequence $t_1, \ldots, t_n$ will be $t_1 = pick\_up(X)$, $t_2 = R(done)$ with $n = 2$.

9. a) Suppose that in s_0 the fluents $has(self, at, h1)$, $is(h1, hall)$, $\neg has(self, at\_end, h1)$ hold true and the following RIL-D program is given:

$while\ \neg has(self, at\_end, X) \wedge is(X, hall)$

$do\ go\_straight\_one\_step.$

A possible trajectory $t_1, \ldots, t_n$ can be $t_1 = go\_straight\_one\_step$, ..., $t_6 = go\_straight\_one\_step$, $t_7 = R(done)$ with $n = 7$. This sequence would be given if, after moving forward the 6 steps ($t_1, \ldots, t_6$), the robot has reached the end of hall h1. Other possible trajectories vary based on how many steps are required to reach the end of the hall.

b) Suppose that in s_0 the fluents $has(self, at, h1)$, $is(h1, hall)$, $\neg has(self, at\_end, h1)$ hold true and the following RIL-D program is given:

$while\ no\_new\_instruction$

$do\ go\_straight\_one\_step.$

A possible sequence $t_1, \ldots, t_n$ can be $t_1 = go\_straight\_one\_step$, ..., $t_6 = go\_straight\_one\_step$, $t_7 = R(done)$ with $n = 7$. This sequence would be given if immediately after t_6 (before t_7) the robot receives a new instruction from the director.

2.3 RIL-L

The RIL-L language is for the human to impart knowledge to the robot. The knowledge given to the robot may be in the form of a description of the environment, new actions the robot can perform, or the goals of other agents in the world.

The Syntax of RIL-L

The syntax of RIL-L is specified in Table 2.3 along with brief descriptions. It should be noted that while the implementation of a belief model is not within the scope of this work,

beliefs and confidence are important in communication especially when exchanging knowledge between agents.

	Syntax	Intuitive Meaning
1	New Description: If $h(X_1, \ldots, X_n)$ is a predicate and $b(X_1, \ldots, X_n)$ is a formula then $h(X_1, \ldots, X_n)$: $-b(X_1, \ldots, X_n)$ is an RIL-L statement.	Update the knowledge-base to include a rule such that if $b(X_1, \ldots, X_n)$ is satisfied, then $h(X_1, \ldots, X_n)$ holds true.
2	New Simple Action: If u is an unknown action and $a_1, \ldots, a_m$ is an ordered list of existing actions (simple, or simple in parallel) then $u \leftarrow a_1, \ldots, a_m$ is an RIL-L statement.	Update the knowledge base to include an action u defined as the actions $a_1, \ldots, a_m$ performed in order.
3	New Parameterized Action: If $u(X_1, \ldots, X_n)$ is an unknown action, $a_1, \ldots, a_m$ is an ordered list of existing actions (simple, parameterized from the variables $X_1, \ldots, X_n$, or parallel), and $f(X_1, \ldots, X_n)$ is a formula then $u(X_1, \ldots, X_n) \leftarrow a_1, \ldots, a_m : f(X_1, \ldots, X_n)$ is an RIL-L statement.	Update the knowledge base to include an action $u(X_1, \ldots, X_n)$ defined as the actions $a_1, \ldots, a_m$ performed in order such that $X_1, \ldots, X_n$ are of the sorts to satisfy $f(X_1, \ldots, X_n)$.
4	Other's Goal: If a is a non-$self$ agent, $\phi(X_1, \ldots, X_n)$ is a temporal formula and $f(X_1, \ldots, X_n)$ is a formula, then $goal(a, \phi(X_1, \ldots, X_n)) : f(X_1, \ldots, X_n)$ is an RIL-L statement.	Update the knowledge base such that a is executing a plan to satisfy $\phi(X_1, \ldots, X_n)$ where $X_1, \ldots, X_n$ satisfy the formula $f(X_1, \ldots, X_n)$.

Table 2.3: The Syntax of RIL-L

Syntax Examples

The following examples illustrate these constructs:

1. "A medical kit is a white box with a red cross on it.":

$$is(M, med\_kit) :- not\ exception(M, med\_kit) \wedge has(M, color, white) \wedge is(M, box) \wedge has(M, feature, Z) \wedge has(Z, color, red) \wedge is(Z, cross).$$

2. "To push, raise your arm then step forward.":

$$push \leftarrow raise\_right\_arm, go\_straight\_one\_step.$$

3. "To open a door, push it.":

   $open(X) \leftarrow push(X) : is(X, door)$.

4. Bob says "I am going to see John now.":

   $goal(bob, \Diamond has(bob, at, john))$.

The Semantics of RIL-L

In order to define the semantics of RIL-L, I need to precisely define a robot knowledge base. A robot knowledge base at a particular time is a set of facts about a given set of predicates (say, $p_1, \ldots p_n$) describing the domain of the world and the robot's mental state. When adding a new fact to the knowledge base, if it is in conflict with a previous fact, the new fact will take precedence starting at the current time. The knowledge base follows the inertia rule such that unless a fact is known to have been changed from one time step to the next, it is assumed that it stayed the same.

The knowledge base, KB, used in the following semantic definitions is defined as written above. The semantics of RIL-L are defined as follows for RIL-L program p.

1. for $p = h(X_1, \ldots, X_n)$: $-b(X_1, \ldots, X_n)$ where $h(X_1, \ldots, X_n)$ is a predicate and $b(X_1, \ldots, X_n)$ is a formula, KB = $KB \cup \{h(X_1, \ldots, X_n)$: $-b(X_1, \ldots, X_n)\}$.

2. for $p = u \leftarrow a_1, \ldots, a_m$ where u is an unknown action and $a_1, \ldots, a_m$ is an ordered list of existing actions, KB = $KB \cup \{u \leftarrow a_1, \ldots, a_m\}$.

3. for $p = u(X_1, \ldots, X_n) \leftarrow a_1, \ldots, a_m : f(X_1, \ldots, X_n)$ where $u(X_1, \ldots, X_n)$ is an unknown action, $a_1, \ldots, a_m$ is an ordered list of known actions and $f(X_1, \ldots, X_n)$ is a formula, KB = $KB \cup \{u(X_1, \ldots, X_n) \leftarrow a_1, \ldots, a_m : f(X_1, \ldots, X_n)\}$.

4. for $p = goal(a, \phi(X_1, \ldots, X_n)) : f(X_1, \ldots, X_n)$ where a is a non-$self$ agent, $\phi(X_1, \ldots, X_n)$ is a temporal formula and $f(X_1, \ldots, X_n)$ is a formula, KB = $KB \cup \{goal(a, \phi(X_1, \ldots, X_n)) : f(X_1, \ldots, X_n)\}$.

2.4 RIL-Q

The RIL-Q language is for the human to request specific knowledge from the robot. This knowledge comes from what the robot has already learned from previous observations of the environment.

The Syntax of RIL-Q

The syntax of RIL-Q is specified in Table 2.4 along with brief descriptions.

	Syntax	Intuitive Meaning
1	Simple Query: If Φ is a first-order logic formula without free variables, then $query(\Phi)$ is an RIL-Q query.	Queries whether Φ holds true with respect to the robot's current knowledge base.
2	Parameterized Query: If $\Phi(X_1, \ldots, X_n)$ is a first-order logic formula with free variables $X_1, \ldots X_n$, then $query(\lambda X_1 \ldots . \lambda X_n . \Phi(X_1, \ldots, X_n))$ is an RIL-Q query.	Queries the values of $(X_1, \ldots, X_n)$ that would cause $\Phi(X_1, \ldots, X_n)$ to hold with respect to the robot's knowledge base at the current state.

Table 2.4: The Syntax of RIL-Q

Syntax Examples

The following examples illustrate these constructs:

1. "Was there a blue box near the table?":

   $query(is(X, box) \wedge has(X, color, blue) \wedge is(T, table) \wedge has(X, near, T)).$

2. "In what room is box number 7?":

   $query(\lambda R.is(R, room) \wedge is(X, box) \wedge has(X, number\_on, 7) \wedge has(X, at, R)).$

The Semantics of RIL-Q

The knowledge base, KB, used in the following semantic definitions is defined as in Section 2.3. The semantics of RIL-Q are defined as follows for RIL-Q program p.

1. for $p = query(\Phi)$ where Φ is a first-order logic formula without free variables, if $KB \models \Phi$ the response is "yes", if $KB \models \neg\Phi$ the response is "no", and if $KB \not\models \Phi$ and $KB \not\models \neg\Phi$ the response is "unknown".

2. for $p = query(\lambda X_1 \ldots \lambda X_n . \Phi(X_1, \ldots, X_n))$ where $\Phi(X_1, \ldots, X_n)$ is a first-order logic formula with free variables $X_1, \ldots X_n$, returns the values of $v_1, \ldots, v_n$ such that $KB \models \Phi(v_1, \ldots, v_n)$. If there is no $v_1, \ldots, v_n$ such that $KB \models \Phi(v_1, \ldots, v_n)$ the response is "unknown".

2.5 RIL-A

The RIL-A language is the formal representation of the answer given by the human to a previous question from the robot in the language ROL-Q.

The Syntax of RIL-A

The response given in RIL-A will depend on the specific type of ROL-Q question. A table of question / response pairs is shown in Table 2.5 using the following definition of a goal conflict. A conflict between two goals, ϕ_1 and ϕ_2, occurs when each goal can be individually achieved according to RIL-D but both goals cannot be achieved when executed together as a single joint goal $\phi_1 \wedge \phi_2$. A conflict can be removed by any of the following means: removal of at least one of the goals, modification to at least one of the goals such that both goals can be achieved together according to RIL-D, or an RIL-L modification to the knowledge base such that the two goals can subsequently be achieved together according to RIL-D. A conflict on a single goal, ϕ, occurs when the goal cannot be achieved according to RIL-D. Such a conflict can be removed through similar methods as before: removal of the goal itself, modification to the goal such that the goal can be achieved according to RIL-D, or an RIL-L modification to the knowledge base such that the goal can subsequently be achieved according to RIL-D.

Syntax Examples

The following examples illustrate these constructs:

1. When given the choice between picking up a box on the right or a box on the left, the robot may query:
   ROL-Q: "$select((turn\_right; pick\_up(b1)), (turn\_left; pick\_up(b2)))$"
   RIL-A: $turn\_right; pick\_up(b1)$

	Question	Syntax
1	$select((a_{11};\ldots;a_{1n_1}),\ldots,(a_{k1};\ldots;a_{kn_k}))$.	If $(a_1;\ldots;a_n)$ is one of $(a_{11};\ldots;a_{1n_1}),\ldots,(a_{k1};\ldots;a_{kn_k})$ or a new plan of the same form, then $(a_1;\ldots;a_n)$ is an RIL-A answer to this question.
2	$should(a_1;\ldots;a_n)$ or $should(a_1;\ldots;a_n,\phi)$.	Either yes or no are RIL-A answers to this question.
3	$clarify(\phi)$ or $clarify(\phi_2,\phi_1)$.	Either an RIL-D program or an RIL-L statement is an RIL-A answer to this question with the intuitive understanding that the answer removes the conflict.
4	$\lambda X_1.\ldots\lambda X_n.\Phi(X_1,\ldots X_n)$.	$X_1,\ldots X_n$ is an RIL-A answer with the intuitive understanding that the values in $X_1,\ldots X_n$ should satisfy the formula Φ.

Table 2.5: The Syntax of RIL-A

2. Following the previous example, the robot could also query:

   ROL-Q: "$should(turn\_right; pick\_up(b1), (pick\_up(X) : is(X, box)$
   $\wedge(has(X, location, right) \vee has(X, location, left))))$"

   RII-A: yes.

3. If the robot is given the goal of obtaining a medical kit but does not know how to recognize a medical kit, it may query:

   ROL-Q: "$clarify(goal(self, \Diamond has(self, have, M)) : is(M, med\_kit))$"

   RIL-A: $is(M, med\_kit) :- not\ exception(M, med\_kit) \wedge has(M, color, white)$
   $\wedge is(M, box) \wedge has(M, feature, Z)$
   $\wedge has(Z, color, red) \wedge is(Z, cross).$

4. If the robot is searching for the box numbered 7, it may request more information such as the color as follows:

   ROL-Q: "$\lambda Y.is(X, box) \wedge has(X, number\_on, 7) \wedge has(X, color, Y)$."

   RIL-A: $green$.

The Semantics of RIL-A

The knowledge base, KB, used in the following semantic definitions is defined as in Section 2.3. The definition of a conflict used below can be found at the beginning of this section. The semantics of RIL-A are defined as follows for RIL-A program p.

1. for $p = (a_1; \ldots; a_n)$ where the question is $select((a_{11}; \ldots; a_{1n_1}), \ldots, (a_{k1}; \ldots; a_{kn_k}))$ and $(a_1; \ldots; a_n)$ is one of $(a_{11}; \ldots; a_{1n_1}), \ldots, (a_{k1}; \ldots; a_{kn_k})$ or a new plan of the same form, $a_1; \ldots; a_n$ is a new directive in the RIL-D language and follows the semantics of the sequence statement.

2. for $p =$ "yes" where the question is either $should(a_1; \ldots; a_n)$ or $should(a_1; \ldots; a_n, \phi)$, $a_1; \ldots; a_n$ is a new directive in the RIL-D language and follows the semantics of the sequence statement.

3. for $p =$ "no" where the question is either $should(a_1; \ldots; a_n)$ or $should(a_1; \ldots; a_n, \phi)$, the directive chosen to execute cannot be $a_1; \ldots; a_n$.

4. for $p = p_1$ where p_1 is an RIL-D program, if the question is $clarify(\phi)$, then the execution of p_1 according to the semantics of RIL-D removes the conflict on the goal ϕ. If the question is $clarify(\phi_2, \phi_1)$, then the execution of p_1 according to the semantics of RIL-D removes the conflict between goals ϕ_1 and ϕ_2.

5. for $p = p_1$ where p_1 is an RIL-L program, if the question is $clarify(\phi)$, then the execution of p_1 according to the semantics of RIL-L removes the conflict on goal ϕ due to the updated knowledge base of the robot. If the question is $clarify(\phi_2, \phi_1)$, then the execution of p_1 according to the semantics of RIL-L removes the conflict between goals ϕ_1 and ϕ_2 due to the updated knowledge.

6. for $p = v_1, \ldots v_n$ where the query $\lambda X_1 \ldots . \lambda X_n . \Phi(X_1, \ldots X_n)$ was given, KB = $KB \cup \{\Phi(v_1, \ldots, v_n)\}$.

2.6 ROL-R

When the robot receives a command or a directive in the language RIL-D, it processes that command and may respond to that directive. For example, it may say that the given command is not doable or that it has executed that command.

The semantics of RIL-D only require a simple ROL-R syntax such as yes, no, $done$, $failed$, $acknowledged$, or the value(s) $v_1, \ldots, v_n$ from a sensing action. These replies are intended to be simplistic to provide the basic details on whether or not an RIL-D program was received, satisfied, or failed. Using the context of the RIL-D program combined with the ROL-R output, a language generation system could provide a more varied and natural means of expressing the ROL-R report.

The Semantics of ROL-R

The knowledge base, KB, used in the following semantic definitions is defined as in Section 2.3. The semantics of ROL-R are defined as follows for ROL-R program p.

1. for $p =$ "yes" in reply to an RIL-D observational command $sense() : \phi$, $KB = KB \cup \{\phi\}$.

2. for $p =$ "no" in reply to an RIL-D observational command $sense() : \phi$, $KB = KB \cup \{\neg\phi\}$.

3. for $p =$ "$done$" in reply to an RIL-D directive p_2, the directive p_2 was completed according to the semantics of RIL-D.

4. for $p =$ "$failed$" in reply to an RIL-D directive p_2, the directive p_2 was not able to be completed according to the semantics of RIL-D.

5. for $p =$ "$acknowledged$" in reply to an RIL-D goal ϕ, there is a trajectory that satisfies ϕ.

6. for $p = v_1, \ldots, v_n$ in reply to an RIL-D sensing directive $sense(X_1, \ldots, X_n) : f(X_1, \ldots, X_n)$, $KB = KB \cup \{f(v_1, \ldots, v_n)\}$.

2.7 ROL-A

When the robot receives a query in the language RIL-Q it may answer it by yes, no, $unknown$, or when the RIL-Q question is $\lambda X_1, \ldots \lambda X_n \Phi(X_1, \ldots, X_n)$, with $n \geq 1$ then it may give the value of $X_1, \ldots, X_n$. Similarly to ROL-R, the ROL-A could be combined with the prompting RIL-Q query, or even the English used to form the query, to generate a natural language answer to the query.

To continue the example from section RIL-Q: Given that the robot has previously seen the box with the number 7 inside of room r3, the result to the query "$query(\lambda R.is(R, room) \wedge is(X, box) \wedge has(X, number\_on, 7) \wedge has(X, at, R))$" would simply be $r3$.

The Semantics of ROL-A

The knowledge base, KB, used in the following semantic definitions is defined as in Section 2.3. The semantics of ROL-A are defined as follows for ROL-A program p.

1. for $p =$ "yes" in reply to an RIL-Q query $query(\Phi)$, $KB \models \Phi$.
2. for $p =$ "no" in reply to an RIL-Q query $query(\Phi)$, $KB \models \neg\Phi$.
3. for $p =$ "$unknown$" in reply to an RIL-Q query $query(\Phi)$, $KB \not\models \Phi$ and $KB \not\models \neg\Phi$.
4. for $p =$ "$unknown$" in reply to an RIL-Q query $query(\lambda X_1 \ldots . \lambda X_n .$ $\Phi(X_1, \ldots, X_n))$, there is no $v_1, \ldots, v_n$ such that $KB \models \Phi(v_1, \ldots, v_n)$.
5. for $p = v_1, \ldots, v_n$ in reply to an RIL-Q query $query(\lambda X_1 \ldots . \lambda X_n .$ $\Phi(X_1, \ldots, X_n))$, $KB \models \Phi(v_1, \ldots, v_n)$.

2.8 ROL-Q

The syntax of ROL-Q is as shown in Table 2.6 with examples found in section 2.5.

	Syntax	Intuitive Meaning
1	Selection: If $(a_{11}; \ldots; a_{1n_1}), ..., (a_{k1}; \ldots; a_{kn_k})$ are sequences of actions, then $select((a_{11}; \ldots; a_{1n_1}), ..., (a_{k1}; \ldots; a_{kn_k}))$ is an ROL-Q question.	Prompts for which plan of action should be followed.
2	Approval: If $a_1; \ldots; a_n$ is a sequence of actions, then $should(a_1; \ldots; a_n)$ is an ROL-Q question.	Requests confirmation that the robot should undertake the stated actions $a_1; \ldots; a_n$.
3	Goal Approval: If $a_1; \ldots; a_n$ is a sequence of actions and ϕ is a current goal, then $should(a_1; \ldots; a_n, \phi)$ is an ROL-Q question.	Requests confirmation that the robot should undertake the stated actions $a_1; \ldots; a_n$ towards completing goal ϕ.
4	Goal Clarification: If ϕ is a goal, then $clarify(\phi)$ is an ROL-Q question.	Requests clarification for the goal ϕ.
5	Conflict Clarification: If ϕ_1 and ϕ_2 are goals, then $clarify(\phi_2, \phi_1)$ is an ROL-Q question.	Requests clarification in that goal ϕ_2 is in conflict with the prior goal ϕ_1.
6	Knowledge Query: If $\Phi(X_1, \ldots, X_n)$ is a first-order logic formula with free variables $X_1, \ldots X_n$, then $\lambda X_1 . \ldots \lambda X_n . \Phi(X_1, \ldots, X_n)$ is an ROL-Q question.	Requests specific knowledge in the same format as an RIL-Q query.

Table 2.6: The Syntax of ROL-Q

The Semantics of ROL-Q

The knowledge base, KB, used in the following semantic definitions is defined as in Section 2.3. The definition of a conflict used below can be found in Section 2.5. The semantics of ROL-Q are defined as follows for ROL-Q program p.

1. for $p = select((a_{11}; \ldots; a_{1n_1}), ..., (a_{k1}; \ldots; a_{kn_k}))$ where $(a_{11}; \ldots; a_{1n_1}), ...,$ $(a_{k1}; \ldots; a_{kn_k})$ are executable sequences of actions according to RIL-D, given a response $p_2 = (a_1; \ldots; a_n)$, p_2 will be executed as an RIL-D program.

2. for $p = should(a_1; \ldots; a_n)$ where $a_1; \ldots; a_n$ is an executable sequence of actions according to RIL-D, given the response "yes", the sequence $a_1; \ldots; a_n$ will be executed as an RIL-D program. Given the response "no", the sequence $a_1; \ldots; a_n$ will not be executed.

3. for $p = should(a_1; \ldots; a_n, \phi)$ where $a_1; \ldots; a_n$ is an executable sequence of actions according to RIL-D and ϕ is a current goal, given the response "yes", the sequence $a_1; \ldots; a_n$ will be executed as an RIL-D program. Given the response "no", the sequence $a_1; \ldots; a_n$ will not be executed.

4. for $p = clarify(\phi)$ where there is a conflict on goal ϕ, given the execution of an RIL-A response, the conflict on goal ϕ will be removed.

5. for $p = clarify(\phi_2, \phi_1)$ where ϕ_1 and ϕ_2 are conflicting goals, given the execution of an RIL-A response, the conflict between ϕ_1 and ϕ_2 will be removed.

6. for $p = \lambda X_1, \ldots \lambda X_n.\Phi(X_1, \ldots, X_n)$ where $\Phi(X_1, \ldots, X_n)$ is a first-order logic formula with free variables $X_1, \ldots X_n$, given the RIL-A response containing values $v_1, \ldots, v_n$, $KB = KB \cup \{\Phi(v_1, \ldots v_n)\}$.

Chapter 3

An Implementation

This chapter details how the robot languages could be implemented beginning first with a scenario demonstrating the potential use of the languages. The chapter then proceeds with an implementation of a significant portion of the RIL-D language using Answer Set Programming. These ASP rules are then used by a robot simulator to demonstrate that a system utilizing these language constructs can navigate through an environment and complete assigned tasks. Following the simulator is a description of a proof-of-concept system developed to translate natural language into a subset of RIL-D.

3.1 Urban Search and Rescue Example

As briefly discussed in the introduction, an important use of human robot interaction is in situations such as urban search and rescue. This is a prime example due to the nature of the environment requiring communication in both directions. When an environment is considered too dangerous for a human rescue team, robots can be guided into the area remotely. Due to the unknown nature of the area, a constant data connection between the human and robot cannot be guaranteed, making a teleoperation approach difficult. Instead, a human director can provide instructions to the robot to guide its progress, though the robot would be fully autonomous to complete these commands. The many unknowns also necessitate the ability to provide feedback to request new instructions or clarification when the environment does not match expectations. Environmental variables can include damage to the building's infrastructure, flooding, fire, and many other dynamic and fixed obstacles.

The dialogue shown in Table 3.1 is an example of a USAR scenario in which the environmental danger is represented by hostile forces. In this scenario shown in Figure 3.1, Cindy is the robot which will interact with Commanders X, Y, and Z. Initially, Commander X and Cindy are together, Commander Y is in the hallway, and Commander Z is with an injured civilian. Commander X wants Cindy to find a medical kit and bring it to Commander Z, but while doing this, Cindy must avoid being seen by the enemy.

Upon entering the hallway, Commander Y will order Cindy to follow him. Cindy will recognize the conflict and request clarification on which goal to follow. She will then find the medical kit and bring it to Commander Z, but will be detected and damaged on the way. After requesting help from Commander Z, her goal to remain undetected is overridden so that she can return to Commanders X and Y.

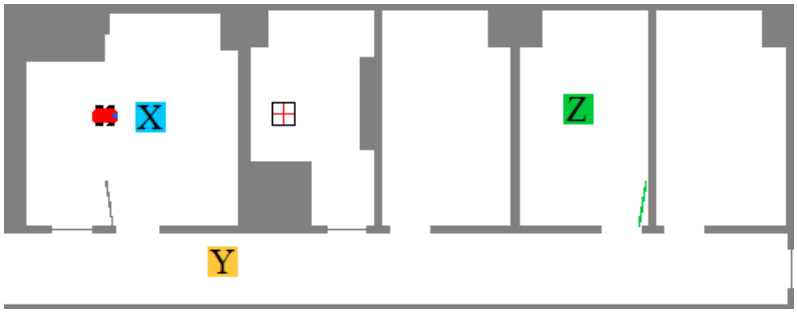

Figure 3.1: USAR Environment

3.2 An ASP Based Implementation

In this section, I demonstrate an implementation of RIL-D using Answer Set Programming. I then integrate the ASP rules with the ADE robot simulator. The ASP implementation handles the majority of RIL-D, namely the following syntactic constructs: Simple Action, Parameterized Action, Self Goal, Sequence, Choice, Parametric Choice, Condition, and While. Sensing was not included in the ASP rules since it depends on a live view of the world to obtain results from the robot's sensor. Sensing is simulated through the ASP by requiring the robot to be at the same location as the object it is attempting to sense, however this is not a true representation of how sensing would occur. Sensing was not implemented through the ADE architecture due to the limited nature of the particular ADE interface being used. Additionally, parallel actions are not presently implemented to simplify the rules by only allowing a single action to occur at any given time. Parallel actions can potentially be implemented by creating rules to limit which actions are actually able to run simultaneously and then removing the rule that restricts actions to one at a time.

The following is a subset of the logic programming rules that can be used to

	Dialogue	Translation
X	Cindy, CmdrZ really needs a medical kit.	$goal(cmdrZ, \Diamond has(cmdrZ, have, M)) : is(M, med\_kit)$
X	There should be one in the first room in the hallway down to the left	$is(m, med\_kit) \wedge has(m, at, r1) \wedge is(r1, room) \wedge is(z, room) \wedge has(z, past, r1) \wedge is(h, hallway) \wedge has(r1, connected, h) \wedge has(z, connected, h)$
C	OK.	$acknowledged$
C	Should I get it for CmdrZ?	$goal(self, \Diamond has(cmdrZ, have, m))$
X	Yes.	$goal(self, \Diamond has(cmdrZ, have, m))$
X	He is in the room with the green door.	$has(cmdrZ, at, r2) \wedge is(r2, room) \wedge is(d, door) \wedge has(d, doorconnected, r2) \wedge has(d, color, green)$
C	OK.	$acknowledged$
X	But remain undetected.	$goal(self, \Box(has(self, state, undetected)))$
C	OK.	$acknowledged$
C	What is a medical kit?	$query(\lambda X.has(med\_kit, appearance, X))$
X	It's a white box with a red cross.	$has(m, color, white) \wedge is(m, box) \wedge has(m, visual\_feature, z) \wedge has(z, color, red) \wedge is(z, cross)$
C	Alright, I'm on my way.	$acknowledged$
Y	Cindy, follow me.	$goal(self, \Box has(self, follow, cmdrY))$
C	I really need to deliver the medical kit to CmdrZ.	$clarify(goal(self, \Box has(self, follow, cmdrY)), goal(self, \Diamond has(cmdrZ, have, M)) : is(M, med\_kit))$
Y	OK.	$acknowledged$
Y	I'm going to see X now.	$goal(cmdrY, \Diamond has(cmdrY, at, cmdrX))$
C	OK.	$acknowledged$
Y	Meet me when you are done.	$goal(self, \Diamond(has(self, state, available) \wedge \Diamond has(self, at, cmdrY)))$
C	OK.	$acknowledged$
C	CmdrX, the door is closed, what shall I do?	$clarify(goal(self, \Diamond has(cmdrZ, have, M)) : is(M, med\_kit) \wedge is(d, door) \wedge has(d, state, closed))$
X	Use your hand to push it.	$goal(self, \Diamond push\_with(d, selfs\_hand))$
C	Got it.	$done$
C	CmdrZ, take the medical kit, my arm motors are not working.	$\neg(functional(self\_arm\_motors)) \rightarrow take(CmdrZ, x) \wedge is(x, med\_kit)$
Z	Thank you Cindy.	$done$
C	I need to meet CmdrY, but there are enemies.	$clarify(goal(self, \Diamond has(self, at, cmdrY)) \wedge has(E, at, h1) \wedge is(E, enemy) \wedge is(h1, hallway))$
Z	Go back to Y right away.	$goal(self, \Diamond has(self, at, cmdrY))$
C	OK, I'll do my best.	$acknowledged$

Table 3.1: Urban Search and Rescue Dialogue

express the RIL-D semantics. Based on the work in (Son et al. 2001), I define a predicate $trans(p, t_1, t_n)$ which holds in a stable model S iff $s(t_1), a_{t_1}, \ldots, s(t_n)$ is a trace of p where $s(i) = \{holds(f, i) \in S \mid f$ is a fluent$\}$ and a_i is either an action or response-set such that $occ(a_i, i) \in S$ indicates that action a_i occurs at time interval i. The other predicates used in the ASP rules are defined in Table 3.2

	Predicate	Intuitive Meaning
1	$time(X)$.	X represents a single point in time.
2	$action(X)$.	X represents a valid action.
3	$leq(X_1, X_2)$.	X_1 and X_2 are two time points in which X_1 is smaller than X_2.
4	$goal(X_1, X_2)$.	X_1 is a program that is satisfied in time X_2.
5	$htf(X_1, X_2)$.	The temporal formula X_1 holds in time X_2.
6	$proc(X)$.	X is a procedure consisting of a head and a tail.
7	$head(X_1, X_2)$.	X_1 is a procedure with the head: program X_2.
8	$tail(X_1, X_2)$.	X_1 is a procedure with the tail: program X_2.
9	$choiceAction(X)$.	X is a choice action consisting of possible programs represented by the $in(X_1, X_2)$ predicate.
10	$in(X_1, X_2)$.	X_1 is a program within the list of possible programs X_2.
11	$choiceArgs(X_1, X_2, X_3)$.	X_1 is a program in which formula X_2 holds at the current time and program X_3 is executed.
12	$hf(X_1, X_2)$.	Formula X_1 holds at time X_2.
13	$if(X_1, X_2, X_3, X_4)$.	X_1 is a program in which either program X_3 is executed if formula X_2 holds at the current time otherwise program X_4 is executed.
14	$while(X_1, X_2, X_3)$.	X_1 is a program in which so long as formula X_2 holds, program X_3 will be executed.

Table 3.2: The ASP Predicates

$$
\begin{aligned}
trans(null, T, T) &\leftarrow time(T).\\
trans(A, T, T+1) &\leftarrow time(T), action(A), A \neq null,\\
&\qquad occ(A, T).\\
trans(A, T1, T2) &\leftarrow time(T1), time(T2), leq(T1, T2),\\
&\qquad goal(A, TF), htf(TF, T2).\\
trans(A, T, T) &\leftarrow time(T), goal(A, TF), htf(TF, T).\\
trans(P, T1, T2) &\leftarrow time(T1), time(T2), leq(T1, T2),
\end{aligned}
$$

$$
\begin{aligned}
& time(T3), leq(T1, T3), \\
& leq(T3, T2), proc(P), \\
& head(P, P1), tail(P, P2), \\
& trans(P1, T1, T3), \\
& trans(P2, T3, T2).
\end{aligned}
$$

$$
\begin{aligned}
trans(N, T1, T2) \leftarrow\ & time(T1), time(T2), leq(T1, T2), \\
& choiceAction(N), in(P1, N), \\
& trans(P1, T1, T2).
\end{aligned}
$$

$$
\begin{aligned}
trans(S, T1, T2) \leftarrow\ & time(T1), time(T2), leq(T1, T2), \\
& choiceArgs(S, F, P), hf(F, T1), \\
& trans(P, T1, T2).
\end{aligned}
$$

$$
\begin{aligned}
trans(I, T1, T2) \leftarrow\ & time(T1), time(T2), leq(T1, T2), \\
& if(I, F, P1, P2), hf(F, T1), \\
& trans(P1, T1, T2).
\end{aligned}
$$

$$
\begin{aligned}
trans(I, T1, T2) \leftarrow\ & time(T1), time(T2), leq(T1, T2), \\
& if(I, F, P1, P2), not\ hf(F, T1), \\
& trans(P2, T1, T2).
\end{aligned}
$$

$$
\begin{aligned}
trans(W, T1, T2) \leftarrow\ & time(T1), time(T2), leq(T1, T2), \\
& while(W, F, P), hf(F, T1), \\
& time(T3), leq(T1, T3), \\
& leq(T3, T2), trans(P, T1, T3), \\
& trans(W, T3, T2).
\end{aligned}
$$

$$
\begin{aligned}
trans(W, T, T) \leftarrow\ & time(T), while(W, F, P), \\
& not\ hf(F, T).
\end{aligned}
$$

Below are the rules used to define the satisfiability of a propositional or temporal formula N at a time T ($hf(N, T)$ and $htf(N, T)$) and between times T and $T1$ ($hd(N, T, T1)$).

$$
hf(N, T) \leftarrow literal(N), time(T), holds(N, T).
$$

$$
\begin{aligned}
hf(N,T) &\leftarrow time(T), formula(N), formula(N1), isneg(N,N1), \\
&\quad not\ hf(N1,T). \\
not\_hc(N,T) &\leftarrow time(T), conj(N), in(N1,N), not\ htf(N1,T). \\
hf(N,T) &\leftarrow time(T), conj(N), not\ not\_hc(N,T). \\
hf(N,T) &\leftarrow time(T), disj(N), in(N1,N), htf(N1,T). \\
hf(N,T) &\leftarrow tformula(N), tf(N,N1), htf(N1,T). \\
htf(N,T) &\leftarrow formula(N), hf(N,T). \\
htf(N,T) &\leftarrow tformula(N), tf(N,until(N1,N2)), hd(N1,T,T1), \\
&\quad htf(N2,T1). \\
htf(N,T) &\leftarrow tformula(N), tf(N,always(N1)), hd(N1,T,length+1). \\
htf(N,T) &\leftarrow tformula(N), tf(N,eventually(N1)), htf(N1,T1), T <= T1, \\
&\quad time(T). \\
htf(N,T) &\leftarrow tformula(N), tf(N,next(N1)), htf(N1,T+1), time(T). \\
not\_hd(N,T,T1) &\leftarrow tformula(N), not\ htf(N,T2), T <= T2, T2 < T1, time(T), \\
&\quad time(T1), time(T2). \\
hd(N,T,T1) &\leftarrow tformula(N), htf(N,T), not\ not\_hd(N,T,T1), time(T1).
\end{aligned}
$$

The full set of logic programming rules needed for this implementation of RIL-D within a simple planning system can be found in Appendix A. This planning system was used in conjunction with the Agent Development Environment (ADE)[1] developed by Matthias Scheutz, James Kramer, and Paul Schermerhorn. The two systems are connected in the following manner, the ADE simulator generates the initial appearance of the world and waits for a command from the user to be sent to ASP system for plan generation. Once the ASP implementation of RIL-D returns a plan, the simulator executes the plan with the corresponding ROL output. Following execution, the ADE system continues waiting for further commands from the user to repeat the process starting from the current state of the simulated world.

The ADE ArchImpl.java implementation can be seen in its initial running state in Figure 3.2 and is included in Appendix B. The java portion of this system includes a

[1] http://ade.sourceforge.net/

basic demonstration of the ROL languages, namely that it can output ROL-R reports such as “acknowledged” and “done” as well as values from the simulated sensing action. In addition to some ROL-R responses, the java system is also capable of asking the RIL-Q Goal Clarification question when it is either unable to complete an action due to an unforseen obstacle, or if it is instructed to perform an action that the java system does not know.

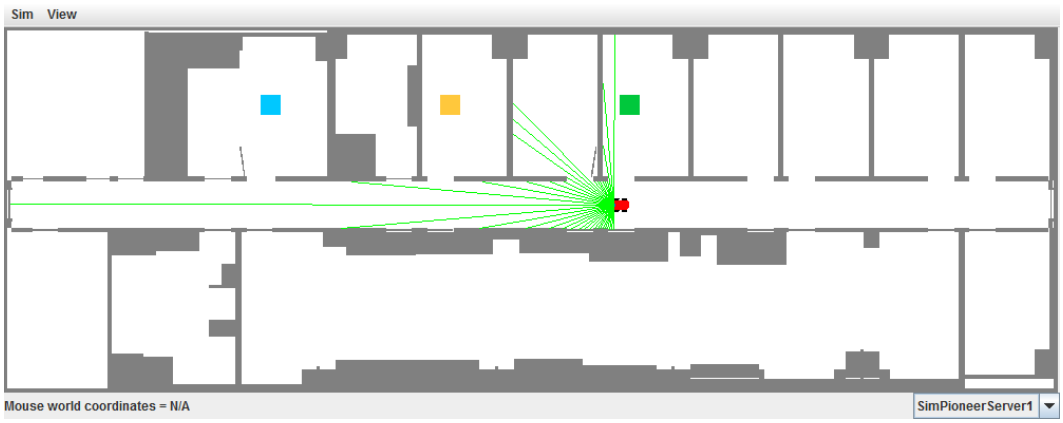

Figure 3.2: ADE Simulator Initial State

In order for the ASP logic rules to handle RIL-D statements, the statements must first be translated into the logic programming format. A sample “dialogue” manually translated to ASP is included in Appendices C, D, and E. The logic rules in each of these three appendices can be separated into four meaningful sections. The first section contains the current state of the world with the primary differences between each translation being the present location of the robot. The following set of statements simply sets up the rules that chain together the goals in the proper order along with references to those goals. The third section contains the high level definitions of the goals that were chained together, namely each goal is represented as a chain of heads and tails of procedures to represent the given goal. The final section contains the definitions of each procedure and is most closely related to the RIL-D language. It should be noted that an implementation of RIL-L New Descriptions and New Actions (either parameterized or simple) is trivial in that any additional knowledge of these forms can be provided as ASP rules such as those in the first section of each dialogue declaring the state of the world.

Implementing the RIL-L construct Other's Goal involves multi-agent planning which was not included in this program.

Beginning from the state seen in Figure 3.2, the robot is given the following commands based on the USAR dialogue from Table 3.1:

%Go through door d1 ahead on your right

$while\ \neg has(d1, location, right)$

$do\ go\_straight\_one\_step.$

$goal(self, \Diamond(has(self, at, d1) \wedge \bigcirc(has(self, prev\_action, go\_straight\_one\_step)))).$

%Go to Commander X

$goal(self, \Diamond has(self, at, cmdrX)).$

These RIL-D commands are translated into ASP as follows:

$\%while\neg has(d1, location, right)$

$\%do\ go\_straight\_one\_step.$

$while(find\_door1\_right, neg(has(d1, location, right)), go\_straight\_one\_step).$

$\%goal(self, \Diamond(has(self, at, d1)$

$\% \wedge \bigcirc(has(self, prev\_action, go\_straight\_one\_step)))).$

$achieve(go\_through\_d1, achieve\_through\_d1).$

$tf(achieve\_through\_d1, eventually(go\_through\_d1\_steps)).$

$conj(go\_through\_d1\_steps).$

$in(has(self, at, d1), go\_through\_d1\_steps).$

$in(next\_go\_straight, go\_through\_d1\_steps).$

$tf(next\_go\_straight, next(has(self, prev\_action, go\_straight\_one\_step))).$

$tformula(achieve\_through\_d1).$

$tformula(next\_go\_straight).$

$\%goal(self, \Diamond has(self, at, cmdrX)).$

$achieve(go\_to\_cmdrX, achieve\_cmdrX).$

$tf(achieve\_cmdrX, eventually(has(self, at, cmdrX))).$

$tformula(achieve\_cmdrX).$

These rules are then linked with the previously defined ASP code from Section 3.2 to run RIL-D commands through the following rules to define the order of the operations:

$goal(s1,T) :- time(T), trans(s1,0,T).$

$goal(s2,T2) :- time(T1), time(T2), trans(s2,T1,T2), goal(s1,T1), T1 <= T2.$

$goal(T) :- time(T), goal(s2,T).$

$goal(T+1) :- time(T), T < length, goal(s2,T).$

$:- notgoal(length).$

$proc(s1).$

$head(s1, find\_door1\_right).$

$tail(s1, s1b).$

$proc(s1b).$

$head(s1b, go\_through\_d1).$

$tail(s1b, null).$

$proc(s2).$

$head(s2, go\_to\_cmdrX).$

$tail(s2, null).$

The full ASP translation can be found in Appendix C, this translation includes rules containing the current state of the world needed to run these commands in conjunction with the planner from Appendix A.

When running these ASP programs together through an answer set solver, such as the clingo solver used by the simulator, the following model is generated:

$plan(0, go\_straight\_one\_step, 5, 0, west)\ \ plan(1, go\_straight\_one\_step, 4, 0, west)$

$plan(2, go\_straight\_one\_step, 3, 0, west)\ \ plan(3, go\_straight\_one\_step, 2, 0, west)$

$plan(4, turn\_right, 2, 0, north)\ \ plan(5, go\_straight\_one\_step, 2, 1, north)$

$plan(6, go\_straight\_one\_step, 2, 2, north)$

The $plan$ predicate is defined syntactically as $plan(T, A, X, Y, O)$ where T is the time-step used to order the predicates, A is the action which occurs at a given time, X and

Y are the desired spatial coordinates of the robot following the action, and O is the robot's desired orientation following A. Upon generating this plan, the java ADE system will order the result according to the time-step and store each $plan$ predicate in distinct objects.

Once the plan has been read, the robot responds with "Acknowledged" and the ordered list of plan objects is then used by the ADE $runArchitecture$ algorithm, presented in Algorithm 1, to handle the execution of the commands on the robot within the simulator.

Once the robot completes the specified plan by reaching the blue box which is intended to represent CmdrX as seen in Figure 3.3 the robot outputs "Done."

Figure 3.3: ADE Simulator Following Dialogue #1

After reaching CmdrX, the following RIL-D commands are given:

%Go to the medical supplies

$goal(self, \Diamond has(self, at, ms)).$

%Pick them up

$pick\_up(ms))$

The first RIL-D command is translated into ASP as follows:

$\%goal(self, \Diamond has(self, at, ms)).$

$achieve(go\_to\_ms, achieve\_ms).$

$tf(achieve\_ms, eventually(has(self, at, ms))).$

$tformula(achieve\_ms).$

Algorithm 1 Algorithm for $runArchitecture$

$GETSENSORS()$
if $state$ is "$turn\_right$" or "$turn\_left$" **then**
  $goalOrient \Leftarrow plan.GET(step).orient$
  $goalAngle \Leftarrow$ compute change in angle to $goalOrient$
  $rotVel \Leftarrow$ compute rotational velocity
  if $goalAngle <= 1$ **then**
    $rotVel \Leftarrow 0; step \Leftarrow step + 1$; $state \Leftarrow plan.GET(step).action$
  end if
else if $state$ is "$go\_straight\_one\_step$" **then**
  $goalDist \Leftarrow GETDIST(x, y)$
  if obstacle in front and $goalDist <= 0.1$ **then**
    $transVel \Leftarrow 0$; $rotVel \Leftarrow 0$
    $step \Leftarrow step + 1$; $state \Leftarrow plan.GET(step).action$
  else if obstacle in front and closer than $goalDist$ **then**
    $DISPLAY($"$clarify(go\_straight\_one\_step)$"$)$
    $transVel \Leftarrow 0$; $rotVel \Leftarrow 0$; $state \Leftarrow$ "$stop$"
  else
    $transVel \Leftarrow 0.7$
    if $goalDist < 1$ **then**
      $transVel \Leftarrow goalDist/3$
    end if
  end if
  if $goalDist < 0.05$ **then**
    $transVel \Leftarrow 0$; $rotVel \Leftarrow 0$
    $step \Leftarrow step + 1$; $state \Leftarrow plan.GET(step).action$
  end if
else if $state$ contains "$pick\_up$" **then**
  $transVel \Leftarrow 0$; $rotVel \Leftarrow 0$
  $object \Leftarrow GETOBJECT(state); PICKUP(object)$
  $step \Leftarrow step + 1$; $state \Leftarrow plan.GET(step).action$
else if $state$ contains "put_down" **then**
  $transVel \Leftarrow 0$; $rotVel \Leftarrow 0$
  $object \Leftarrow GETOBJECT(state); PUTDOWN(object)$
  $step \Leftarrow step + 1$; $state \Leftarrow plan.GET(step).action$
else if $state$ contains "$sense$" **then**
  $transVel \Leftarrow 0$; $rotVel \Leftarrow 0; sensed \Leftarrow GETSENSED()$
  for all $value$ in $sensed$ **do**
    $DISPLAY(value)$
  end for
  $step \Leftarrow step + 1$; $state \Leftarrow plan.GET(step).action$
else if $state$ is "$stop$" **then**
  $transVel \Leftarrow 0$; $rotVel \Leftarrow 0$
end if
$SETVELS(transVel, rotVel)$

The second RIL-D command to pick up the supplies does not require this extra translation step as it is already a simple action that can be represented within the $head$ predicate as seen below when linking the ASP translation with the code from Section 3.2:

$goal(s1, T) : -time(T), trans(s1, 0, T).$

$goal(s2, T2) : -time(T1), time(T2), trans(s2, T1, T2), goal(s1, T1), T1 <= T2.$

$goal(T) : -time(T), goal(s2, T).$

$goal(T + 1) : -time(T), T < length, goal(s2, T).$

$: -notgoal(length).$

$proc(s1).$

$head(s1, go\_to\_ms).$

$tail(s1, null).$

$proc(s2).$

$head(s2, pick\_up(ms)).\%Note, thisisanaction$

$tail(s2, null).$

The full translation for these commands, with rules representing the environment, can be found in Appendix D.

Running this program with the ASP code from Appendix A through the answer set solver generates following model:

$plan(0, turn\_left, 2, 2, west)\ \ plan(1, turn\_left, 2, 2, south)$

$plan(2, go\_straight\_one\_step, 2, 1, south)$

$plan(3, go\_straight\_one\_step, 2, 0, south)$

$plan(4, turn\_left, 2, 0, east)\ \ plan(5, go\_straight\_one\_step, 3, 0, east)$

$plan(6, go\_straight\_one\_step, 4, 0, east)\ \ plan(7, turn\_left, 4, 0, north)$

$plan(8, go\_straight\_one\_step, 4, 1, north)$

$plan(9, go\_straight\_one\_step, 4, 2, north)$

$plan(10, pick\_up(ms), 4, 2, north)$

As before, the simulator parses each predicate into an ordered list of objects based on the time-step, then the robot responds with “Acknowledged” before beginning

the execution of the ordered plan within $runArchitecture$. Then the robot reaches the position of the yellow box in Figure 3.4 which is intended to represent the medical supplies. The yellow box is removed from the simulation as the robot has completed the command to pick it up. As before, the robot announces “Done” upon completing the given commands.

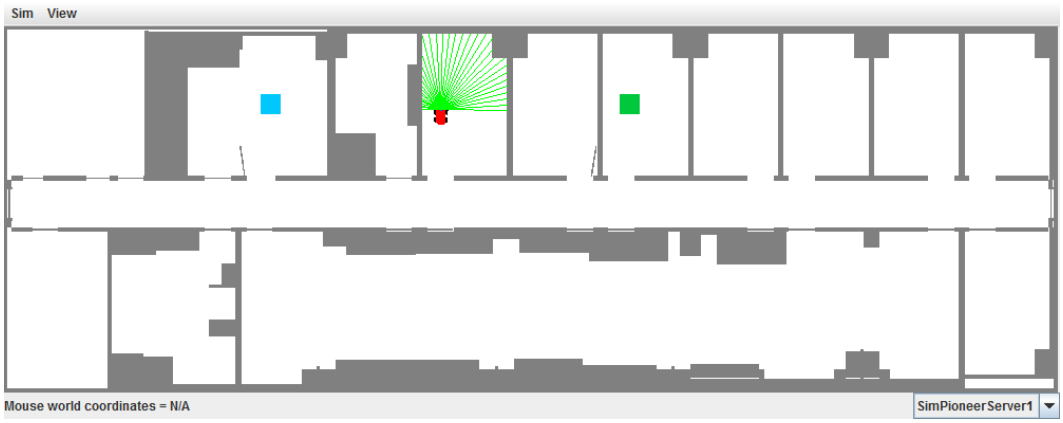

Figure 3.4: ADE Simulator Following Dialogue #2

After picking up the supplies, the following RIL-D commands are given:

%Meet Commander Z

$goal(self, \Diamond has(self, at, cmdrZ)).$

%Check on Commander Z

$sense(status(cmdrZ)).$

%If Commander Z is hurt but awake, give him the medical supplies

$if\ has(cmdrZ, status, injured) \wedge has(cmdrZ, status, conscious)$

$then\ pick(M, put\_down(M)) : has(self, carrying, M) \wedge is(M, medical\_supplies).$

The first RIL-D command is translated into ASP as follows:

$\%goal(self, \Diamond has(self, at, cmdrZ)).$

$achieve(go\_to\_cmdrZ, achieve\_cmdrZ).$

$tf(achieve\_cmdrZ, eventually(has(self, at, cmdrZ))).$

$tformula(achieve\_cmdrZ).$

As with the previous set of commands, the second RIL-D command does not require any extra translation as simple sensing can also be represented within the $head$

predicate. The third RIL-D command does need to be translated and can be seen in ASP below:

$\%if\ has(cmdrZ, status, injured) \wedge has(cmdrZ, status, conscious)$

$\%then\ pick(M, put\_down(M)) : has(self, carrying, M) \wedge is(M, medical\_supplies).$

$if(if1, awake\_injured, place\_meds, null).$

$conj(awake\_injured).$

$in(has(self, sensed, status(cmdrZ, injured)), awake\_injured).$

$in(has(self, sensed, status(cmdrZ, conscious)), awake\_injured).$

$choiceArgs(place\_meds, has(self, carrying, M), put\_down(M)) : -$

$is(M, medical\_supplies).$

These rules are linked with the other ASP code through the following rules, noting that there are three commands given in this dialogue as opposed to the previous two distinct commands:

$goal(s1, T) : -time(T), trans(s1, 0, T).$

$goal(s2, T2) : -time(T1), time(T2), trans(s2, T1, T2), goal(s1, T1), T1 <= T2.$

$goal(s3, T2) : -time(T1), time(T2), trans(s3, T1, T2), goal(s2, T1), T1 <= T2.$

$goal(T) : -time(T), goal(s3, T).$

$goal(T + 1) : -time(T), T < length, goal(s3, T).$

$: -notgoal(length).$

$proc(s1).$

$head(s1, go\_to\_cmdrZ).$

$tail(s1, null).$

$\%sense(Y) : status(cmdrZ, Y), person(cmdrZ).$

$proc(s2).$

$head(s2, sense(status(cmdrZ))).$

$tail(s2, null).$

$proc(s3).$

$head(s3, if1).$

$tail(s3, null)$.

The full translation for this dialogue can be found in Appendix E with the corresponding state of the world. This dialogue is then run through the answer set solver along with Appendix A to generate the following model:

$plan(0, turn\_left, 4, 2, west)$ $plan(1, turn\_left, 4, 2, south)$

$plan(2, go\_straight\_one\_step, 4, 1, south)$

$plan(3, go\_straight\_one\_step, 4, 0, south)$

$plan(4, turn\_left, 4, 0, east)$ $plan(5, go\_straight\_one\_step, 5, 0, east)$

$plan(6, go\_straight\_one\_step, 6, 0, east)$ $plan(7, turn\_left, 6, 0, north)$

$plan(8, go\_straight\_one\_step, 6, 1, north)$

$plan(9, go\_straight\_one\_step, 6, 2, north)$

$plan(10, sense(status(cmdrZ)), 6, 2, north)$

$plan(10, sensed, cmdrZ, injured)$ $plan(10, sensed, cmdrZ, conscious)$

$plan(11, put\_down(ms), 6, 2, north)$

As with the previous dialogues, the $plan(T, A, X, Y, O)$ predicate is used to generate a temporally ordered list of plan steps. However, in this output there is a new predicate visible of the form $plan(T, sensed, P, S)$ where P is a person and S is their status that would be reported by the sensing action. It should be noted that this is in effect retrieving the person's status from the planner rather than the robot's sensors. This was implemented in this way due to the limited nature of the chosen version of the ADE architecture being unable to sense the status of a person. Given a working robot that is truly able to sense the status of an individual, this new predicate would not be needed. The values from the new $plan$ predicate are saved into their own objects to be recalled when the sensing action is performed.

After reading in the current plan, the robot replies with "Acknowledged" and begins executing the runArchitecture on the action plan. Once the robot reaches the green box in Figure 3.5 intended to represent CmdrZ, the robot outputs "conscious" and "injured" as the status of CmdrZ before finishing with the statement, "Done."

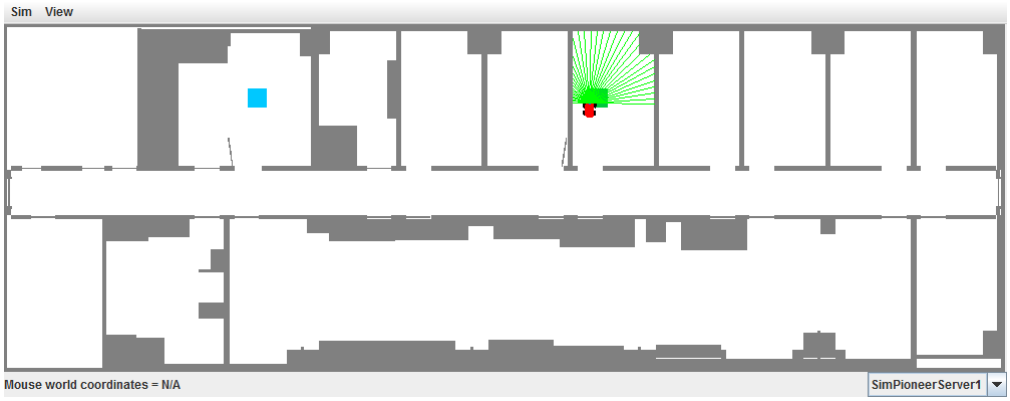

Figure 3.5: ADE Simulator Following Dialogue #3

3.3 Automated Translation of English

A demonstration of translating English into RIL-D was performed as part of a class project by Barry Lumpkin and Jenny Hastings (Hastings and Lumpkin 2012) with the assistance of Juraj Dzifcak. This project utilizes a machine learning approach to natural language translation with the assistance of an inverse lambda algorithm (Gonzalez 2010) to help the system learn the semantic definitions of more words than are actually provided in the initial dictionary. In this project, an initial dictionary of 68 semantic definitions was created as well as a training corpus of 30 sentences and a testing corpus of 10 sentences. While this is a very limited dataset, it was only intended to be a proof-of-concept demonstration of the ability to translate goal and sensing statements. Using Juraj's PCCG system (Baral et al. 2011) the initial dictionary was expanded to 172 definitions with each definition given a weighting determined by machine learning on the training corpus. The PCCG system was shown to be able to learn new words that were not initially given semantic meanings through inverse lambda and thus provided accurate translations to sentences that either contained only words that were initially given, or that contained new words that were provided in the training data but not semantically defined in the original dictionary. Of the training and testing set, the only sentence that was not translatable was from the testing set due to one word never having been included in the dictionary or training corpus. The course paper (Hastings and Lumpkin 2012) contains the details on this project as it is beyond the scope of this work.

Chapter 4

Conclusion and Future Work

In a human-robot interaction scenario one of the important modes of communication is via natural language. To facilitate this communication, in this paper, I proposed a formal high level language with multiple components. The proposed language has two main parts: RIL and ROL which refer to the Robot Input Language and the Robot Output Language.

The RIL has four sections RIL-D, RIL-L, RIL-Q and RIL-A, which express directives, learning, queries and answers, respectively. The ROL has three parts ROL-R, ROL-A and ROL-Q, which express reports, answers (to queries) and questions, respectively. The syntax and semantics of each of these sub-languages are based on their needs, and for some of them I borrow constructs from the literature and make appropriate modifications. For example, the RIL-D language borrows several constructs from GOLOG, but at the same time avoids features from GOLOG that were considered to be inappropriate from an HRI viewpoint. These GOLOG constructs are then extended with temporal logic and "goal" statements.

I demonstrate various sections of the RIL and ROL languages through a simulated robot by combining the Agent Development Environment architecture with a set of Answer Set Programming rules. The ASP rules are able to handle the majority of the RIL-D syntax to generate a plan which is then provided to the ADE simulator for execution. The simulator environment then generates the corresponding ROL output as needed.

Finally, to complete the link from natural language to a robot architecture, I described a proof-of-concept system that is able to perform translations from English into either RIL-D goal or sensing statements. This system demonstrates the ability to create a translator for English to RIL. The ASP rules show the ability to translate RIL-D into a working plan which can then be interpreted by a robot architecture as demonstrated via the ADE simulation which also provides the ROL output.

While this work provides languages to handle various human-robot dialogues, there are several areas that would be desired extensions for this research:

1. When imparting knowledge through natural language, statements are often qualified with various phrases such as “should”, “may”, or “I believe”. These phrases are intended to express a level of uncertainty in the knowledge such that the receiving party would know not to believe the statement with 100% confidence. An extension would be to provide a measure of the level of uncertainty in the dialogue so that the robot would be able to decide when to trust versus when to validate the knowledge.

2. Along the lines of providing confidence, the language does not currently provide much in the way of semantics towards the generation of a belief model. A useful extension is that the robot should be able to model its own knowledge and beliefs as well as those of the director.

3. The current language does not provide a construct for goal revision. One possible approach would be to implement a language such as ER-LTL, a non-monotonic temporal logic that is designed for the revision of goals (Zhao 2010). This would be a desirable approach as ER-LTL does not require the previous goal to be retracted in order to provide the revision, thus saving the costs of providing an entirely new goal.

4. Due to the particular domains being considered in this work, the language is designed for a human director and a robot agent such that the robot is able to receive commands, but not to command the human in return. An interesting area to move forward would be enabling the robot to take on the role of director, effectively interchanging the use of RIL and ROL.

5. Taking the architecture further into a multi-agent domain is also an exciting focus of research. This could include enabling multiple robots to work together or

modifications to how commands are received from multiple directors. In the USAR example, the robot Cindy took instructions from three different commanders and when a conflict arose, she simply asked the nearest commander to clarify what she should do. If the directors actually have different ranks within a command hierarchy, then the robot should instead base its decisions with the priority given to highest ranking directors.

6. For the RIL-Q language, the query definition $\Phi(X_1, \ldots, X_n)$ should to be extended to handle temporal constructs.

7. For the ROL-A language, it would be useful for the robot to indicate possible reasons for why a command may have failed or whether something unexpected occurs in the environment.

REFERENCES

Baral, C.; Dzifcak, J.; Gonzalez, M.; and Zhou, J. 2011. Using Inverse Lambda and Generalization to Translate English to Formal Languages. In *Proceedings of International Conference on Computational Semantics (IWCS) 2011*, Oxford.

Chen, D.L.; and Mooney, R.J.; 2011. Learning to Interpret Natural Language Navigation Instructions from Observations. *Association for the Advancement of Artificial Intelligence (AAAI) 2011*, Cambridge, MA.

Clodic, A.; Alami, R.; Montreuil, V.; Li, S.; Wrede, B.; and Swadzba, A. 2007. A study of interaction between dialog and decision for human-robot collaborative task achievement. In *Robot and Human interactive Communication, 2007. RO-MAN 2007. The 16th IEEE International Symposium on*, 913-918. IEEE.

Eberhard, K.; Nicholson, H.; Kuebler, S.; Gundersen, S.; and Scheutz, M. 2010. The Indiana cooperative remote search task (crest) corpus. In *Proceedings of LREC 2010: Language Resources and Evaluation Conference*.

Fainekos, G.; Girard, A.; Kress-Gazit, H.; Pappas, G. 2009. Temporal Logic Motion Planning for Dynamic Mobile Robots. *Automatica, Elsevier*, 45(2):343-352.

Fischer, K. 2011. How people talk with robots: Designing dialog to reduce user uncertainty. *AI Magazine*, 32(4):31-38.

Fong, T.; Nourbakhsh, I.; Kunz, C.; Flückiger, L.; Schreiner, J.; Ambrose, R.; Burridge, R.; Simmons, R.; Hiatt, L.; Schultz, A.; et al. 2005. The peer-to-peer human-robot interaction project. *Space*, 6750.

Fong, T.; Kunz, C.; Hiatt, L.; and Bugajska, M. 2006. The human-robot interaction operating system. In *Proceedings of the 1st ACM SIGCHI/SIGART conference on Human-robot interaction*, 41-48. ACM.

Gonzalez, M. 2010. An inverse lambda calculus algorithm for natural language processing. *Master's thesis, Arizona State University*.

Hastings, J.; and Lumpkin, B. 2012. From English to Robot Input Language. *Unpublished manuscript, Arizona State University*.

Kress-Gazit, H.; Fainekos, G.; Pappas, G. 2009. Temporal Logic-based Reactive Mission and Motion Planning. *IEEE Transactions on Robotics*, 25(6):1370-1381.

Levesque, H.; Reiter, R.; Lespérance, Y.; Lin, F.; and Scherl, R. 1997. Golog: A logic programming language for dynamic domains. *The Journal of Logic Programming* 31(1- 3):59-83.

Pnueli, A. 1977. The temporal logic of programs. In *Proceedings of the 18th IEEE symposium foundations of computer science*, 46-57.

Scheutz, M.; Cantrell, R.; and Schermerhorn, P. 2011. Toward Humanlike Task-Based Dialogue Processing for Human Robot Interaction. *AI Magazine*, 32(4):77-84.

Scheutz, M.; Schermerhorn, P.; Kramer, J.; and Anderson, D. 2007. First steps toward natural human-like HRI. *Autonomous Robots*, 22(4):411-423.

Scheutz, M.; Schermerhorn, P.; Kramer, J.; and Middendorff, C. 2006. The utility of affect expression in natural language interactions in joint human-robot tasks. In *Proceedings of the 1st ACM International Conference on Human-Robot Interaction*, 226-233.

Scheutz, M.; Schermerhorn, P.; Middendorff, C.; Kramer, J.; Anderson, D.; and Dingler, A. 2005. Toward affective cognitive robots for human-robot interaction. In *AAAI 2005 Robot Workshop*.

Scholtz, J. 2002. Human-robot interactions: Creating synergistic cyber forces. *Multi-robot systems: from swarms to intelligent automata*.

Son, T.; Baral, C.; McIlraith, S. 2001. Planning with Different Forms of Domain Dependent Control Knowledge - An Answer Set Programming Approach. In *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR 2001*, Vienna, Austria, 226-239.

Tellex, S.; Kollar, T.; Dickerson, S.; Walter, M.R.; Banerjee, A.G.; Teller, S.; and Roy, N. 2011. Understanding Natural Language Commands for Robotic Navigation and Mobile Manipulation. In *Proceedings of AAAI 2011*.

Zhao, J. 2010. Representing and Reasoning about Goals and Policies of Agents. *Ph.D. Dissertation, Arizona State University*.

APPENDIX A

DEMOPLANNER.LP - AN ASP IMPLEMENTATION

```
%-----------------------------------------------------------------
% Robot Interaction Language
% Barry Lumpkin
%-----------------------------------------------------------------

time(0..length).

fluent(has(O,at,P)) :- is(O,object), is(P,place).
fluent(pos(self,X,Y)) :- position(X,Y).
fluent(pos(O,X,Y)) :- is(O,object), position(X,Y).
fluent(has(self,orient,D)) :- is(D,cardinal).
fluent(has(O,location,D)) :- is(O,object), is(D,direction).
fluent(has(self,carrying,O)) :- is(O,object).
fluent(has(self,at_end)).
fluent(has(self,at_end,H)) :- is(H,hallway).
fluent(has(self,at,O)) :- is(O,object).
fluent(has(self,sensed,P)) :- is(P,property).
fluent(has(O,status,S)) :- is(O,object), is(S,status).
fluent(has(self,prev_action,A)) :- action(A).

is(location(O,D),property) :- fluent(has(O,location,D)).
is(status(O,S),property) :- fluent(has(O,status,S)).

%-----------------------------------------------------------------
% RIL-D Translation
%-----------------------------------------------------------------

formula(L) :- literal(L).
formula(S) :- conj(S).
formula(S) :- disj(S).
formula(N) :- isneg(N).

hf(L,T) :- literal(L), time(T), holds(L,T).
hf(N,T) :- time(T), formula(N), formula(N1), isneg(N, N1), not hf(N1,T).

% Conjunctions
not_hc(S,T) :- time(T), conj(S), in(N,S), not htf(N,T).
hf(S,T) :- time(T), conj(S), not not_hc(S,T).

% Disjunction
hf(S,T) :- time(T), disj(S), in(N,S), htf(N,T).

% Temporal rules
```

```
hf(N,T) :- tformula(N), tf(N,N1), htf(N1,T).

tformula(N) :- formula(N).

htf(N,T) :- formula(N), hf(N,T).
htf(N,T) :- tformula(N), tf(N,until(N1,N2)), hd(N1,T,T1), htf(N2,T1).
htf(N,T) :- tformula(N), tf(N,always(N1)), hd(N1,T,length+1).
htf(N,T) :- tformula(N), tf(N,eventually(N1)), htf(N1,T1), T<=T1, time(T).
htf(N,T) :- tformula(N), tf(N,next(N1)), htf(N1,T+1), time(T).
not_hd(N,T,T1) :- tformula(N), not htf(N,T2), T<=T2, T2<T1, time(T), time(T1), time(T2).
hd(N,T,T1) :- tformula(N), htf(N,T), not not_hd(N,T,T1), time(T1).

% No action
trans(null,T,T) :- time(T).

% Action A
trans(A,T,T+1) :- time(T), action(A), A!=null, occ(A,T).
%%response(done,A,T+1) :- time(T), action(A), A!=null, occ(A,T).

% Formula testing (change to sensing)
trans(F,T,T) :- time(T), formula(F), hf(F,T).

% Achieve
trans(A,T1,T2) :- time(T1), time(T2), leq(T1,T2), achieve(A,TF), htf(TF,T2).
trans(A,T,T) :- time(T), achieve(A,TF), htf(TF,T).

% Sequence p1;p2
trans(P,T1,T2) :- time(T1), time(T2), leq(T1,T2), time(T3), leq(T1, T3), leq(T3,T2),
                  proc(P), head(P,P1), trans(P1,T1,T3), tail(P,P2), trans(P2,T3,T2).

% Choice p1|p2
trans(N,T1,T2) :- time(T1), time(T2), leq(T1,T2), choiceAction(N), in(P1,N),
                  trans(P1,T1,T2).

% pick(X,F(X),P(X))
trans(S,T1,T2) :- time(T1), time(T2), leq(T1,T2), choiceArgs(S,F,P), hf(F, T1),
                  trans(P, T1, T2).

% If formula then p1 else p2
trans(I,T1,T2) :- time(T1), time(T2), leq(T1,T2), if(I,F,P1,P2), hf(F,T1), trans(P1,T1,T2).
trans(I,T1,T2) :- time(T1), time(T2), leq(T1,T2), if(I,F,P1,P2), not hf(F,T1),
                  trans(P2,T1,T2).

% While formula do P
trans(W,T1,T2) :- time(T1), time(T2), leq(T1,T2), while(W,F,P), hf(F,T1), time(T3),
                  leq(T1, T3), leq(T3,T2), trans(P,T1,T3), trans(W,T3,T2).
trans(W,T,T) :- time(T), while(W,F,P), not hf(F,T).
```

```
%-----------------------------------------------------------------------
% Ramifications (static causal laws)
%-----------------------------------------------------------------------

% Currently detects relative location through walls. Consider adding a rule that all
                        positions must be consecutive.
location(front,X,Y2,T) :- time(T), holds(pos(self,X,Y1),T),
                        holds(has(self,orient,north),T),position(X,Y2), Y2>Y1.
location(front,X,Y2,T) :- time(T), holds(pos(self,X,Y1),T),
                        holds(has(self,orient,south),T),position(X,Y2), Y2<Y1.
location(front,X2,Y,T) :- time(T), holds(pos(self,X1,Y),T),
                        holds(has(self,orient,east),T),position(X2,Y), X2>X1.
location(front,X2,Y,T) :- time(T), holds(pos(self,X1,Y),T),
                        holds(has(self,orient,west),T),position(X2,Y), X2<X1.

location(back,X,Y2,T) :- time(T), holds(pos(self,X,Y1),T),
                        holds(has(self,orient,south),T),position(X,Y2), Y2>Y1.
location(back,X,Y2,T) :- time(T), holds(pos(self,X,Y1),T),
                        holds(has(self,orient,north),T),position(X,Y2), Y2<Y1.
location(back,X2,Y,T) :- time(T), holds(pos(self,X1,Y),T),
                        holds(has(self,orient,west),T),position(X2,Y), X2>X1.
location(back,X2,Y,T) :- time(T), holds(pos(self,X1,Y),T),
                        holds(has(self,orient,east),T),position(X2,Y), X2<X1.

location(left,X,Y2,T) :- time(T), holds(pos(self,X,Y1),T),
                        holds(has(self,orient,east),T),position(X,Y2), Y2>Y1.
location(left,X,Y2,T) :- time(T), holds(pos(self,X,Y1),T),
                        holds(has(self,orient,west),T),position(X,Y2), Y2<Y1.
location(left,X2,Y,T) :- time(T), holds(pos(self,X1,Y),T),
                        holds(has(self,orient,south),T),position(X2,Y), X2>X1.
location(left,X2,Y,T) :- time(T), holds(pos(self,X1,Y),T),
                        holds(has(self,orient,north),T),position(X2,Y), X2<X1.

location(right,X,Y2,T) :- time(T), holds(pos(self,X,Y1),T),
                        holds(has(self,orient,west),T),position(X,Y2), Y2>Y1.
location(right,X,Y2,T) :- time(T), holds(pos(self,X,Y1),T),
                        holds(has(self,orient,east),T),position(X,Y2), Y2<Y1.
location(right,X2,Y,T) :- time(T), holds(pos(self,X1,Y),T),
                        holds(has(self,orient,north),T),position(X2,Y), X2>X1.
location(right,X2,Y,T) :- time(T), holds(pos(self,X1,Y),T),
                        holds(has(self,orient,south),T),position(X2,Y), X2<X1.


holds(has(O,location,D),T) :- time(T), location(D,X,Y,T), holds(pos(O,X,Y),T),
                                is(D,direction),is(O,object).
holds(neg(has(O,location,D)),T) :- time(T), not location(D,X,Y,T), holds(pos(O,X,Y),T),
                                is(D,direction), is(O,object).
```

```
holds(has(self,at_end),T) :- time(T), not possible(go_straight_one_step,T).
holds(neg(has(self,at_end)),T) :- time(T), possible(go_straight_one_step,T).

holds(has(self,at_end,H),T) :- time(T), holds(has(self,at,P),T), end(H,P), is(P,hall),
                                is(H,hallway), has(H,hall_segment,P).
holds(neg(has(self,at_end,H)),T) :- time(T), holds(has(self,at,P),T), not end(H,P),
                                is(H,hallway).


holds(has(self,at,O),T) :- time(T), holds(pos(self,X,Y),T), holds(pos(O,X,Y),T).
holds(neg(has(self,at,O)),T) :- time(T), holds(pos(self,X,Y),T), not holds(pos(O,X,Y),T),
                                is(O,object).

%-----------------------------------------------------------------
% Actions (dynamic causal laws)
%-----------------------------------------------------------------

action(turn_right).
action(turn_left).
action(go_straight_one_step).
action(put_down(O)) :- is(O,object).
action(pick_up(O)) :- is(O,object).
action(sense(status(P))) :- is(P,person).

% Turn Right
holds(has(self,orient,north),T+1) :- occ(turn_right,T), holds(has(self,orient,west),T).
holds(has(self,orient,east),T+1) :- occ(turn_right,T), holds(has(self,orient,north),T).
holds(has(self,orient,south),T+1) :- occ(turn_right,T), holds(has(self,orient,east),T).
holds(has(self,orient,west),T+1) :- occ(turn_right,T), holds(has(self,orient,south),T).

% Turn Left
holds(has(self,orient,north),T+1) :- occ(turn_left,T), holds(has(self,orient,east),T).
holds(has(self,orient,east),T+1) :- occ(turn_left,T), holds(has(self,orient,south),T).
holds(has(self,orient,south),T+1) :- occ(turn_left,T), holds(has(self,orient,west),T).
holds(has(self,orient,west),T+1) :- occ(turn_left,T), holds(has(self,orient,north),T).

% Go Straight One Step
holds(pos(self,X+1,Y),T+1) :- occ(go_straight_one_step,T), holds(pos(self,X,Y),T),
                                holds(has(self,orient,east),T).
holds(pos(self,X-1,Y),T+1) :- occ(go_straight_one_step,T), holds(pos(self,X,Y),T),
                                holds(has(self,orient,west),T).
holds(pos(self,X,Y+1),T+1) :- occ(go_straight_one_step,T), holds(pos(self,X,Y),T),
                                holds(has(self,orient,north),T).
holds(pos(self,X,Y-1),T+1) :- occ(go_straight_one_step,T), holds(pos(self,X,Y),T),
                                holds(has(self,orient,south),T).


% Put Down Object
holds(pos(O,X,Y),T+1) :- occ(put_down(O),T), is(O,object), holds(pos(self,X,Y),T).
```

```
holds(neg(has(self,carrying,O)),T+1) :- occ(put_down(O),T), is(O,object),
                                        holds(pos(self,X,Y),T).

% Pick Up Object
holds(has(self,carrying,O),T+1) :- occ(pick_up(O),T), is(O,object).
found_contrary(pos(O,X,Y),T+1) :- occ(pick_up(O),T), holds(pos(O,X,Y),T).


% Sense status
holds(has(self,sensed,status(P,S)),T+1) :- occ(sense(status(P)),T),
                                           holds(has(P,status,S),T).


possible(turn_right,T) :- time(T).
possible(turn_left,T) :- time(T).
possible(A,T) :- action(A), time(T), executable(A,S), hf(S,T).
possible(go_straight_one_step,T) :- time(T), holds(pos(self,X,Y),T),
        holds(has(self,orient,east),T), position(X+1,Y), pos(P1,X,Y), pos(P2,X+1,Y),
        is(P1,place), is(P2,place), connected(P1,P2,east).
possible(go_straight_one_step,T) :- time(T), holds(pos(self,X,Y),T),
        holds(has(self,orient,west),T), position(X-1,Y), pos(P1,X,Y), pos(P2,X-1,Y),
        is(P1,place), is(P2,place), connected(P2,P1,east).
possible(go_straight_one_step,T) :- time(T), holds(pos(self,X,Y),T),
        holds(has(self,orient,north),T), position(X,Y+1), pos(P1,X,Y), pos(P2,X,Y+1),
        is(P1,place), is(P2,place), connected(P1,P2,north).
possible(go_straight_one_step,T) :- time(T), holds(pos(self,X,Y),T),
        holds(has(self,orient,south),T), position(X,Y-1), pos(P1,X,Y), pos(P2,X,Y-1),
        is(P1,place), is(P2,place), connected(P2,P1,north).
possible(put_down(O),T) :- time(T), is(O,object), holds(has(self,carrying,O),T).
possible(pick_up(O),T) :- time(T), is(O,object), holds(pos(self,X,Y),T),
        holds(pos(O,X,Y),T).
possible(sense(status(P)),T) :- time(T), holds(has(P,status,S),T),
        holds(pos(self,X,Y),T),holds(pos(P,X,Y),T).

nocc(A,T) :- action(A), action(B), time(T), A!=B, occ(B,T), T<length.
occ(A,T) :- action(A), time(T), T<length, possible(A,T), not nocc(A,T).

holds(has(self,prev_action,A),T+1) :- occ(A,T).

:- time(T), occ(null,T).
:- time(T), fluent(F), holds(F, T), contrary(F,G), holds(G, T).

action(null).
possible(null, T):- time(T).

%-------------------------------------------------------------
% Inertia
%-------------------------------------------------------------
```

```
% Check for any instance of a contrary
found_contrary(L,T+1) :- contrary(L,G), holds(L,T), holds(G,T+1), time(T), T<length.
holds(L,T+1) :- holds(L,T), not found_contrary(L,T+1), time(T), T<length.

%-----------------------------------------------------------------
% auxiliary axioms
%-----------------------------------------------------------------

literal(G) :- fluent(G).
literal(neg(G)) :- fluent(G).

contrary(F, neg(F)) :- fluent(F).
contrary(neg(F), F) :- fluent(F).

contrary(has(self,orient,D1), has(self,orient,D2)) :- is(D1,cardinal), is(D2,cardinal),
                D1!=D2.
contrary(pos(self,X1,Y1), pos(self,X2,Y2)) :- X1!=X2, position(X1,Y1), position(X2,Y2).
contrary(pos(self,X1,Y1), pos(self,X2,Y2)) :- Y1!=Y2, position(X1,Y1), position(X2,Y2).
contrary(has(O,location,D1), has(O,location,D2)) :- is(O,object), is(D1,direction),
                is(D2,direction), D1!=D2.
contrary(has(self,prev_action,A1), has(self,prev_action,A2)) :- action(A1), action(A2),
                A1!=A2.

leq(T,T) :- time(T).
leq(T1,T2) :- time(T1), time(T2), T1 < T2.

holds(L,0) :- literal(L), initially(L).

%-----------------------------------------------------------------
% Modeling Environment
%-----------------------------------------------------------------

is(north,cardinal).
is(east,cardinal).
is(south,cardinal).
is(west,cardinal).

is(right,direction).
is(left,direction).
is(front,direction).
is(back,direction).

is(healthy,status).
is(injured,status).
is(dead,status).
is(conscious,status).
is(unconscious,status).
```

```
is(X,object) :- is(X,person).
is(X,object) :- is(X,door).
is(X,object) :- is(X,medical_supplies).
is(X,object) :- is(X,hazmat_sensor).

is(P,place) :- is(P,hall).
is(P,place) :- is(P,room).
is(P,place) :- is(P,door).

is(hs,hazmat_sensor).
is(ms,medical_supplies).
is(cmdrX,person).
is(cmdrZ,person).

is(d1,door).
is(d2,door).
is(d3,door).
is(d4,door).
is(d5,door).
is(d6,door).
is(d7,door).
is(d8,door).

is(h1,hall).
is(h2,hall).
is(h3,hall).
is(h4,hall).
is(h5,hall).
is(h6,hall).
is(h7,hall).
is(h8,hall).
is(h9,hall).
is(h10,hall).
is(h11,hall).

is(r5,room).
is(r7,room).
is(r8,room).
is(r9,room).
is(r10,room).
is(r11,room).
is(r12,room).
is(r13,room).

is(hall1,hallway).

has(hall1,hall_segment,h1).
```

```
has(hall1,hall_segment,h2).
has(hall1,hall_segment,h3).
has(hall1,hall_segment,h4).
has(hall1,hall_segment,h5).
has(hall1,hall_segment,h6).
has(hall1,hall_segment,h7).
has(hall1,hall_segment,h8).
has(hall1,hall_segment,h9).
has(hall1,hall_segment,h10).
has(hall1,hall_segment,h11).
end(hall1, h1).
end(hall1, h11).

connected(h1, h2, east).
connected(h2, h3, east).
connected(h3, h4, east).
connected(h4, h5, east).
connected(h5, h6, east).
connected(h6, h7, east).
connected(h7, h8, east).
connected(h8, h9, east).
connected(h9, h10, east).
connected(h10, h11, east).
connected(h3, d1, north).
connected(h5, d2, north).
connected(h6, d3, north).
connected(h7, d4, north).
connected(h8, d5, north).
connected(h9, d6, north).
connected(h10, d7, north).
connected(h11, d8, north).
connected(d1, r5, north).
connected(d2, r7, north).
connected(d3, r8, north).
connected(d4, r9, north).
connected(d5, r10, north).
connected(d6, r11, north).
connected(d7, r12, north).
connected(d8, r13, north).

position(0,0).
position(1,0).
position(2,0).
position(3,0).
position(4,0).
position(5,0).
position(6,0).
```

```
position(7,0).
position(8,0).
position(9,0).
position(10,0).
position(2,1).
position(4,1).
position(5,1).
position(6,1).
position(7,1).
position(8,1).
position(9,1).
position(10,1).
position(2,2).
position(4,2).
position(5,2).
position(6,2).
position(7,2).
position(8,2).
position(9,2).
position(10,2).

pos(h1,0,0).
pos(h2,1,0).
pos(h3,2,0).
pos(h4,3,0).
pos(h5,4,0).
pos(h6,5,0).
pos(h7,6,0).
pos(h8,7,0).
pos(h9,8,0).
pos(h10,9,0).
pos(h11,10,0).
pos(r5,2,2).
pos(r7,4,2).
pos(r8,5,2).
pos(r9,6,2).
pos(r10,7,2).
pos(r11,8,2).
pos(r12,9,2).
pos(r13,10,2).

pos(d1,2,1).
pos(d2,4,1).
pos(d3,5,1).
pos(d4,6,1).
pos(d5,7,1).
pos(d6,8,1).
```

```
pos(d7,9,1).
pos(d8,10,1).

holds(pos(O,X,Y),T) :- pos(O,X,Y), time(T), is(O,object).

%----------------------------------------------------------------------
% Output
%----------------------------------------------------------------------


plan(T,A,X,Y,D) :- occ(A,T), holds(pos(self,X,Y),T+1), holds(has(self,orient,D),T+1),
                   action(A).
plan(T,sensed,P,S) :- occ(A,T), action(A), A=sense(status(P)), is(P,person),
                   holds(has(self,sensed,status(P,S)),T+1).

%----------------------------------------------------------------------
% Hiding/Showing Answerset
%----------------------------------------------------------------------
#hide.
#show plan/5.
#show plan/4.
```

APPENDIX B

ARCHIMPL.JAVA - AN IMPLEMENTATION OF ADE

```
// Robot Interaction Demo
// Template (c) Matthias Scheutz
// Barry Lumpkin

import com.action.ActionServerImpl;
import ade.exceptions.ADECallException;
import java.rmi.*;
import java.util.HashMap;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.InputStreamReader;
import java.io.Writer;
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.lang.Math;
import java.util.ArrayList;
import javax.swing.JTextField;
import javax.swing.JTextArea;
import javax.swing.JScrollPane;
import javax.swing.JDialog;
import javax.swing.JOptionPane;
import javax.swing.JFrame;
import java.beans.*;
import java.awt.*;
import java.awt.event.*;

public class ArchImpl extends ActionServerImpl implements Arch {
private static final long serialVersionUID = 1L;

final double UNIT = 1.37175; // This is the distance of one “hallway UNIT”

int step = 0;
HashMap<Integer,PlanStep> plan;
HashMap<Integer,SenseStep> sensing;
HashMap<Integer,Double> mapX;
HashMap<Integer,Double> mapY;

String state = “”;
double rotVel = 0;
double transVel = 0;
double goalAngle = 0;
double orientDirection = 0;
```

```
double goalDist = -1;
boolean verbose = false;
boolean doneOrienting = false;
boolean initialized = false;

JFrame frame;
CustomDialog dialog;

/**
* <code>runArchitecture</code> is called periodically to perform
* whatever sensing and acting is required by the architecture.
*/
public void runArchitecture()
{
if(!initialized)
{
frame = new JFrame("Dialog Handler");
dialog = new CustomDialog(frame, this);
frame.setSize(800, 400);
frame.setVisible(false);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
dialog.setVisible(true);
dialog.setSize(800, 800);

readMap();
state = "stop";
initialized = true;
}

// Obstacles
// blocking front 0.4 @ 90 deg
boolean obstacleFront = false;

// get perceptions
double[] laser = getLaserReadings();
double orient = getOrientation();
double[] position = getPos();

if(laser[90] < 0.4)
obstacleFront = true;

if(state.equals("")) // DEBUG State
{
if(verbose)
System.out.println(laser[90]);
}
else if(state.equals("turn_right") || state.equals("turn_left"))
```

```
{
// Turn to orient to given angle
// Use angle specified in next step


orientDirection = plan.get(step).orient;
goalAngle = Math.min(Math.abs(orientDirection - orient), 360 -
                Math.abs(orientDirection - orient));
rotVel = ((Math.sin(goalAngle / 180.0 * Math.PI/2)) * 2);

if(goalAngle != 0 && rotVel < 0.02)
rotVel = 0.02;

if(goalAngle == Math.abs(orientDirection - orient))
rotVel *= Math.signum(orientDirection - orient); // negative = Turning right
else
rotVel *= -1 * Math.signum(orientDirection - orient); // negative = Turning right

if(goalAngle <= 1)
{
if(doneOrienting)
{
rotVel = 0;
step++; // Done Orienting, next step
if(step < plan.size())
state = plan.get(step).action;
else
{
dialog.displayResponse("Done.");
state = "stop";
}
}
// Satisfy goalAngle <= 1 two consecutive times to prevent match from overshooting
doneOrienting = true;
if(verbose)
System.out.println("*Orienting*");
}
else
doneOrienting = false;
}
else if(state.equals("go_straight_one_step"))
{
// Move forward to given destination
// Use goal location specified in plan
// Obstacle = stop and ask for assistance

// If going in a straight line, find the endpoint and go there. Don't stop at each "point" on
the path
```

```
while(step+1 < plan.size() && plan.get(step+1).action.equals("go_straight_one_step"))
step++;


double goalX = plan.get(step).posX;
double goalY = plan.get(step).posY;
goalDist = Math.sqrt( Math.pow((goalX-position[0]),2) +
            Math.pow((goalY-position[1]),2));

if(obstacleFront && goalDist <= 0.1) // Close enough. Don't hit the obstacle
{
transVel = 0;
rotVel = 0;
step++;
if(step < plan.size())
state = plan.get(step).action;
else
{
dialog.displayResponse("Done.");
state = "stop";
}
}
else if(obstacleFront && laser[90] - goalDist > 0.1) // Obstacle & not close enough to goal
{
transVel = 0;
rotVel = 0;
state = "stop";
//NOTE: This is displaying clarify(A) where A is an action, not clarify(phi) where phi is a
goal.
dialog.displayResponse("clarify(go_straight_one_step).");
}
else
{
transVel = .7; // Default speed
if(goalDist < UNIT/2) // Slow down when approaching walls/obstacles
transVel = goalDist/3; // Start at .165 then slow down linearly w.r.t. wall distance
}

// Keep aligned towards goal
orientDirection = Math.toDegrees(Math.atan2((goalY-position[1]),(goalX-position[0])));
if(orientDirection<0)
orientDirection = 360+orientDirection;

if(verbose)
System.out.println("X: " + goalX + " Y: " + goalY + " orient: " + orientDirection);


goalAngle = Math.min(Math.abs(orientDirection - orient), 360 -
```

```
                Math.abs(orientDirection - orient));
rotVel = ((Math.sin(goalAngle / 180.0 * Math.PI/2)) * 2);

if(goalAngle != 0 && rotVel < 0.01)
rotVel = 0.01;

if(goalAngle == Math.abs(orientDirection - orient))
rotVel *= Math.signum(orientDirection - orient); // negative = Turning right
else
rotVel *= -1 * Math.signum(orientDirection - orient); // negative = Turning right

if(verbose)
System.out.println("Dist to stop: " + goalDist);

if(goalDist < 0.05) // Close enough to goal
{
transVel = 0;
rotVel = 0;
goalDist = -1;
step++;
if(step < plan.size())
state = plan.get(step).action;
else
{
dialog.displayResponse("Done.");
state = "stop";
}
}
}
else if(state.contains("pick_up"))
{
//NOTE: Doesn't pick anything up. The robot architecture does not support picking up
blocks
transVel = 0;
rotVel = 0;
step++;
if(step < plan.size())
state = plan.get(step).action;
else
{
dialog.displayResponse("Done.");
state = "stop";
}
}
else if(state.contains("put_down"))
{
//NOTE: Doesn't put anything down. The robot architecture does not support placing
```

```
blocks
transVel = 0;
rotVel = 0;
step++;
if(step < plan.size())
state = plan.get(step).action;
else
{
dialog.displayResponse(“Done.”);
state = “stop”;
}
}
else if(state.contains(“sense”))
{
//NOTE: The robot itself does not actually sense. The robot architecture does not support
this.
//Values obtained from the planner.
transVel = 0;
rotVel = 0;

SenseStep sensed = sensing.get(step);
if(sensed == null)
{
dialog.displayResponse(“SENSING ERROR.”);
}
else
{
for(String str : sensed.status)
dialog.displayResponse(str);
}

step++;
if(step < plan.size())
state = plan.get(step).action;
else
{
dialog.displayResponse(“Done.”);
state = “stop”;
}
}
else if(state.equals(“stop”))
{
transVel = 0;
rotVel = 0;
}
else
{
```

```
dialog.displayResponse(“clarify(” + state + “).”);
}

setVels(transVel,rotVel);
return;
}

// Read output from clingo and translate into a sequence of action steps
public void readPlan()
{
try
{
String str;
state = “stop”;
plan = new HashMap<Integer,PlanStep>();
sensing = new HashMap<Integer,SenseStep>();


Runtime rt = Runtime.getRuntime();
Process pr = rt.exec(“clingo tempInitState.lp demoPlanner.lp 1”);
BufferedReader br = new BufferedReader(new
                        InputStreamReader(pr.getInputStream()));

while((str = br.readLine()) != null) {
if(str.contains(“plan(”))
{
String plans[] = str.split(“
) plan
(”);
for(int i=0; i<plans.length; i++)
{
plans[i]=plans[i].replace(“ ”, “”);
plans[i]=plans[i].replace(“plan(”, “”);
plans[i]=plans[i].replace(“)”, “”);

//0,go_straight_one_step,5,0,west
String[] planInfo = plans[i].split(“,”);
//0
//go_straight_one_step
//5
//0
//west
if(planInfo[1].equals(“sensed”))
{
SenseStep ss = sensing.get(Integer.parseInt(planInfo[0]));
if(ss == null)
{
```

```
ss = new SenseStep();
ss.timestep = Integer.parseInt(planInfo[0]);
ss.object = planInfo[2];
ss.status = new ArrayList<String>();
}
ss.status.add(planInfo[3]);

sensing.put(ss.timestep, ss);
}
else
{
PlanStep planStep = new PlanStep();
planStep.timestep = Integer.parseInt(planInfo[0]);
planStep.action = planInfo[1];
planStep.posX = mapX.get(Integer.parseInt(planInfo[2]));
planStep.posY = mapY.get(Integer.parseInt(planInfo[3]));
planStep.orient = convertOrient(planInfo[4]);

plan.put(planStep.timestep, planStep);
}
}
}
}

if(plan.size() > 0)
{
step = 0;
state = plan.get(step).action;
dialog.displayResponse("Acknowledged.");
}
else
{
step = -1;
state = "stop";
dialog.displayResponse("Failed. No plan could be generated");
}
br.close();
}catch (Exception e){
System.err.println("Error: " + e.getMessage());
}
}

// Read from map.txt and translate the ASP output into a sequence of action steps
void readMap()
{
try
{
```

```
BufferedReader br = new BufferedReader(new FileReader("demoMap.txt"));

String str;
mapX = new HashMap<Integer,Double>();
mapY = new HashMap<Integer,Double>();

while((str = br.readLine()) != null) {
if(str.contains(":"))
{
//X:0=-14.6
String coordInfo[] = str.split(":");

//X
//0=-14.6
String coords[] = coordInfo[1].split("=");

if(coordInfo[0].equals("X"))
mapX.put(Integer.parseInt(coords[0]), Double.parseDouble(coords[1]));
else
mapY.put(Integer.parseInt(coords[0]), Double.parseDouble(coords[1]));
}
}
br.close();
}catch (Exception e){
System.err.println("Error: " + e.getMessage());
}
}

/**
* Get the absolute position of the agent.
* @return the global x, y, t position
*/
public double[] getPos()// throws RemoteException
{
double[] position = null;

if (!checkMethod("getPoseGlobal")) {
return null;
}
try {
position = (double[])callMethod("getPoseGlobal", new Object[0]);
current_x = position[0];
current_y = position[1];
current_t = position[2];
} catch (ADECallException ace) {
System.out.println("getPoseGlobal: error checking position");
}
```

```
return position;
}

public void newCommand(String str)
{
dialog.displayResponse(“Processing”);
if(verbose)
System.out.println(“New Command: ” + str);

try {
Writer output = new BufferedWriter(new FileWriter(“tempInitState.lp”));
output.write(str);
output.close();
}
catch(IOException e) {
e.printStackTrace();
}

readPlan();
}

public int convertOrient(String str)
{
int orient = 0;

if(str.equals(“east”))
orient = 0;
if(str.equals(“north”))
orient = 90;
if(str.equals(“west”))
orient = 180;
if(str.equals(“south”))
orient = 270;

return orient;
}

/**
* Constructs the architecture
*/
public ArchImpl() throws RemoteException {
super();
}

//————————————————————————————————————————
// Plan Storage
```

```
//————————————————————————————————————————————————————————————

private class PlanStep
{
public int timestep; //6
public String action; //go_straight_one_step
public double posX, posY; //1, 2
public int orient; //90
}

private class SenseStep
{
public int timestep; //6
public String object; //cmdrX
public ArrayList<String> status; //injured
}

//————————————————————————————————————————————————————————————
// Dialogue Window
//————————————————————————————————————————————————————————————
private class CustomDialog extends JDialog
implements ActionListener,
PropertyChangeListener {
private String typedText = null;
private JTextArea textField;
private JTextArea responseField;

private JScrollPane textPane;
private JScrollPane responsePane;

private JOptionPane optionPane;

private String btnString1 = “Enter”;

private ArchImpl archImpl;

/**
* Returns null if the typed string was invalid;
* otherwise, returns the string as the user entered it.
*/
public String getValidatedText() {
return typedText;
}

/** Creates the reusable dialog. */
public CustomDialog(Frame aFrame, ArchImpl arch) {
super(aFrame, true);
```

```
setModal(false);
archImpl = arch;

setTitle("Dialog Window");

//Create textArea to enter commands
textField = new JTextArea(10, 60);
textPane = new JScrollPane(textField);
textField.setEditable(true);

//Create textArea to display responses
responseField = new JTextArea(10, 60);
responsePane = new JScrollPane(responseField);
responseField.setEditable(false);
Color color=new Color(220,220,220);
responseField.setBackground(color);

//Create an array of the text and components to be displayed.
String msgString1 = "Please Enter a Command";
Object[] array = {msgString1, textPane, responsePane};

//Create an array specifying the number of dialog buttons
//and their text.
Object[] options = {btnString1};

//Create the JOptionPane.
optionPane = new JOptionPane(array,
JOptionPane.QUESTION_MESSAGE,
JOptionPane.YES_NO_OPTION,
null,
options,
options[0]);

//Make this dialog display it.
setContentPane(optionPane);

//Handle window closing correctly.
setDefaultCloseOperation(DO_NOTHING_ON_CLOSE);
addWindowListener(new WindowAdapter() {
public void windowClosing(WindowEvent we) {
/*
* Instead of directly closing the window,
* we're going to change the JOptionPane's
* value property.
*/
optionPane.setValue(new Integer(
JOptionPane.CLOSED_OPTION));
```

```
}
});

//Ensure the text field always gets the first focus.
addComponentListener(new ComponentAdapter() {
public void componentShown(ComponentEvent ce) {
textField.requestFocusInWindow();
}
});

//Register an event handler that puts the text into the option pane.
// textField.addActionListener(this);

//Register an event handler that reacts to option pane state changes.
optionPane.addPropertyChangeListener(this);
}

/** This method handles events for the text field. */
public void actionPerformed(ActionEvent e) {
optionPane.setValue(btnString1);
}

/** This method reacts to state changes in the option pane. */
public void propertyChange(PropertyChangeEvent e) {
String prop = e.getPropertyName();

if (isVisible()
&& (e.getSource() == optionPane)
&& (JOptionPane.VALUE_PROPERTY.equals(prop) ||
JOptionPane.INPUT_VALUE_PROPERTY.equals(prop))) {
Object value = optionPane.getValue();

if (value == JOptionPane.UNINITIALIZED_VALUE) {
//ignore reset
return;
}

//Reset the JOptionPane's value.
//If you don't do this, then if the user
//presses the same button next time, no
//property change event will be fired.
optionPane.setValue(
JOptionPane.UNINITIALIZED_VALUE);

if (btnString1.equals(value)) {
typedText = textField.getText();
String ucText = typedText.toUpperCase();
```

```
if (ucText.equals(“QUIT”)) {
//we’re done; clear and dismiss the dialog
clearAndHide();
} else {
archImpl.newCommand(typedText);
}
} else { //user closed dialog or clicked cancel
typedText = null;
clearAndHide();
}
}
}

public void displayResponse(String str)
{
responseField.setText(responseField.getText() + str + “
n”);
}

/** This method clears the dialog and hides it. */
public void clearAndHide() {
textField.setText(null);
setVisible(false);
}
}
}
```

APPENDIX C

SAMPLE DIALOG IN ASP - PART 1

```
#const length=7.

% Initial state
initially(pos(self,6,0)).
initially(has(self,orient,west)).
initially(has(self,carrying,hs)).
initially(pos(cmdrX,2,2)).
initially(has(cmdrX,status,healthy)).
initially(has(cmdrX,status,conscious)).
initially(pos(ms,4,2)).
initially(pos(cmdrZ,6,2)).
initially(has(cmdrZ,status,injured)).
initially(has(cmdrZ,status,conscious)).

goal(s1,T):- time(T), trans(s1, 0, T).
goal(s2,T2):- time(T1), time(T2), trans(s2, T1, T2), goal(s1,T1), T1<=T2.
goal(T):- time(T), goal(s2,T).
goal(T+1):- time(T), T < length, goal(s2,T).
:- not goal(length).

proc(s1).
head(s1, find_door1_right).
tail(s1, s1b).

proc(s1b).
head(s1b, go_through_d1).
tail(s1b, null).

proc(s2).
head(s2, go_to_cmdrX).
tail(s2, null).

% while -has(d1,location,right)
% do go_straight_one_step.
while(find_door1_right,neg(has(d1,location,right)),go_straight_one_step).

% goal(self,<>(has(self,at,d1) ∧ ()(has(self,prev_action,go_straight_one_step)))).
achieve(go_through_d1,achieve_through_d1).
tf(achieve_through_d1,eventually(go_through_d1_steps)).
conj(go_through_d1_steps).
in(has(self,at,d1), go_through_d1_steps).
in(next_go_straight, go_through_d1_steps).
tf(next_go_straight,next(has(self,prev_action,go_straight_one_step))).
```

```
tformula(achieve_through_d1).
tformula(next_go_straight).

% goal(self,<>has(self,at,cmdrX)).
achieve(go_to_cmdrX,achieve_cmdrX).
tf(achieve_cmdrX,eventually(has(self,at,cmdrX))).
tformula(achieve_cmdrX).
```

APPENDIX D

SAMPLE DIALOG IN ASP - PART 2

```
#const length=11.

% Initial state
initially(pos(self,2,2)).
initially(has(self,orient,north)).
initially(has(self,carrying,hs)).
initially(pos(cmdrX,2,2)).
initially(has(cmdrX,status,healthy)).
initially(has(cmdrX,status,conscious)).
initially(pos(ms,4,2)).
initially(pos(cmdrZ,6,2)).
initially(has(cmdrZ,status,injured)).
initially(has(cmdrZ,status,conscious)).

goal(s1,T):- time(T), trans(s1, 0, T).
goal(s2,T2):- time(T1), time(T2), trans(s2, T1, T2), goal(s1,T1), T1<=T2.
goal(T):- time(T), goal(s2,T).
goal(T+1):- time(T), T < length, goal(s2,T).
:- not goal(length).

proc(s1).
head(s1, go_to_ms).
tail(s1, null).

proc(s2).
head(s2, pick_up(ms)). % Note, this is an action
tail(s2, null).

% goal(self,<>has(self,at,ms)).
achieve(go_to_ms,achieve_ms).
tf(achieve_ms,eventually(has(self,at,ms))).
tformula(achieve_ms).
```

APPENDIX E

SAMPLE DIALOG IN ASP - PART 3

```
#const length=12.

% Initial state
initially(pos(self,4,2)).
initially(has(self,orient,north)).
initially(has(self,carrying,hs)).
initially(has(self,carrying,ms)).
initially(pos(cmdrX,2,2)).
initially(has(cmdrX,status,healthy)).
initially(has(cmdrX,status,conscious)).
initially(pos(cmdrZ,6,2)).
initially(has(cmdrZ,status,injured)).
initially(has(cmdrZ,status,conscious)).

goal(s1,T):- time(T), trans(s1, 0, T).
goal(s2,T2):- time(T1), time(T2), trans(s2, T1, T2), goal(s1,T1), T1<=T2.
goal(s3,T2):- time(T1), time(T2), trans(s3, T1, T2), goal(s2,T1), T1<=T2.
goal(T):- time(T), goal(s3,T).
goal(T+1):- time(T), T < length, goal(s3,T).
:- not goal(length).

proc(s1).
head(s1, go_to_cmdrZ).
tail(s1, null).

% sense(Y): status(cmdrZ,Y), person(cmdrZ).
proc(s2).
head(s2, sense(status(cmdrZ))).
tail(s2, null).

proc(s3).
head(s3, if1).
tail(s3, null).

% goal(self,<>has(self,at,cmdrZ)).
achieve(go_to_cmdrZ,achieve_cmdrZ).
tf(achieve_cmdrZ,eventually(has(self,at,cmdrZ))).
tformula(achieve_cmdrZ).

% If the person is awake but injured, give them medical supplies
% if has(cmdrZ,status,injured) ∧ has(cmdrZ,status,conscious)
% then pick(M, put_down(M)) : has(self,carrying,M) ∧ is(M,medical_supplies).
if(if1,awake_injured,place_meds,null).
```

```
conj(awake_injured).
in(has(self,sensed,status(cmdrZ,injured)), awake_injured).
in(has(self,sensed,status(cmdrZ,conscious)), awake_injured).

% pick medical_supplies to put down
% (see if statement above for RIL-D translation)
choiceArgs(place_meds,has(self,carrying,M),put_down(M)) :- is(M,medical_supplies).
```