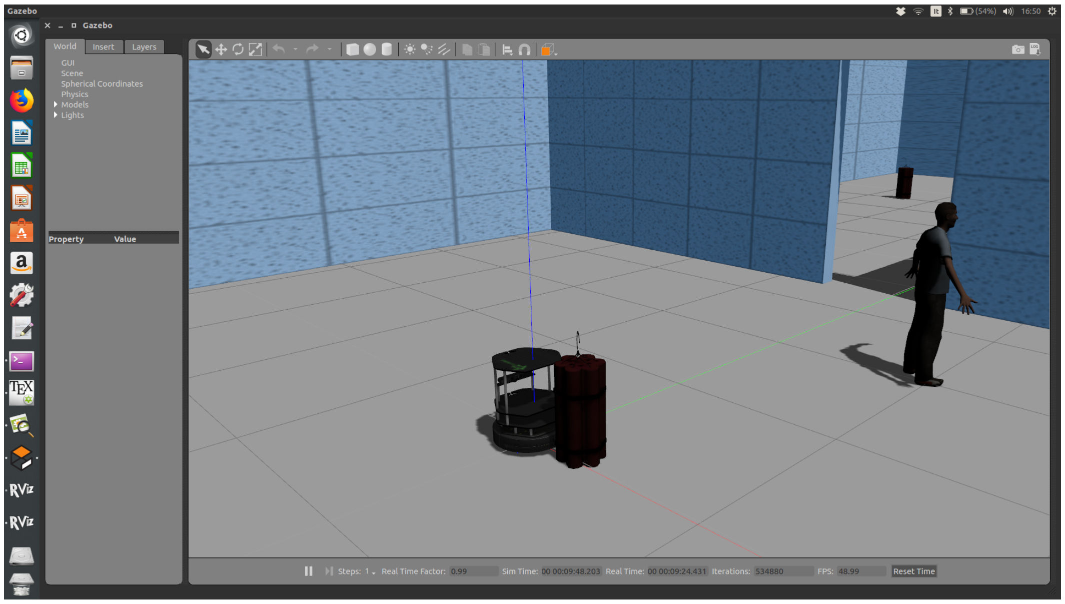
HRI - Reasoning Robots
Course Project 2017/2018

Robot Control through Natural Language



by Gianluca Massimiani

November 24, 2018

Contents

1 Project Outline

As human beings, the ability to reason at different levels of abstraction is one of our greatest skills. Natural language is the tool that allows us to communicate our reasoning, no matter how complicated or abstract it may be. The power of natural language indeed resides in its high expressiveness, although there is some price to pay. In fact, natural language can often be ambiguous. Knowing the meaning of each single word in a sentence may not be enough to grasp the meaning of the sentence as a whole. There are cases when the same sentence may have different meanings depending on the context in which it is told.

Differently from humans, the ability to reason at abstract levels is something that most of the robots are still missing. A robot is usually able to carry out tasks by executing a finite set of low level actions (e.g. move forward, turn left/right, etc.). Robot control is thus achieved at this very low level. The problem of controlling a robot with natural language is quite challenging because when we command a robot using language, our orders are usually at a higher level than what the robot can actually do. To get the robot to understand and execute our commands, we need to create a controller that takes our high level commands and refines them into lower level ones, down to a point where the robot can execute them. The goal of this project is to create such a controller. To achieve this, it is necessary to bridge the gap between the high expressiveness (and abstractness) of natural language and the very low level at which robots usually operate.

2 Problem Modelling

The aim of the project is to develop a system that allows a user to control a robot using natural language commands. The problem is challenging, because natural language is extremely powerful and expressive. For example, some natural language commands might be: "*Bring me some food*", "*Unload the truck*", "*Turn the music volume down*". Depending on the context, these are all valid commands. However, it's really hard to design a robot that is able to carry out such a diverse set of orders. For this reason, we restrict our problem by focusing only on one robot scenario and we make a few assumptions to simplify our problem, although without affecting the validity of our approach.

We consider a simple robot, such as the Turtlebot shown in Fig. 1. This robot is designed to work in a particular *search and rescue* scenario, which can be represented as an environment made of:

- a building with rooms (regions) connected with doors and separated by walls
- human hostages
- bombs

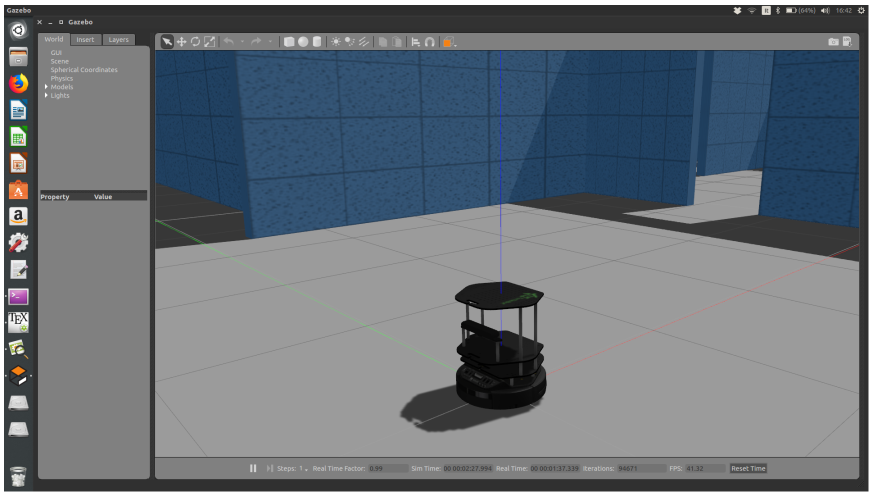


Figure 1: The system performance is evaluated in a simulated environment using a Turtlebot.

Within this environment, the robot is able to carry out the following high level tasks:

- basic navigation tasks, such as moving between rooms
- search and rescue tasks, such as detecting hostages and interacting with them, as well as detecting and defusing bombs.

As long as the natural language commands involve one of the above robot tasks, the robot must be able to execute them. Commands which are not related to the above scenario will not be executed nor understood by the robot. To assess the overall performance of the system, we will simulate the robot behavior using a popular simulation platform. The user will be able to give written natural language commands to the Turtlebot, and watch them being carried out on the computer screen. Further details on these simulations (and links to recorded videos) are provided later on in this report (section 4).

At the present state, the system supports written natural language commands only expressed in the English language. An interesting (and challenging) extension would be to allow for support of natural language utterances, i.e. vocal commands. The choice of using a simulation platform, rather than implementing the system on a real robot, was forced by a lack of hardware resources (including the robot itself). However, an advantage of using a simulation platform is that one can test the system without the need of particuar testing environment nor hardware requirement, other than a mid-level computer. The following section describes the overall system architecture.

3 System Architecture

This project is inspired by the works in [1], [2] in which a method to achieve robot control from natural language specification is proposed. The approach follows these steps:

- *Syntactic Parsing.* Natural language sentences are parsed and a hierarchical structure is assigned to them. This structure is called *parse tree*, and it usually depends on the particular language and its grammar. By recovering the syntactic structure of a natural language sentence, we are able to handle particular language phenomena such as negation (e.g. "***Don't** go to the hall*") and coordination (e.g. "*Go to the bedroom **and** the kitchen*").
- *Semantic interpretation.* This step consists in using the result of the syntactic parsing (parse tree), to extract verbs and their arguments (subject and object). From the recognized verbs and arguments, we can create a data structure that encodes the extracted semantic information of the natural language sentence.
- *Translation into Linear Temporal Logic (LTL).* Once the semantics has been extracted from the sentence, we can use this information to generate logical formulas defining the low level robot tasks. In this case Linear Temporal Logic is used, which is a modal logic that includes temporal operators. This allows us to generate formulas that specify the truth values of atomic propositions over time.
- *Synthesis of an automaton.* The LTL formulas generated in the previous step are combined together to form an LTL specification, from which a finite state machine is then created. This machine, also called automaton, will drive the robot during the execution of its tasks.
- *Continuous controller.* The automaton is used to generate the desired robot behavior. More specifically, the automaton can be seen as a discrete controller for the robot, specifying which actions must be executed at each state. By following the automaton the robot will be able to satisfy the user specification. However, the actions specified by the automaton are discretized (i.e. high level), so there is the need to fill the gaps by generating lower level actions that allow to implement the high level actions.

The system implemented for this project reflects upon the previous four steps. The main components in the overall system architecture are therefore: a syntactic parser, a semantic parser, an LTL formula generator, and an automaton synthesizer. These components are described more in detail in the following paragraphs.

3.1 NLP Pipeline

The natural language processing pipeline consists of a syntactic parser and a semantic parser. Given a user specification, which normally consists of several sentences expressed in natural language, the NLP pipeline will process each sentence, first by applying the syntactic parser and then passing the result of the syntactic analysis (parse tree) to the semantic parser. The latter then proceeds to extract the meaning from the parse tree and encode it into an appropriate data structure. Currently the pipeline is able to process only sentences written in the English language.

3.1.1 Syntactic parser

The syntactic parser decomposes a natural language sentence into atomic components and arrange them in a hierarchical structure called *parse tree* according to the language grammar. For example, given the input sentence "*If you see a bomb, defuse it*" the syntactic parser produces the parse tree shown in Figure 2. The syntactic parser used for this project is the Bikel parser [3] provided (open source) by the University of Pennsylvania [1]. As shown in Figure 3, this parser is among the top parsers, although more recent ones (e.g. Charniak and Berkeley parsers) achieve higher accuracy and thus represent the state-of-the-art in syntactic parsing. The Bikel parser is combined with a null element restoration [5] to parse sentences. This allows to recover null elements in sentences, which are those silent subjects and objects that appear in structures such as imperatives and relative clauses. For example, in a sentence such as "*Go to r1*", where *r1* is a location's name, there is a silent subject *you* which is the subject of the *go* verb. The parse tree produced by the syntactic parser is passed to the semantic parser, and null element restoration allows to ease the semantic interpretation. However, for the purpose of this project null element restoration itself is not enough. To facilitate the work of the semantic parser, a coordinate structure handling feature was implemented. After the parse tree has been produced by the syntactic parser, coordinate structures are identified (e.g. by finding *and* conjunctions), and the parse tree is properly split. In this way, sentences that contain conjunctions are split such that they are equivalent to multiple full clauses. For example, a sentence like "*Go to r3 and r4*" becomes "*[You] Go to r3*" *AND* "*[You] Go to r4*". This results in two separates parse trees, as shown in Figure 4.

3.1.2 Semantic Parser

The semantic parser traverses the parse tree identifying verbs and their arguments, and then creates a data structure which encodes the extracted semantic information. For example, in the sentence "*If you see a bomb, defuse it*", the desired structure is a *defuse* command with an object *bomb*, subject to the condition *see(bomb).* This data structure is represented in Figure 5a. Every natural language command is translated into a similar data structure, which encodes the semantics of the command. The command "*Don't go to r4*", for example, is translated into the structure shown in Figure 5b. However, natural language is highly flexible and it allows to convey the same meaning in many different ways.

[1] https://github.com/PennNLP/

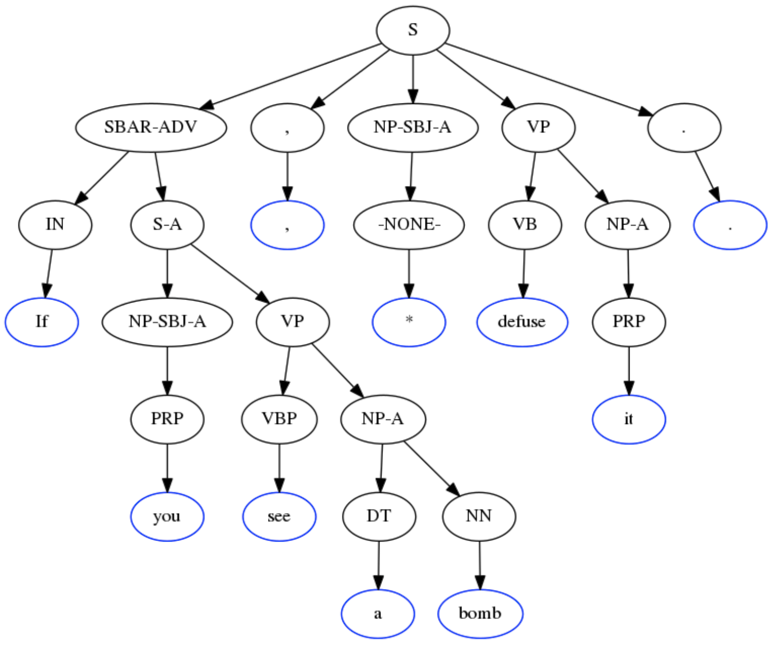
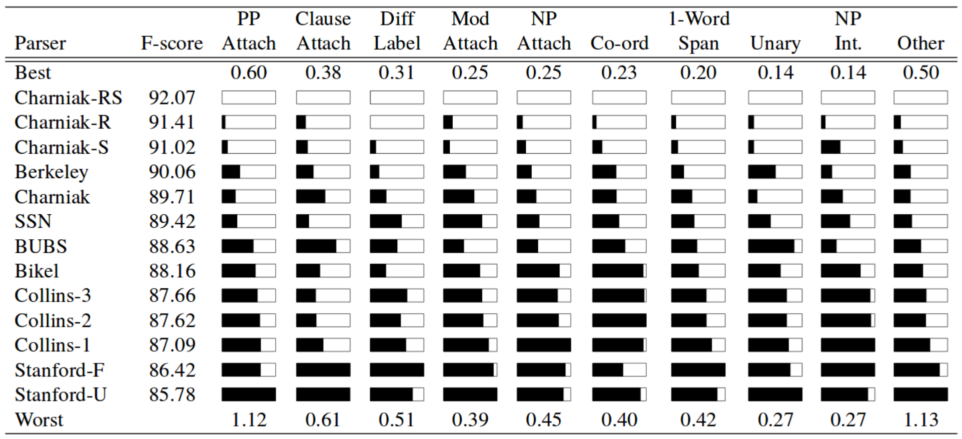


Figure 2: Example of parse tree produced by the syntactic parser.



Parser	F-score	PP Attach	Clause Attach	Diff Label	Mod Attach	NP Attach	Co-ord	1-Word Span	Unary	NP Int.	Other
Best		0.60	0.38	0.31	0.25	0.25	0.23	0.20	0.14	0.14	0.50
Charniak-RS	92.07										
Charniak-R	91.41										
Charniak-S	91.02										
Berkeley	90.06										
Charniak	89.71										
SSN	89.42										
BUBS	88.63										
Bikel	88.16										
Collins-3	87.66										
Collins-2	87.62										
Collins-1	87.09										
Stanford-F	86.42										
Stanford-U	85.78										
Worst		1.12	0.61	0.51	0.39	0.45	0.40	0.42	0.27	0.27	1.13

Figure 3: Top syntactic parsers (listed by rows) by average number of errors per sentence due to top ten error types (listed by column). The scale for each column is indicated by the best and worst value. F-score is a measure of the overall parser accuracy. Taken from [4].

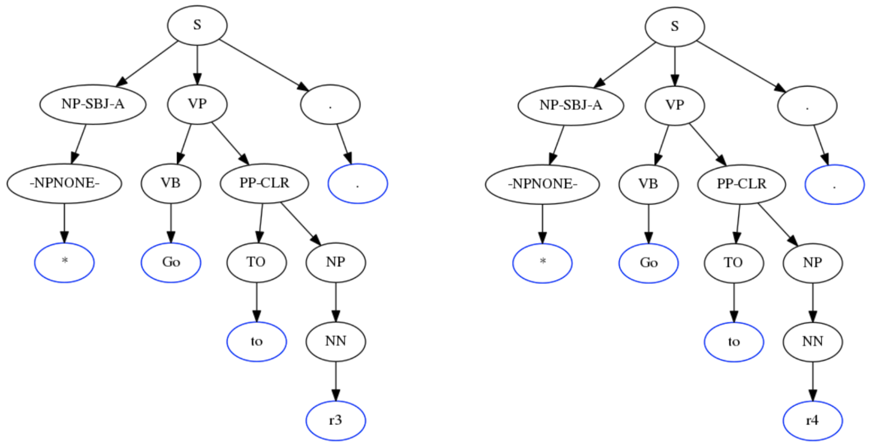


Figure 4: The parse tree of a sentence containing conjunctions is splitted, and separate parse trees are created. For example, the command "*Go to r3 and r4*" results in two separate parse trees "*[You] Go to r3*" and "*[You] Go to r4*".

The commands "*Don't go to r4*" and "*Avoid r4*" for example are (semantically) equivalent, therefore they should both be translated to the same structure of Figure 5b. This will give the user more freedom when expressing the commands in natural language. To allow this flexibility, the semantic parser implemented for this project uses a database of verbs (and the arguments that they can take), called *VerbNet* [2]. The database groups together verbs that belong to the same senses, i.e. verbs that have similar meanings in similar contexts. For example, the verbs *see, detect*, and *perceive* all belong to the general sense *SEE*, because they generally share the same meaning (that of seeing something). Therefore, whether the user writes "*If you see a bomb...*" or "*If you detect a bomb...*" or "*If you perceive a bomb...*" doesn't make any difference for the system, because all these sentences are mapped to the same command shown in Figure 5a. For each sense, *VerbNet* provides a set of frames which indicates the possible arguments to the sense. For example, a frame for the sense *SEE* is made of three fields: [*AGENT*, *VERB*, *THEME*]. This corresponds to a simple sentence like "*I saw a bomb.*". A more complex sentence such as "*If you see a bomb, defuse it*" is also represented by the frame [*AGENT*, *VERB*, *THEME*], but in this case the field *THEME* has a subfield *CONDITION*, as shown in Figure 5a. For a given sense, *VerbNet* does not provide only one frame, but rather many, because a verb can be used in different syntactic structures, i.e. it can take many different arguments. The semantic parser must choose the frame that most resemble the natural language sentence. This is done comparing each frame against the parse tree. In fact, the best frame is the one that perfectly matches the parse tree, i.e. the one where each semantic role maps to a part of the parse tree. Once the best match is found, it is straightforward to create the data structure as the

[2]https://verbs.colorado.edu/∼mpalmer/projects/verbnet.html

ones shown in Figure 5a and Figure 5b. The work of the semantic parser can thus be summarized in the following steps:

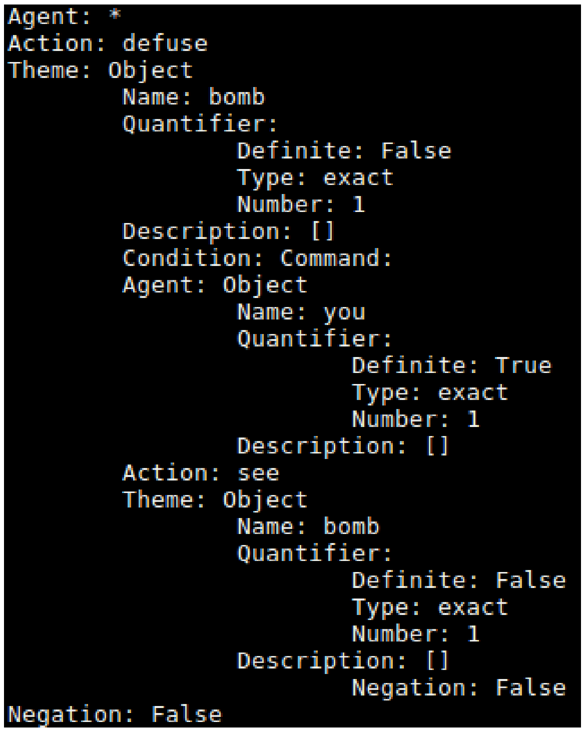
- Verbs and arguments are extracted from the parse tree and the corresponding senses are identified using the *VerbNet* database.
- Possible matching frames for each verb are identified, again consulting *VerbNet*, and the frame that expresses the most semantic roles with the parse tree is chosen.
- From the matched frame, a semantic structure is created which encodes the meaning of the natural language command written by the user.

Note that the advantage of using a large database such as *VerbNet* is that each natural language command is mapped to a set of *VerbNet* senses. This means that commands that contains different verbs but share the same meaning (again, for example "*If you see a bomb, defuse it.*" and "*If you detect a bomb, defuse it.*") are mapped to the same semantic structure. This allows the user to use a variety of individual verbs to express the same command, making the system more robust and able to capture the expressiveness of natural language. Furthermore, the system can be expanded to support further verbs and it is only limited in its vocabulary coverage by the content of *VerbNet*. Of course, the bigger the database the wider the vocabulary coverage, with the result that the user is given more freedom in terms of expressiveness.

3.2 Linear Temporal Logic Formula Generator

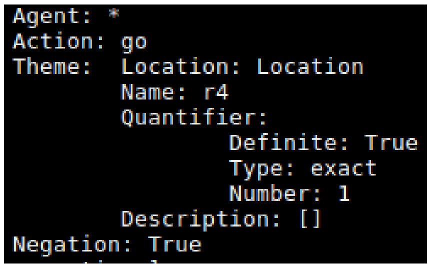
As we just explained, the output of the NLP pipeline is a set of semantic structures, each encoding the meaning of a natural language sentence written by the user in his/her specification. The next step is to translate these structures into logical formulas defining robot tasks. The logical formalism used for this project is *Linear Temporal Logic (LTL)*, a modal logic that includes temporal operators and thus allows to specify the truth values of atomic propositions over time. In order to generate LTL formulas from the semantic structures, we follow [1] and [2] and make the following assumptions:

- We assume that the positions of the robot in the environment can be discretized, i.e. they are represented by the finite set $\mathcal{R} = \{r_1, r_2, ..., r_n\}$, where exactly one r_i can be true at any time. The same applies for the robot actions $\mathcal{A} = \{a_1, a_2, ..., a_k\}$. Therefore, the set of propositions that the robot can act upon is given by $\mathcal{Y} = \{R, A\}$.
- We assume that the robot interacts with the environment using binary sensors and denote the sensor propositions by $\mathcal{X} = \{x_1, x_2, ..., x_m\}$.
- We assume that the natural language specification written by the user can include motion, for example "*Go to r3*" or "*Avoid r4*", as well as actions that the robot must perform subject to a particular stimulus from the environment, such as for example "*If you see a bomb, defuse it*".



```
Agent: *
Action: defuse
Theme: Object
        Name: bomb
        Quantifier:
                Definite: False
                Type: exact
                Number: 1
        Description: []
        Condition: Command:
        Agent: Object
                Name: you
                Quantifier:
                        Definite: True
                        Type: exact
                        Number: 1
                Description: []
        Action: see
        Theme: Object
                Name: bomb
                Quantifier:
                        Definite: False
                        Type: exact
                        Number: 1
                Description: []
                        Negation: False
Negation: False
```

(a)



```
Agent: *
Action: go
Theme:  Location: Location
        Name: r4
        Quantifier:
                Definite: True
                Type: exact
                Number: 1
        Description: []
Negation: True
```

(b)

Figure 5: The output of the semantic parser is a data structure that encodes the semantics of the user's command: **a)** semantic structure for the command "*If you see a bomb, defuse it*"; **b)** semantic structure for the command "*Don't go to r4*".

We apply these assumptions to the search and rescue scenario that we described in section 2. We assume we have a small map of 4 regions, in which the Turtlebot is able to navigate. Furthermore, the Turtlebot is able to detect and defuse bombs, as well as to detect and interact with hostages. This is represented by $\mathcal{Y} = \{r_1, r_2, r_3, r_4, defuse, interact\}$, and $\mathcal{X} = \{bomb, hostage\}$.

3.2.1 LTL syntax and semantics

Let $AP = \mathcal{X} \cup \mathcal{Y}$ be a set of atomic propositions. We construct LTL formulas from atomic propositions $\pi \in AP$ according to the following syntax:

$$\phi ::= \pi \mid \neg\phi \mid \phi \vee \phi \mid \bigcirc\phi \mid \Diamond\phi$$

where $\bigcirc$ is the *next* operator and $\Diamond$ is the *eventually* operator. Furthermore, given negation ($\neg$) and disjunction ($\vee$) we can define conjunction ($\wedge$), implication ($\Rightarrow$) and equivalence ($\Leftrightarrow$), as well as the *always* operator $\Box\phi = \neg\Diamond\neg\phi$.

The semantics of a formula ϕ is defined on a *trace*, i.e. an infinite sequence σ of truth assignments to the atomic propositions $\pi \in AP$ [6]. For example, the sequence σ satisfies the formula $\bigcirc\phi$ if ϕ is true in the next "step" of the trace (the next position in the sequence), and the formula $\Box\phi$ (always operator) if ϕ is true in every step of the trace. Similarly, the formula $\Diamond\phi$ expresses that ϕ is true at some position in the trace, and $\Box\Diamond\phi$ that ϕ is true infinitely often.

3.2.2 Generalized Reactivity(1) formulas

In this project, following the approach of [1] and [2], a particular subset of LTL is considered. This fragment of LTL is called *Generalized Reactivity(1)* or *GR(1)* [7]. The reasons for this choice are explained in Section 3.3. The *GR(1)* formulas considered here will have the form:

$$\phi = (\phi_e \Rightarrow \phi_s) \tag{1}$$

where the formula ϕ_e is an assumption about the environment (i.e. about the robot sensor propositions), while ϕ_s represents the desired robot behavior. These formulas have the form:

$$\phi_e = \phi_i^e \wedge \phi_t^e \wedge \phi_g^e \tag{2}$$

$$\phi_s = \phi_i^s \wedge \phi_t^s \wedge \phi_g^s \tag{3}$$

where:

- ϕ_i^e and ϕ_i^s are non-temporal formulas describing respectively the initial conditions of the environment (i.e. initial values of sensor propositions $\mathcal{X}$) and the robot ($\mathcal{Y}$).
- ϕ_t^e represents the possible evolution of the state of the environment, or in other words it constrains the next possible sensor values $\bigcirc\mathcal{X}$ based on

the current sensor $\mathcal{X}$ and robot values $\mathcal{Y}$. The formula consists of a conjunction of formulas of the form $\Box B_i$, where each B_i is a boolean formula constructed from subformulas in $\mathcal{X} \cup \mathcal{Y} \cup \bigcirc\mathcal{X}$ with $\bigcirc\mathcal{X} = \{\bigcirc x_1, ..., \bigcirc x_n\}$.

- ϕ_t^s represents the possible evolution of the state of the robot, or in other words it constrains the moves that the robot can make. The formula consists of a conjunction of formulas of the form $\Box B_i$, where each B_i is a boolean formula constructed from subformulas in $\mathcal{X} \cup \mathcal{Y} \cup \bigcirc\mathcal{X} \bigcirc \mathcal{Y}$.
- ϕ_g^e and ϕ_g^s represent the assumed goals of the environment (these may not exist) and the desired goals of the robot (these instead shall exist and are given by the user's specification). Both consists of a conjunction of formulas of the form $\Box\Diamond B_i$ where each B_i is a boolean formula.

3.2.3 A simple example

By a first look, *GR(1)* formulas appear very limited because of their restricted structure. However, this does not imply a big loss in terms of expressivity, as a wide range of natural language specifications can be covered using this class of formulas. Furthermore, they closely reflect the way in which most sensor-based robotics task are usually carried out. As an example, consider our search and rescue scenario where the robot can move between two regions (for simplicity). The robot is also able to sense bombs and defuse them, thus $\mathcal{Y} = \{r_1, r_2, defuse\}$. Given its initial position (e.g. r_1), we want the robot to move to the other region, and if it encounters a bomb along the way it should defuse the bomb. A natural language specification representing these tasks may be as simple as this: "*Go to r2. If you see a bomb, defuse it.*". We start by modeling the assumptions about the environment (i.e. about the sensor propositions $\mathcal{X}$), which are represented by Eq. 2. Since a bomb is part of the environment, the sensor proposition will simply be $\mathcal{X} = \{bomb\}$ which becomes true if the robot sensor detects a bomb. Furthermore, we assume that in the very first moment the robot hasn't sensed the environment yet, because at each step the robot first senses the environment and then moves. For this reason, the initial condition is $\phi_i^e = \neg bomb$. Thus the assumptions about the environment are given by:

$$\phi_e = \phi_i^e \wedge \phi_t^e \wedge \phi_g^e = \neg bomb \wedge \Box(bomb \Rightarrow \bigcirc bomb) \wedge \Box\Diamond True$$

where $\phi_t^e = \Box(bomb \Rightarrow \bigcirc bomb)$ represents the possible evolution of the state of the environment, in this case expressing that once the robot detects a bomb, the bomb doesn't move. Notice also that we do not assume any goal for the environment, thus we write $\phi_g^e = \Box\Diamond True$. Now we proceed onto modeling the assumptions about the robot (i.e. about $\mathcal{Y}$), which are given by Eq. 3. In doing so we will also consider the natural language specification. Initially the robot is in r_1, thus $\phi_i^s = (r_1 \wedge \neg r_2)$, and no other initial condition is needed. The possible changes in the robot state are instead expressed by ϕ_t^s which includes the possible transitions between regions, the mutual exclusion constraint (that is, at any step only one of r_1 and r_2 can be true), and a subformula defined by the natural language specification:

$$\begin{aligned}\phi_t^s = \quad & \Box\Big(r_1 \Rightarrow (\bigcirc r_1 \vee \bigcirc r_2)\Big) \wedge \Box\Big(r_2 \Rightarrow (\bigcirc r_1 \vee \bigcirc r_2)\Big) \bigwedge \\ & \Box\Big((\bigcirc r_1 \wedge \bigcirc\neg r_2) \vee (\bigcirc r_2 \wedge \bigcirc\neg r_1)\Big) \bigwedge \\ & \Box\Big((r_1 \wedge \bigcirc bomb) \Rightarrow \bigcirc defuse\Big) \wedge \Box\Big((r_2 \wedge \bigcirc bomb) \Rightarrow \bigcirc defuse\Big)\end{aligned}$$

The first two subformulas (i.e. transitions and mutual exclusion respectively in first and second row) can be automatically generated given a map of the environment, while the last subformula (third row) depends on the specification. In this case, the subformula expresses the fact that if the robot senses a bomb, it must defuse it. Finally, we have to model the desired robot goals which, again, depend on the specification. In this example the goal is only one, that is the robot must go to r_2, thus $\phi_g^s = \Box\Diamond r_2$. Summarizing, the desired robot behavior is expressed as:

$$\begin{aligned}\phi_s =& \phi_i^s \wedge \phi_t^s \wedge \phi_g^s \\ =& (r_1 \wedge \neg r_2) \bigwedge \\ & \Box\Big(r_1 \Rightarrow (\bigcirc r_1 \vee \bigcirc r_2)\Big) \wedge \Box\Big(r_2 \Rightarrow (\bigcirc r_1 \vee \bigcirc r_2)\Big) \bigwedge \\ & \Box\Big((\bigcirc r_1 \wedge \bigcirc\neg r_2) \vee (\bigcirc r_2 \wedge \bigcirc\neg r_1)\Big) \bigwedge \\ & \Box\Big((r_1 \wedge \bigcirc bomb) \Rightarrow \bigcirc defuse\Big) \wedge \Box\Big((r_2 \wedge \bigcirc bomb) \Rightarrow \bigcirc defuse\Big) \bigwedge \\ & \Box\Diamond r_2\end{aligned}$$

Now that we have completed our modeling of both the environment and robot specifications, we can combine everything together to obtain the required formula in Eq. 1.

3.2.4 Satisfiable robot behaviors

We just showed an example of what can be done using *GR(1)* formulas. In fact, these formulas can express a wide range of natural language specifications. However, due to the limited purposes of this project, at the present state the system can efficiently translate and synthesize (more details on this in Section 3.3) only some types of natural language commands. This will result in the robot being able to show only specific behaviors, thus satisfying only certain natural language specifications. Again following [2], we distinguish between two robot behaviors: *safety behaviors* are all those behaviors that the robot must *always* satisfy, while *liveness behaviors* are all those behaviors that the robot should *always eventually* satisfy. For example, the command "*Go to r2*"is a liveness command, while "*Avoid r4*" and "*If you detect a bomb, defuse it*" are safety commands because they must be satisfied at each step (i.e. they constrain the moves that the robot can make at each step). For these reasons, safety commands are usually encoded in ϕ_t^s formula and are of the form $\Box(formula)$, while liveness commands are encoded in ϕ_g^s formula and are of the form $\Box\Diamond(formula)$. At the present state, the natural language commands supported by the system are the followings:

- *Motion commands.* A natural language command that requires the robot to move to a region, such as "*Go to r*", is a liveness command and thus it is encoded in the ϕ_g^s formula. The LTL translation of this command is the formula:
$$\phi_{Go}^s = \Box\Diamond r$$
This formula guarantees that the robot will visit region r infinitely often, but doesn't guarantee that it will stay in r after arriving there. In order to force the robot to stay there, we must add a safety behavior (which will be encoded in ϕ_t^s) so that the LTL translation becomes:
$$\phi_{GoStay}^s = \Box\Diamond r \wedge \Box(r \Rightarrow \bigcirc r)$$
which states that if the robot is in r, it must be in r in the next step as well. If the natural language command involves motion to more than one region, e.g. "*Go to r2, r3 and r4*", the system splits the command at "and" and "," conjunctions and translates it into a conjunction of ϕ_{GoStay}^s formulas.

- *Avoidance commands.* Commands such as "*Avoid r*" or "*Don't go to r*" are translated into the temporal formula:
$$\phi_{Avoid}^s = \Box(\neg \bigcirc r)$$
which is a safety behavior (thus encoded in ϕ_t^s) and guarantees that in the next step the robot will not be in r. Again, in the case when the command involves multiple regions to be avoided, the system translates it into a conjunction of ϕ_{Avoid}^s formulas.

- *Conditional commands.* The last types of commands that the system currently supports are those commands in which the robot should do something based on stimuli from the environment. A template for such natural language commands is "If (not) *condition*, then do (not) action", where for "action" we usually mean an action primitive. This is translated into:
$$\phi_{Do}^s = \Box(OnCondition \Rightarrow \bigcirc a)$$
or in the negative case:
$$\phi_{Do}^s = \Box(OffCondition \Rightarrow \neg \bigcirc a)$$

For example, the command "*If you see a bomb, defuse it.*" is translated into:

$$\phi^s_{Do} = \Box(bomb \Rightarrow \bigcirc defuse)$$

Just like avoidance commands, conditional commands express a safety behavior and thus are encoded in ϕ^s_t. The system also accepts conditional commands that contain conjunctions or disjunctions. The latter case is easier to handle because we just have to split the command into separate commands: "If *condition1* or *condition2*, do *action*" becomes "If *condition1*, do *action*" $\wedge$ "If *condition2*, do *action*". Commands containing conjunctions instead require a different handling. For example the command "*If you see a bomb and a hostage, don't defuse the bomb*" is translated into the temporal formula:

$$\phi^s_{Do} = \Box\Big((bomb \wedge hostage) \Rightarrow \neg \bigcirc defuse\Big)$$

3.3 Automaton Synthesizer

After modelling a scenario using Eq. 1, we must now synthesize a controller that generates robot behavior which satisfies the formula ϕ, if the scenario is possible (i.e. if the formula is realizable). *Reactive synthesis* is an automated procedure to obtain a correct-by-construction reactive system from a temporal logic specification [8]. Instead of manually constructing a system and using model checking to verify its compliance with the specification, synthesis allows to automatically obtain a correct implementation (if this exists) of the system for a given specification. In the case of reactive synthesis, an implementation is typically an *automaton* that accepts input from the environment (e.g. from robot sensors) and produces the system's output (e.g. on robot actuators). In other words, the synthesis problem consists of finding an automaton whose behavior satisfies a given LTL formula (if this automaton exists). One of the main problems of applying reactive synthesis is its worst-case complexity, in fact for LTL the synthesis problem is generally proven to be doubly exponential in the size of the formula [8]. To address this problem, Pnueli et al. [7] suggested to restrict to the GR(1) fragment of LTL and proposed a symbolic synthesis algorithm which runs in polynomial $O(n^3)$, where n is the number of valuations of the sensor and state variables. The algorithm is informally described in the next paragraph.

3.3.1 Polinomial-time synthesis algorithm

The algorithm proposed in [7] models the synthesis problem as a two-players game, with one player being the system (i.e. robot) and its adversary being the environment. The environment has control over the input variables (i.e. sensor propositions $\mathcal{X}$) while the system controls the output variables (i.e. robot propositions $\mathcal{Y}$). Both the system and the environment start from an initial state, and then each player makes a transition according to its transition rules.

Making a transition simply means that the player gives a truth assignment to the variables it controls, if this assignment is allowed by the player's transition rules. The environment is the first to make a move, and then the system. The winning condition for the game is given as a GR(1) formula ϕ having the form of Eq. 1. We say that the system wins if, no matter what the environment does (i.e. for every possible transition the environment might take), there exists at least one countering move that the system can make such that it continues to satisfy ϕ. If the system wins we can extract an automaton for the robot. However, if the environment can make a move that falsify ϕ, we say that the environment wins and the desired robot behavior is unrealizable (i.e. we cannot extract an automaton). We can relate the formulas defined in Section 3.2.2 with the two-players game in the following way:

- ϕ_i^e and ϕ_i^s represent the initial states of the environment and the system respectively
- ϕ_t^e and ϕ_t^s represent the transition rules that the environment and the system respectively must observe when making a move
- the GR(1) formula $\phi = (\phi_g^e \Rightarrow \phi_g^s)$ is the winning condition of the game.

Note that $\phi = (\phi_g^e \Rightarrow \phi_g^s) = \neg\phi_g^e \vee \phi_g^s$, which means that ϕ is satisfied either when ϕ_g^s is true, i.e. when the desired robot behavior is satisfied, **OR** when ϕ_g^e is false, i.e. when the environment did not reach its goals (either because the environment is faulty or the system prevented it from reaching its goals). In both cases, we say that the system wins the game. Furthermore, note that if the environment does not behave as expected, i.e. if it violates its assumed behavior $\phi_i^e \wedge \phi_t^e$, the automaton will not be valid anymore.

Given an LTL specification expressed as a GR(1) formula ϕ the synthesis algorithm first checks whether the formula is realizable. If it is, the algorithm extracts an automaton which implements a strategy to satisfy the specification. By following this strategy the robot will satisfy the desired behavior. In this project we use the synthesis algorithm originally developed by Nir Piterman, Amir Pnueli and Yaniv Saar in [7], which is publicly available online [3].

3.3.2 Automaton creation

The synthesis algorithm generates an automaton which we can model as a tuple $\mathcal{A} = (\mathcal{X}, \mathcal{Y}, Q, q_0, \delta, \gamma)$ where:

- $\mathcal{X}$ is the set of environment propositions (inputs)
- $\mathcal{Y}$ is the set of system propositions (outputs)
- Q is the set of states

[3] https://github.com/VerifiableRobotics/LTLMoP/tree/development/src/etc/jtlv/GROne. The algorithm requires the Java Runtime Environment (JRE) and Java Development Kit (JDK) as well as the installation of the JTLV library: http://jtlv.ysaar.net/

- q_0 is the initial state
- $\delta : Q \times 2^{\mathcal{X}} \rightarrow 2^Q$ is the transition relation , i.e. given a state $q \in Q$ and a subset of inputs (sensor propositions) $X \subseteq \mathcal{X}$ that are true, we have that a successor of q is $\delta(q, X) = Q' \subseteq Q$
- $\gamma : Q \rightarrow 2^{\mathcal{Y}}$ is the state labeling function where $\gamma(q) = y$ and $y \in 2^{\mathcal{Y}}$ is the set of state propositions that are true in state q

We say that an input sequence $X_1, X_2, ..., X_j \in 2^{\mathcal{X}}$ is admissible if it satisfies ϕ_e. Under an admissible input sequence, a run of the automaton is a sequence of states $\sigma = q_0, q_1, etc.$ starting from the initial state q_0. At each state q_i the transition to the following state q_{i+1} is made according to the transition relation δ and the input propositions $X_i \in \mathcal{X}$ which are true in state q_i. This can be expressed as $q_{i+1} \in \delta(q_i, X_i)$. An interpretation of a run σ is a sequence $y_0, y_1, etc.$, where each $y_i = \gamma(q_i)$ is the set of outputs which are true in q_i. We define y_i as the "label" of state q_i, and we use the sequence of labels to represent the discrete path that the robot must follow: when the robot is in state q_i it will produce the outputs (actions) contained in the set y_i, when it is in q_{i+1} the outputs contained in y_{i+1}, and so on. We thus use the automaton as a discrete controller for the robot, at each step defining which actions must be executed. The controller is discrete in the sense that we have a finite set of states and the actions that can be executed are discretized. For example, moving to another region usually requires many low level actions (turning the wheels, accelerating and decelerating, avoiding obstacles, etc.). We intentionally omit these low level actions in the automaton, thus defining some a-priori granularity in the actions to be executed. Note that, if a non-admissible input sequence is given (i.e. one that violates ϕ_e), we will not be able to construct a correct path for the robot and thus the automaton will no longer be valid.

As an example, fig. 6 shows the automaton synthesized from the specification: "*Go to r2. If you see a bomb, defuse it.*". The number at the top of each node is the state ID, while the propositions written inside each node represent the state's label, namely the set of robot outputs that are true in that state. Note that a further proposition is used in this case, that is, the goal proposition mem_visit_r2. This acts as a "memory" of the goal that the robot must pursue (in this case, reaching $r2$), thus facilitating the control of the robot during a run of the automaton. In simple words, it tells the controller when the robot goal has indeed been achieved. Furthermore, each edge in the automaton represent a transition relation $\delta(q, X)$ for a given state q and a set of true sensor propositions X (inputs from the environment). In this trivial example, the set of sensor propositions is simply $\mathcal{X} = \{bomb\}$, therefore at each step the set of true sensor propositions $X \in \mathcal{X}$ can either be $X = \{\}$ or $X = \{bomb\} = \mathcal{X}$. Thus, edges in fig. 6 are either unlabelled or labelled with *bomb*. Finally, note that by following the automaton, the robot indeed satisfies the above specification. For example, starting in $r1$ (node 0), if the robot sees a bomb it proceeds to defuse it (node 2), otherwise it will move to $r2$ satisfying the goal proposition mem_visit_r2.

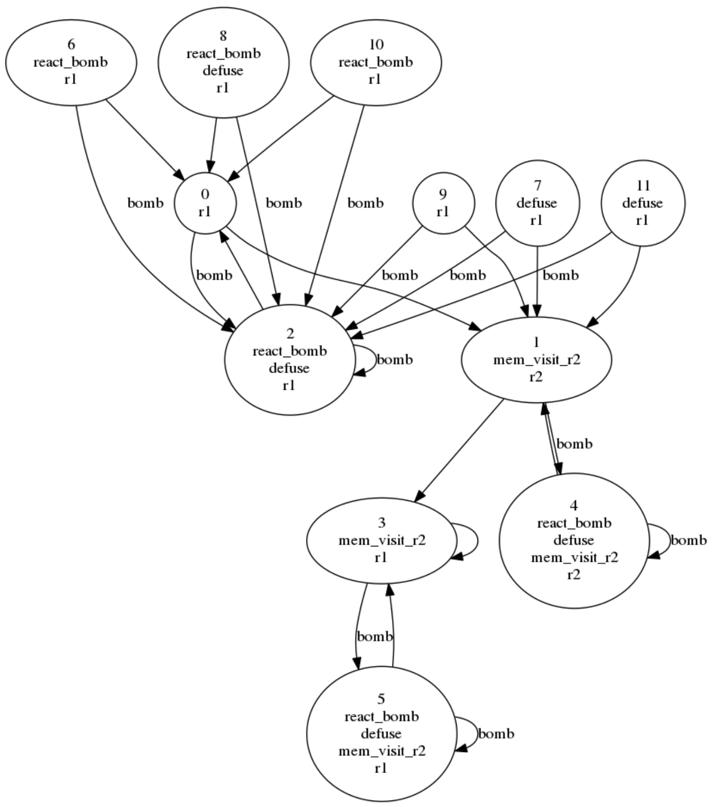


Figure 6: Automaton synthesized from the specification "*Go to r2. If you see a bomb, defuse it.*". Each node is characterized by a number (state ID), while the propositions written inside each node represent those propositions that are true in that state. Labelled edges represent trasitions due to inputs from the environment (in this case, seeing a bomb), while unlabelled edges represent tranisitions when no input is received from the environment.

In the first case, after the bomb has been defused, the robot goes back to state 0, and then to state 2 where the goal is reached.

3.4 Continuous Robot Controller

After an automaton has been synthesized which satisfies a given specification, the last step involves running the automaton to extract a discrete path for the robot. By following this path, the robot will be able to satisfy the specification. In this project, the path extraction from the automaton and its execution are carried out simultaneously. Initially, the robot is in state q_0, placed in some region r_{i_0} of the environment. The robot first senses its environment, thus determining X_0, i.e. all sensor propositions that are true in q_0. The next state

q_1 is selected from the automaton such that $q_1 \in \delta(q_o, X_0)$. This is equivalent to choosing the edge in the automaton whose label correspond exactly to X_0. From the next state q_1 we can extract the next robot ouput $\gamma(q_1) = y_1$, which includes the next region r_{i_1} that the robot should reach, as well as all the actions that shall be executed in state q_1. At that point, the robot moves to state q_1, i.e. it reaches region r_{i_1} and executes the required actions (if any). From there, we repeat the process in the same exact way, moving to the next states q_3, q_4, etc. until the goal is reached. This process is achieved through a simple algorithm that continuosly extracts the next state from the automaton given the sensor outputs from the robot, and then drives the robot to that state. The following pseudocode summarizes these steps:

Algorithm 1 Continuous robot controller (pseudocode)

```
1: $r_{i_j} \leftarrow robot.getPosition()$
2: $a_j \leftarrow \emptyset$                    //no actions in the very first state
3: $y_j \leftarrow \{r_{i_j}, a_j\}$
4: $q_j \leftarrow automaton.getState(y_j)$
5: $g \leftarrow getUnsatisfiedGoals(q_j)$
6: **while** $g \neq \emptyset$ **do**
7:     $X_j \leftarrow robot.senseEnvironment()$
8:     $q_{j+1} \leftarrow automaton.getNextState(q_j, X_j)$
9:     $y_{j+1} \leftarrow \gamma(q_{j+1})$
10:    $robot.produceOutputs(y_{j+1})$    //move to $r_{i_{j+1}}$ and do actions $a_{j+1}$
11:    $g \leftarrow getUnsatisfiedGoals(q_{j+1})$
12: **end while**
```

4 Testing the system in Gazebo

After developing the system architecture described in section 3, we tested our system in a simulated environment. The testing process is easier with a simulated robot, as there are no strict hardware or software requirements. This allows anybody to test the system on his/her own personal computer, without needing a lab environment and a physical robot.

As we described earlier in section 2, we consider a Turtlebot working in a search and rescue scenario, and we simulate this scenario using the popular robotics simulator Gazebo[4]. For this purpose, we create the environment shown in Figures 7 and 8. This consists of a building with four rooms, as well as human hostages and bombs which may be present in one or more of the rooms. The robot knows the map of the building, but it doesn't know how many hostages nor bombs are present, nor their location inside the building. Note that, every time the simulation is started, bombs and hostages are created and scattered randomly within the rooms. This means that the robot will face a new scenario in each simulation.

[4]http://gazebosim.org/

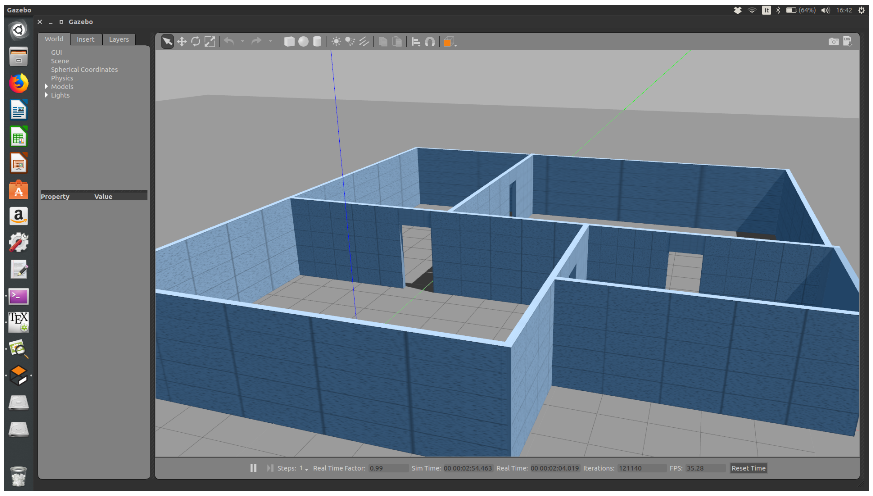


Figure 7: A simple building with four rooms.

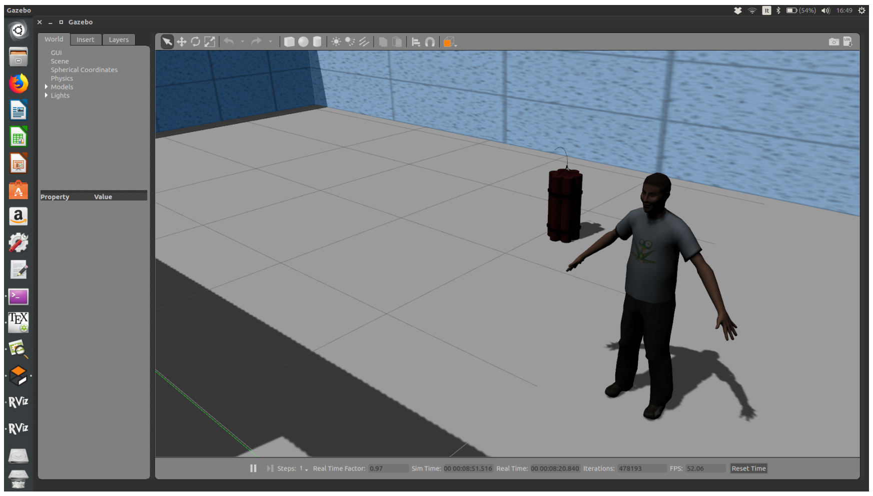


Figure 8: A human hostage and a bomb object in the simulated environment.

We name the four rooms in the map as *r1, r2, r3, r4*. These tags are easier to remember than common names such as *hall, lounge, kitchen*, and so on [5]. The skill set of our robot is quite small: the robot can navigate between rooms, detect and interact with hostages, detect and defuse bombs. The interaction with the hostage takes place in the following way: the robot starts by asking the hostage whether he/she is wounded, if the hostage says "yes" the robot will tell the hostage to wait while it calls for help, while if the hostage says "no" the robot will simply indicate the fastest route from the hostage location to the exit of the building. Despite the skill set of the robot being so restricted, it is enough to test the effectiveness of our system. In particular, we are interested in seeing how the robot responds to navigation commands and conditional action commands. We discuss the results below, and we provide recorded videos of the simulation.

4.1 Testing navigation commands

(Link to the recorded video: `https://youtu.be/pZ2h2DDL5E4`) These are commands that force the robot either to visit one or more rooms in the building, or to avoid them. Through the use of coordinating conjunctions (*and* and *,*) we can force the robot to visit multiple rooms ("*Go to r2, r3, and r4*"), or to avoid them ("*Don't go to r3, and r4*"). The latter case is quite interesting. In fact, negating the verb *go* (i.e. *Don't go*) is not the only way in which we can express negation. Because the system makes use of the *VerbNet* database, we can use positive verbs that implicitly convey negation. For example, as we show in the video, we can say "*Avoid r4*" instead of "*Don't go to r4*", and the robot will understand and execute both commands in the same way, since they have equivalent meaning. Unfortunately, verbs that are not in *VerbNet* will not be understood by the robot. For example, the command "*Stay away from r4*" is not valid, because *stay away* is not included in *VerbNet*. The limitation in this case resides in the database that we are using. The good aspect is that we can always extend this database to include more verbs, or use a more complete database. As a final remark, in the video we show how the system can support commands that include both positive and negative forms, such as "*Go to r2, and avoid r4*."

4.2 Testing conditional commands

(Link to the recorded video: `https://youtu.be/O3EtuK8yWLY`) These are commands of the type "if *stimulus* then *action*", where a stimulus is a certain input from the environment, i.e. something that the robot can detect using its sensors. In our search and rescue scenario, the robot can only detect bombs and/or hostages, thus a stimulus will be the presence of one or both of these objects.

[5]Because the system uses a syntactic parser, it is preferable to name rooms with nouns, and to avoid words that have other syntactic functions, such as verbs, articles or conjunctions. Of course, it is unlikely that someone would name a room using a verb or an article, but this point was worth mentioning.

The second part of the command involves an action that the robot must execute once the stimulus from the environment is verified. In this case, the only two actions that the robot can execute are *defuse* (a bomb), and *interact* (with a hostage). As shown in the video, examples of valid conditional commands are therefore: "*If you see a bomb, defuse it*", "*If you see a hostage, interact with him*", "*If you see a bomb and a hostage, don't defuse the bomb*". In the last command, note that the robot understands conjunctions as well as negation (as in the case of navigation commands). In the second type of command, note that different personal pronouns can be used. In fact, using *him, her* will not affect the correct execution of the command, since execution of a command depends only on the command's semantics, which in this case is not affected by the use of different pronouns. Finally note that, as in the case of navigation commands, we can use alternative verbs to convey the same meaning (as long as these verbs are included in *VerbNet* database). For example, as we show in the video, we can say "*If you see...*" or "*If you detect...*", or even use the verb *perceive* with equal results [6]. This denotes one of the advantages of the implemented system, which is its flexibility to adapt to the expressiveness of natural language sentences.

5 Conclusions

In this project we try to bridge the gap between the high expressiveness of natural language and the low level nature of robot control. We developed a system through which a user can give commands to a robot expressed in natural language, more specifically in written English language. We also provide a simulated environment where the above system can be tested: the user writes commands in English language, and watch them being executed by a simulated robot. This simulated robot is designed to operate on a specific search and rescue scenario, and thus will only understand and execute commands which are relevant to this scenario. Despite these restrictions, the simulated scenario proved to be a good way to test our system, and to point out some of its strenghts and weaknesses (links to videos of the simulations are provided in section 4). As we show in the videos, the system is quite flexible and it adapts well to the complexity of natural language. Human language is characterised by high expressiveness, variability, and ambiguity, and the system is able to cope with these complex aspects, at least to some degree. For example, the robot can deal with subtle syntactical differences such as verb synonyms and pronouns, and it is therefore able to understand sentences that have different syntactical structure but share equal semantics. The system of course is far from being perfect. The syntactic parser in general showed good accuracy but

[6] *see, detect*, and *perceive* are all included in *VerbNet*, thus we can use them interchangeably in our commands. Unfortunately, *VerbNet* does not include any synonym of *defuse* and *interact* verbs, which means that we are forced to use these verbs when referring to robot actions, without alternatives. This highlights the pros and cons of using a database such as *VerbNet*: in some cases we have plenty of choice, while in others not that much.

it is not flawless, and in some cases it may miss some syntactic structures and thus lead to parsing errors. The database of verbs that we use, named *VerbNet*, shows some limitations, for example it does not include a wide range of verb synonyms. However, a good aspect is that the database can always be extended to include more verbs, or substituted with a more accurate and wider database.

Controlling robots with natural language is a challenging problem, mainly because commands expressed in human language are high level and often abstract. Sentences can be ambiguous at times, requiring additional information (context) that is not conveyed by the meaning of the single words. This makes it hard to extract the correct meaning from a given sentence. However, if we focus on a smaller scenario, human language becomes more predictable and less ambiguous. In this project we showed that, in such a scenario, the implemented system can achieve good results. Furthermore, the system itself is open for possible future extension and/or improvement. For example, an interesting extension would be to include speech recognition capabilities, thus allowing the system to process not only written natural language commands, but also spoken commands.

References

[1] Hadas Kress-Gazit, Georgios E. Fainekos and George J. Pappas. 2007. *Where's Waldo? Sensor-Based Temporal Logic Motion Planning.* Proceedings of the 2007 IEEE International Conference on Robotics and Automation.

[2] Hadas Kress-Gazit, Georgios E. Fainekos and George J. Pappas. 2007. *From Structured English to Robot Motion.* IEEE/RSJ International Conference on Intelligent Robots and Systems.

[3] Bikel, D. 2004. *Intricacies of Collins parsing model.* Computational Linguistics 30(4):479511.

[4] Kummerfeld, Jonathan K., et al. 2012. *Parser showdown at the wall street corral: An empirical investigation of error types in parser output.* Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning. Association for Computational Linguistics.

[5] Gabbard, R.; Marcus, M.; and Kulick, S. 2006. *Fully parsing the penn treebank.* In Proceedings of the main conference on Human Language Technology Conference of the North American Chapter of the Association of Computational Lin- guistics, 184191. Association for Computational Linguistics.

[6] E. M. Clarke, O. Grumberg, and D. A. Peled. 1999. *Model Checking.* MIT Press, Cambridge, Massachusetts.

[7] N. Piterman, A. Pnueli, and Y. Saar. *Synthesis of Reactive(1) Designs.* 2006. In VMCAI, pages 364380, Charleston, SC.

[8] A. Pnueli, R. Rosner. *On the Synthesis of a Reactive Module.* 1989. In *POPL '89*: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 179 - 190. ACM Press.

6 Appendix - Code Instructions

Dependencies. The code for the entire project is provided along with this report. It has been written in Python 2 and tested on Ubuntu 16.04. In order to be run, it requires the following dependencies:

- Python 2.7 (code will not work with Python 3)
- Java Runtime Environment (need 1.6 or 1.7) and Java Development Kit:

  ```
  sudo apt-get install default-jre default-jdk
  ```

- Python NumPy, Scipy, and wxPython:

  ```
  sudo apt-get install python-numpy python-scipy python-wxtools
  ```

- Python Polygon2, provided inside the project folder or you can just:

  ```
  pip install Polygon2
  ```

- ROS Kinetic (code might not work if you use other ROS versions)
- Gazebo 7 (code was tested only on Gazebo 7.14.0)

Running the code. Before running the code, make sure you have Python on your PATH and the command *python* should point to your Python 2 version (not Python 3, in case you have both installed).

1. Download the project folder from this google drive:

   `https://drive.google.com/drive/u/1/folders/1xCoy8a98U8dNKKDxFIFIh2TjV6mAYYv-`
   and place it wherever you prefer on your computer

2. Go to your ROS installation, and open the turtlebot navigation parameters file (e.g. using gedit):

   ```
   cd /opt/ros/kinetic/share/turtlebot_navigation/param/
   gedit dwa_local_planner_params.yaml
   ```

   On this file set *max_rot_vel* equal to 1.0 (default value is 5.0 but this causes issues in the turtlebot navigation)

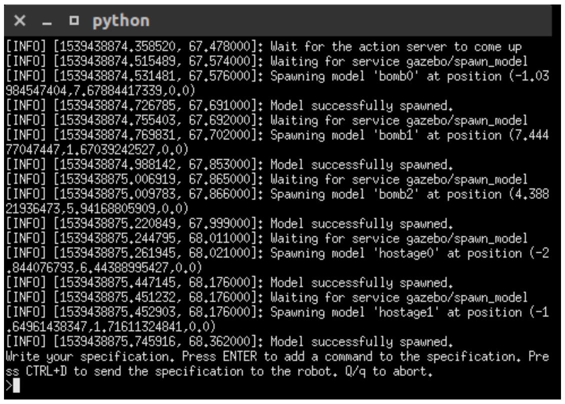
3. Open a terminal and go to the project folder:

   ```
   cd your/path/to/hri_project
   ```

4. Within the project folder, make the script *run.sh* executable and then execute it:

   ```
   chmod +x run.sh
   ./run.sh
   ```

5. At this point, Gazebo should have started, as well as several other terminal windows. Look for the window wich asks you to write your specification:



```
[INFO] [1539438874.358520, 67.478000]: Wait for the action server to come up
[INFO] [1539438874.515489, 67.574000]: Waiting for service gazebo/spawn_model
[INFO] [1539438874.531481, 67.576000]: Spawning model 'bomb0' at position (-1.03
984547404,7.67884417339,0.0)
[INFO] [1539438874.726785, 67.691000]: Model successfully spawned.
[INFO] [1539438874.755403, 67.692000]: Waiting for service gazebo/spawn_model
[INFO] [1539438874.769831, 67.702000]: Spawning model 'bomb1' at position (7.444
77047447,1.67039242527,0.0)
[INFO] [1539438874.988142, 67.853000]: Model successfully spawned.
[INFO] [1539438875.006919, 67.865000]: Waiting for service gazebo/spawn_model
[INFO] [1539438875.009783, 67.866000]: Spawning model 'bomb2' at position (4.388
21936473,5.94168805909,0.0)
[INFO] [1539438875.220849, 67.999000]: Model successfully spawned.
[INFO] [1539438875.244795, 68.011000]: Waiting for service gazebo/spawn_model
[INFO] [1539438875.261945, 68.021000]: Spawning model 'hostage0' at position (-2
.844076793,6.44388995427,0.0)
[INFO] [1539438875.447145, 68.176000]: Model successfully spawned.
[INFO] [1539438875.451232, 68.176000]: Waiting for service gazebo/spawn_model
[INFO] [1539438875.452903, 68.176000]: Spawning model 'hostage1' at position (-1
.64961438347,1.71611324841,0.0)
[INFO] [1539438875.745916, 68.362000]: Model successfully spawned.
Write your specification. Press ENTER to add a command to the specification. Pre
ss CTRL+D to send the specification to the robot. Q/q to abort.
>
```

6. Type your commands on the window, pressing ENTER after each command. When you are done typing commands, press CTRL+D to send the commands to the robot, and watch the robot execute them in Gazebo. Press q instead if you want to exit.