UNIVERSITÄT KOBLENZ · LANDAU
Fachbereich 4: Informatik

Generic task planning using semantic annotations in maps

Masterarbeit
zur Erlangung des Grades
MASTER OF SCIENCE
im Studiengang Computervisualistik

vorgelegt von

Carmen Navarro Luzón

Betreuer: Dipl.-Inform. Viktor Seib, Institut für Computervisualistik, Fachbereich Informatik, Universität Koblenz-Landau
Erstgutachter: Prof. Dr.-Ing. Dietrich Paulus, Institut für Computervisualistik, Fachbereich Informatik, Universität Koblenz-Landau
Zweitgutachter: Dipl.-Inform. Viktor Seib, Institut für Computervisualistik, Fachbereich Informatik, Universität Koblenz-Landau

Koblenz, im September 2011

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Die Vereinbarung der Arbeitsgruppe für Studien- und Abschlussarbeiten habe ich gelesen und anerkannt, insbesondere die Regelung des Nutzungsrechts.

Mit der Einstellung dieser Arbeit in die Bibliothek bin ich einverstanden.	ja ☒	nein ☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	ja ☒	nein ☐

Koblenz, den 22. September 2011

Kurzfassung

Das Ziel dieser Masterarbeit ist, dass der Roboter Lisa komplexe Befehle verarbeiten und Information aus einem Kommando extrahieren kann, die benötigt werden, um eine komplexe Aufgabe als eine Sequenz von kleineren Aufgaben auszuführen. Um dieses Ziel zu erreichen wird das Bild, das Lisa von ihrer Umgebung hat, mit semantischen Informationen angereichert. Diese Informationen werden in ihre Karte eingefügt werden.

Es wird angenommen, dass der komplexe Befehl bereits geparst worden ist. Deshalb ist die Verarbeitung des Inputs, um daraus einen geparsten Befehl zu erstellen, kein Teil dieser Masterarbeit.

Die Karten, die Lisa aufbaut, werden mit semantischen Anmerkungen annotiert. Zu diesen Anmerkungen gehört jede Art von Informationen, die nützlich zur Ausführung allgemeiner Aufgaben sein könnte. Das kann zum Beispiel eine hierarchische Klassifizierungen von Orten, Objekten und Flächen sein.

Die Abarbeitung des Befehls mit den zugehörigen Informationen über die Umgebung wird eine Sequenz von Aufgaben auslösen. Diese Aufgaben sind die bereits implementierten Fähigkeiten von Lisa, wie zum Beispiel Objekterkennung oder Navigation. Das Ziel dieser Masterarbeit ist aber nicht nur, die vorhandenen Aufgaben zu nutzen, sondern auch das Hinzufügen von neuen Aufgaben zu erleichtern.

Abstract

The purpose of this master thesis is to enable the Robot Lisa to process complex commands and extract the necessary information in order to perform a complex task as a sequence of smaller tasks. This is intended to be achieved by the improvement of the understanding that Lisa has of her environment by adding semantics to the maps that she builds.

The complex command itself will be expected to be already parsed. Therefore the way the input is processed to become a parsed command is out of the scope of this work.

Maps that Lisa builds will be improved by the addition of semantic annotations that can include any kind of information that might be useful for the performance of generic tasks. This can include (but not necessarily limited to) hierarchical classifications of locations, objects and surfaces.

The processing of the command in addition to some information of the environment shall trigger the performance of a sequence of actions. These actions are expected to be included in Lisa's currently implemented tasks and will rely on the currently existing modules that perform them. Nevertheless the aim of this work is not only to be able to use currently implemented tasks in a more complex sequence of actions but also make it easier to add new tasks to the complex commands that Lisa can perform.

Acknowledgment

There are many people without whose help this Masterarbeit would not have been possible. In first place I must say thanks to Professor Dietrich Paulus for allowing me to write it under his guidance and to Marcel Häselich, who offered me the opportunity to work with Lisa, both of them disregarding the fact that I was an unknown, poorly-german-speaking Erasmus student. I am very thankful to all the people in Robbie 16 project for being so welcoming and nice that I have sometimes forgotten I was actually working there. I have really felt among friends in the team, I have felt valuable and I have learned so much from everyone. Thanks for showing me the real goods of team work, for an amazing year full of worth experiences and great memories, Robbie 16 will always have a little place in my heart.

Very fondly I have to say thanks to Susi Thierfelder, leader of the homer team and also a good friend. I have learned a lot during these months working together and we have laughed so much, thanks for all of that.

And the truth is that this work would not exist at all without the priceless help of Viktor Seib, best Betreuer ever and a great friend. Thank you for being always available, always helpful and for taking me so seriously without being serious at all. You kept me motivated and gave me the necessary insight to carry out this project.

Contents

Contents 7

List of Figures 13

List of Algorithms 15

List of Tables 15

1 Introduction 19
- 1.1 Motivation . . . 19
- 1.2 Goal . . . 20
- 1.3 Structure . . . 20

2 State of the art 21
- 2.1 Task planning . . . 21
  - 2.1.1 Introduction . . . 21
  - 2.1.2 Finite-state machine behavior . . . 22
  - 2.1.3 Logics-based problem solving . . . 24
  - 2.1.4 Limitations of logic-based approaches . . . 26
  - 2.1.5 Situated agents: A hybrid approach . . . 27
- 2.2 Semantics in maps and data representation . . . 27
  - 2.2.1 Introduction . . . 27
  - 2.2.2 Navigation-oriented mapping . . . 28
  - 2.2.3 Usefulness of semantic information . . . 29
  - 2.2.4 Integrating semantics in maps . . . 30
  - 2.2.5 Knowledge structure . . . 32
  - 2.2.6 Acquisition of semantic knowledge . . . 33
- 2.3 Natural Language Processing . . . 33
  - 2.3.1 Introduction . . . 33
  - 2.3.2 Human-computer interaction cycle . . . 33
  - 2.3.3 Language parsing and understanding . . . 34

2.3.4 Models and algorithms 35

3 Integrating Task Planning and Semantics 37

3.1 Overview of the system 37

3.2 Task planning 38

3.2.1 Current state of task planning 38

3.2.2 Task planning approach 42

3.2.3 Abstraction of tasks and wrapping. 43

3.2.4 Generalization of the messages 47

3.2.5 State machine specification 47

3.2.6 Implementation of the tasks 49

3.2.7 Executing the tasks: The `TaskPlannerModule` 54

3.2.8 Increasing the set of tasks 58

3.3 Semantics in maps and data representation 58

3.3.1 Current map representation 58

3.3.2 Structuring semantic knowledge 59

3.3.3 Semantic knowledge representation 61

3.3.4 New map layers for representing semantic annotations . . 62

3.3.5 Adding semantic information: GUI 63

3.4 String command extraction 64

3.4.1 Current input processing 64

3.4.2 Definition of a desirable valid input 65

3.4.3 Implementation of the Simple command extractor 68

4 Evaluation 71

4.1 Measuring quality 71

4.2 General Purpose 72

4.2.1 Test cases 73

4.2.2 Errors discussion 75

4.3 RIPS 75

4.4 Results overview 76

5 Further work 81

6 Summary 83

A Framework user guide 85

A.1 How does the system work? 85

A.2 How do wrappers work? 86

A.3 How to create a new wrapper 88

A.4 How to use tasks that are already wrapped 89

A.5 Semantic Knowledge structure 90
A.6 GUI general information 90
A.7 FAQ and Troubleshooting 91

List of Figures

2.1 Traditional robot control system 23
2.2 Robot control system proposed by R. Brooks 24
2.3 Maio and Rizzi layered architecture. 31
2.4 Zender, Jensfelt, Mozos *et al.* layered architecture. 31
2.5 Galindo *et al.* semantic representation 32
2.6 Human-Computer interaction cycle by Filipe and Mamede [FM09]. 34

3.1 System overview 38
3.2 Lisa's software architecture. 39
3.3 Current interaction between modules 40
3.4 Proposed interaction between modules 41
3.5 Task class structure 45
3.6 State diagram for TaskModule classes 48
3.7 Class diagram for TaskModule classes' implementation 50
3.8 State diagram for `TaskPlannerModule` 55
3.9 Proposed space knowledge hierarchy 60
3.10 Proposed artifact knowledge hierarchy 61
3.11 Semantics class representation 63
3.12 New GUI screenshot 64

4.1 State diagram of the old GeneralPurposeModule. 78
4.2 State diagram of the new GeneralPurposeModule. 79

A.1 Interaction of the modules with the framework 86
A.2 Wrapper diagram example 87
A.3 State diagram for TaskModule classes 88
A.4 Semantic Tab in the GUI 91

List of Algorithms

1 Point-in-Polygon algorithm. 46

List of Tables

4.1 Comparative of lines of code between the old and new system . . 77
4.2 Comparative of subscribed and sent messages ind old and new modules 77

Chapter 1

Introduction

1.1 Motivation

Robotics have evolved quickly along time from a very experimental field to a rich, purposeful area of study. If people will ever have household robots in the future, these robots must evolve in order to encompass a broader set of household tasks. To this end, they need to develop an understanding of their environment as well as a human-friendly language to communicate with their users. Semantic knowledge about the environment is therefore a desirable feature any household robot should have in order to be useful as a human companion.

However, the nature of Lisa's project sometimes entails difficulties to integrate new features to the current existing software. The main reason for this is that projects like Lisa and Robbie often start as small experiments and grow with many different people's contributions along time. When software evolves in this fashion it is usual that it becomes bigger than expected and many features that were not planned in advance are also added, with the consequent adaptation of the software.

Indeed, sometimes the initial design of the software is not ready for an easy addition of these features. As a result, there are usually moments where it seems difficult to preserve the unity of these projects and the effort that needs to be made in order to maintain this code grows exponentially with the size of the software. This is the reason why sometimes it is necessary to stop walking forward for a moment and consider some restructuring that allows a project to go further on its way.

Accordingly, in order to allow robots to grow further and endow them with the ability to undertake new tasks with an easy integration of them, there is the need to rethink and reengineer their architecture in order to facilitate their adaptation to the always changing set of tasks they are required to tackle. Moreover, there is the further necessity to include in their perception of their environment general and factual knowledge about the world.

1.2 Goal

The aim of this master's thesis is to provide support and infrastructure to hold together these new parts of the software that have been growing independently and enable an easier further development in either new modules and features or currently existing ones. The already implemented modules will also be adapted to the new structure with the purpose of making the growth of the project faster and more effective and facilitating the job of developers, making it easier to combine already implemented tasks and create new ones.

At the same time, it intends to develop a structure that can endow robots with increasing generality in their skills. In order to accomplish this, they will be equipped with a better understanding and knowledge about their environment. This addition of knowledge is also intented to be performed in a fashion that allows further growth in the future.

1.3 Structure

This work is organized as follows. Chapter 2 shows an overview of what the scientific community has already done in the past years to address similar problems to the one this work intends to solve. Considering all the information gathered, chapter 3 depicts the theoretical approach chosen and explains the process of developing the software and the software itself. The results obtained are described and evaluated in chapter 4. Chapter 5 is a reference for the possible further work once this is accomplished and finally chapter 6 shows some conclusions obtained after the completion of this master's thesis.

Chapter 2

State of the art

This chapter is an overview of the current work of the scientific community in the fields related to this work. Section 2.1 describes the current state of art regarding task planning. Section 2.2 analyses the existing work in semantics representation, specially related to the area of robotics, and section 2.3 explains the sources found in string command extraction and natural language processing.

2.1 Task planning

This section describes the current approaches used in task planning for robots. The main approaches will be discussed along with their advantages and disadvantages.

2.1.1 Introduction

From the early times of artificial intelligence (AI), the main goal of this field has been to produce intelligent systems and understand human intelligence. From this beginning the scientific community has been ambitious about the possibilities and there were thoughts about being able to build intelligent robots. This is one of the main reasons why the fields of AI and robotics have been influencing the work in each other since the very beginnings of both [Bro91b].

Several approaches have been considered in order to make robots behave in an intelligent fashion. The part of this job considered as task planning or problem solving, which encompasses the process of planning a high-level sequence of actions that would allow a certain agent to achieve a goal or perform a task, has been a broad field of research in AI for a long time. Even before its application to the field of robotics, some examples of problem solving predecesors can be found in the literature. Traditionally two main solutions have been given to this problem: **finite-state machine behavior** and **logics-based problem solving**.

2.1.2 Finite-state machine behavior

One of the main approaches used to solve the task planning problem in a robot are **finite-state machines** (FSMs). FSMs are a mathematical abstraction designed to model behavior regarding a finite set of possible states and transitions between these states. In this way, a robot would be in one of its finite set of possible states according to its internal values. In addition, its current state would change depending on internal changes or external stimuli caught by its sensors.

Finite-state machines allow a layered decomposition of tasks, endowing robots with increasing abilities. The traditional approach to this idea is shown in figure 2.1. It built a chain of information processing modules that passed the information from the layer connected to sensors to the layer in control of actuators, where every layer between these two only interacted with the layers immediately next to it [Bro91b]. Several authors have applied this traditional approach in the past.

One example of this is **Shakey** [Nil84], a mobile robot that navigated through a specially designed environment trying to satisfy a given goal. Its structure consisted of what Nilsson called *Low-Level Actions*, which defined the interface with the robot's hardware, *Intermediate-Level Actions*, which controlled the performance of the *Low-Level Actions* and communicated with a planning system layer. This planning system layer communicated with the executive layer, which had access to the actuators.

Another example is given by J. Crowley [Cro85], where the local model is updated by the lowest layer. This layer has access to the information supplied by the sensors and the rest of the framework relies on this model. The local model includes dynamic information about surfaces and obstacles detected by the sensors. The remaining modules have access to this model and the highest layer, in this case the *Path execution* layer, is the only one accessing motion commands. Note that in this case the layering is not as strict. Nevertheless, it still isolates the layers that can access the sensors and actuators.

Further approaches changed this view to one where every layer could have access to sensors and actuators but that kept the layered structure. This was done by **augmented finite-state machines** (AFSMs) and was given the name of **subsumption architecture** [Bro86]. As shown in figure 2.2, each layer in a subsumption architecture is able to inject information in the layers below in order to change the usual flow of data. According to this, the control is not centralized [Bro91a], since every layer has its own set of inner states. They change

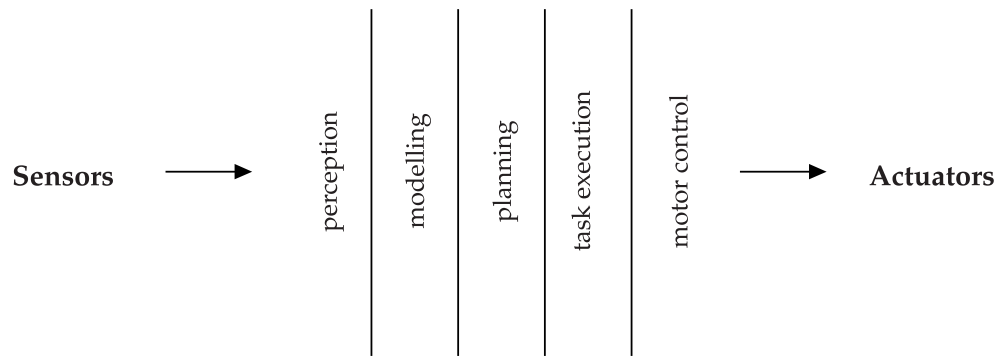


Figure 2.1: Traditional decomposition of the control system of a robot [Bro91b]. Here, only the perception layer is affected by sensors and only the motor control layer can access the actuators.

accordingly to the messages they receive, the values of inner variables that can include also specified timeouts to handle the possibility of misperceptions or error states.

There are some more authors that have used finite state machines to model a robot behavior. For instance, Jerry E. Pratt used a simple state machine for the *Stupid walking* algorithm of his biped walking robot [Pra95]. Collins and Ruina's bipedal robot [CR05] also has a control hierarchy based in a finite state machine, where all inputs are switches and the outputs are on/off activations.

Finite-state machines seem to be an efficient option where a quick answer is needed. One of the problems found by the AI community is that sometimes reasoning systems needed to perform such amount of processing that the answer was not fast enough for a dynamic real world. Moreover, it is reasonable to think that intelligence should be reactive to some aspects of the environment (e.g. collision avoidance) [Bro91b].

Additionally, Agre and Chapman pinpointed that most of people's daily life activity does not really involve problem solving and planning but routines in a dynamic and changing but benevolent environment instead [Bro91b].

All in all, finite-state machines have been broadly chosen as an efficient option that can endow robots with a reactive behavior. This behavior can be modelled through a layered architecture in an either strict or flexible fashion. Moreover, it allows the designer to encapsulate and control the access to sensors and actuators, although some proposals [Bro91b] suggest leaving free access to them.

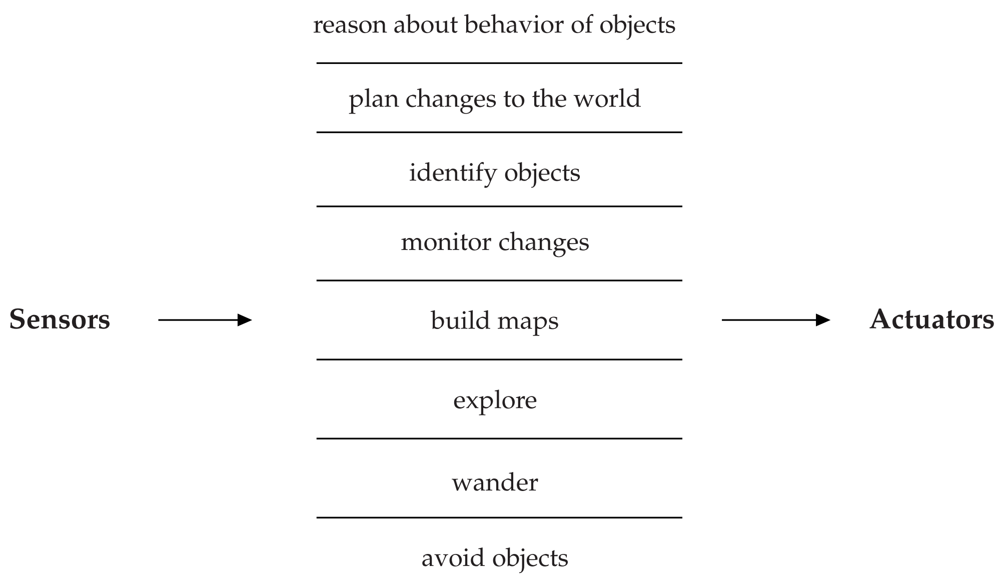


Figure 2.2: Robot control system decomposition in *levels of competence* as proposed by R. Brooks [Bro91a]. Every layer has access to sensors and actuators although each layer is supported by the layers under it.

2.1.3 Logics-based problem solving

Another main approach that has been broadly considered to provide robots with intelligent behavior is the use of **logics**. A relevant early example of these logics-based problem solving can be found in [Gre69]. There, the resolution proof procedure is used to solve problems in an automated way. As an application, an approach on simple robot problem-solving is given.

Following this line of logical reasoning about the world state and change, Stanford Research Institute developed **STRIPS**[1] [FN71], a problem solver based on first-order predicate calculus. STRIPS tries to find a sequence of steps that lead from an initial world model to a final world model, in which a certain goal condition can be proven to be true. Its search space is therefore conformed by the initial world model, the goal statement and a set of operators that can transform a world model into another. Each world model is a set of well-formed formulas and the goal statement is also a well-formed formula. The operators that transform a world model into another are as well defined by two parts. The first is a description of the conditions that need to be met in the current world model so the operator can be applied. The second is a description of the effect that the operator has in the current world model in the form of two sets of well-

[1]STanford Research Institute Problem Solver

formed formulas: the ones that need to be added to the world model and the ones that will no longer be met and need to be removed from it. One application of STRIPS was the mobile robot Shakey [Nil84], whose sequence of actions in its upper planning layer mentioned in the previous section were generated by STRIPS, and these plans further refined by feedback with its sensors. Nevertheless, it has been claimed that Shakey only worked well because its environment was very carefully engineered [Bro91b].

Another application of logics to problem solving in robotics can be seen in **PLANNER** [Hew70]. PLANNER uses deductive logical reasoning over a set of statements or assertions about the state of the world together with a set of problem-solving primitives that can be applied to achieve certain goals.

These beginnings grounded the way to more modern and complex logic-based systems aimed for task planning in robots. This is the case of **GOLOG**[2] [LRL+94], a logic programming language based on an extended version of situation calculus and implemented in Prolog that handles a dynamic domain. As a result, programs are able to reason about the current state of the world and changes that could be made. The solution is presented in the form of a linear plan of actions.

GOLOG has been broadly applied and extended in a diverse set of fields. A practical application of it to robotics can be found in RHINO, the interactive museum tour-guide robot deployed in the *"Deutsches Museum Bonn"* [BCF+98]. RHINO uses an extension of GOLOG called **GOLEX**, which increases the functionality of the original system with a hierarchical and conditional plan structure. In addition, it supplies monitoring to the execution of this sequence of actions with time-out mechanisms and default actions added when these time-outs are met, enhancing the robot behavior with default error handling. Later, the same team developed Minerva [TBB+99], another mobile robot designed for touring and entertaining people in public places. Minerva's decision making was implemented in RPL[3][McD91], a language based on Lisp used for reactive planning. This decision making level was built on top of lower-level control structures using GOLEX.

More adaptations and extensions of GOLOG have been done by De Giacomo and Lespérance [DGLL00], who describe an extension of GOLOG called **ConGolog**, for prioritized concurrency and recursive processes monitoring. Additionally, McIlraith and Son [MS02] built a version of GOLOG for their composition of Semantic Web Services to enable users to customize it by adding personal constraints, making it more generic and usable.

[2]GOLOG stands for *"alGOl in LOGic"*.

[3]Reactive Plan Language.

2.1.4 Limitations of logic-based approaches

Although there is a considerable number of authors that support logics as a useful tool for robot task planning and many languages and systems have been developed, logics have several drawbacks:

- **Symbolic representation of the world.** In order to deliver a solution to a problem, a logic-based robot task planning system must have a symbolic representation of the world. Nevertheless, this representation is heavily task dependent, and therefore makes it difficult to have a generic task planner based on it. [Bro90]

- **The Frame Problem.** As McCarthy and Hayes already pointed out [MH69], logic-based approaches force the developer to explicitly state which actions change any value of an object and which actions **do not**. For example, an object's size does not change by picking it up or opening the window. Thus, there are numerous conditions that need to be written down and most likely many to be forgotten by the human writer, since these invariants might not be so obvious for a human mind. Some solutions to this problem have been already proposed. For instance, separation between theorem proving and searching through a space of world models was considered in STRIPS [FN71]. Furthermore, an automated generation of all these frame axioms can be found in GOLOG [LRL$^+$94].

- **Rigurous nature of logics.** Even the simplest of the problems requires a considerable number of statements and predicates in order to accomplish any goal. One example of this can be the *Monkey and the Bananas* problem, in which a monkey must solve the problem of taking some bananas hanging from a ceiling, and for this purpose he can only move a box under the bananas, climb it and pick them. A solution to this problem using a logic-based problem solver is described in [Gre69] and it already involves a reasonable number of axioms and predicates considering the simplicity of the problem.

- **Artificial sense of generality.** While working with logics it is easy to think that at some point it will be possible to have a very general problem solver that could virtually reach a solution to any problem that could be stated in terms of its logic. However, most of the time there is no such generality, since the problem solver reasoning relies on the symbolic representation of the world, which is task dependent. Indeed, problem solvers are usually circumscribed to a certain domain of expertise and within it is where they are useful for [Bro90].

2.1.5 Situated agents: A hybrid approach

Although the two main approaches to solving the task planning problem have been traditionally Finite State Machines and Logics-based approaches, some authors have tried to reach a compromise between both in order to find a balance between their advantages and disadvantages. This is the case of Kaelbling and Rosenschein, who suggest a **situated representation** in [RK95], based on **situated automata theory** [Ros85]. They point out the importance of the relationship between the internal state of an agent and the conditions that are met in its environment. Thus, whereas logics-based approaches state the triggering of a task in an agent in terms of the state of its environment: *when P is true in the agent's environment, then agent should do A*, situated automata theory suggests that the triggering should be executed in terms of the agent's own inner state: *when P′ is true in the agent's inner state, then agent should do A*. The relationship between the agent and its environment is then the problem to be solved. If it is possible to find an inner state P′ that implies the state of the environment P, P′ will be enough to execute action A. Hence a systematic representation of the relationship between P and P′ is needed. Situated agents are therefore a hybrid approach that use traditional automata as a base and try to build a meaningful relationship between it and its environment through logics.

2.2 Semantics in maps and data representation

This section addresses work developed within the scientific community related to maps, semantics and data and knowledge representation.

2.2.1 Introduction

Approximately twenty years of research have made robot 2D mapping a mature field. It already allows robots to develop very detailed maps of complex indoor environments in real-time [Thr02]. Nevertheless, there are still several issues that need to be addressed regarding the ability of robots to interact with their environment.

Therefore, one of the topics this work is intended to cover is the internal representation and understanding that a robot has of its environment. This plays a very relevant role not only in the extension of the set of tasks that a robot can perform, but also in its future as a human companion. The use of robots is every day going further from areas where only trained operators interact with them towards the general public. As a result, robots will be expected to interact and communicate in ways that are easily understandable for humans [VGNS07, GSC$^+$05, MJZ$^+$07].

2.2.2 Navigation-oriented mapping

The start point of the first approaches to mapping and mobile robotics were mathematical theories of space where points (sometimes also lines) are the main primitive spatial entities [CCPR02, Coh99].

Robot mapping has been long time aimed to solve navigation-oriented problems. As a result, there are two categories in which robot mapping has been traditionally classified: **metric** and **topological** [VGNS07, TGBK98]. The first one is only concerned about geometrical features present in the physical environment of the robot. The second one, topological mapping, considers the relationships and connections between areas that allow the robot to navigate from one place to another.

Regarding to the **metric category**, gridmaps [Elf89, Mor88] have been broadly used [BBC$^+$95, BCF$^+$98, Thr02]. Gridmaps are sampled domains where the presence of an obstacle is represented by a confidence value. Further work during the 1990s focused on **probabilistic techniques**, such as SLAM[4][DDWB00] and CML[5][LF00].

Other ways of representing space made regions become the main spatial entity. This is the case of QSR[6], where QR[7] has been applied to space, taking regions as their basic spatial entity. The principal aim of QSR is to offer a theoretical tool to reason and represent space without the need for the usual quantitative approaches present in computer vision or graphics [Coh99]. To this end, it supports ontologies with qualitative topological notions (i.e. graph-like representation) that allow robotic environmental knowledge [CCPR02].

On the other hand, the **topological approach** generates graph-like descriptions of environments where nodes represent places and edges the connections between these areas [TGBK98]. Some authors relate topology to the cognitive representation that humans have of space. Kuipers states in his *Spatial Semantic Hierarchy* (SSH) [Kui00] that topological information, spetially with hierarchical structure, is effective for planning and not necessarily relies on the existence of metric information at the moment of developing the plan, although this metric information could be used later for further improvement of it. Krieg-Brückner *et al.* also use topology as an important layer in the building of their *Route Graph*

[4]Simutaneous Localization And Mapping
[5]Concurrent Mapping and Localization
[6]Qualitative Space Reasoning
[7]Qualitative Reasoning

model [WKBH00]. Here, route-finding problems are approached by a topological representation that consists of *places*, *paths* and *regions*, that are ultimately represented in a graph. Furthermore, another application of the topological approach can be found in [BNK09]. In this case topology is applied to represent complex relationships between 3D objects through the partitioning of rooms into smaller spatial units. These units are called cells and facilitate tasks like route finding.

Either purely metric or topological approaches to solve the navigation problem have been since long time ago recognized to have several drawbacks. Only metric approaches have scalability problems, since their representation and processing usually involves high memory needs and time complexity. On the other hand, purely topological approaches result on difficulties distinguishing between different places [Thr02]. Thrun and Buecken [Thr98] accomplished a successful integration of both approaches based on the world models in [CL85]. This integration was done in two stages, a first topological stage and a second metric mapping stage. In the topological phase a set of *significant places* are discovered and later a metric map is produced.

2.2.3 Usefulness of semantic information

Although the need for semantic information in maps has been addressed for a long time [CL85], the natural tendency in robotic mapping has been to focus on improving metric and topological mapping, making the perception the robot has from its environment very precise and complex [Thr02]. However, as this field has been further developed it has become clear that semantic information should really be taken into account if robots will ever be able to perform a diverse set of tasks.

Several abilities would be improved in a robot by the addition of semantics [GFMGS07]:

- **Inferring new, not-observed knowledge from its environment**. For example, if a robot is located in a bedroom and it knows that bedrooms always contain at least a bed, it can infer that there must be a bed in it, even if it has not been observed yet.

- **Improving human-robot interaction** using a more human-like language. People that are not familiar with computers or robots would find their interaction with these easier when the language they need to use is not very different from the one that they use to talk to each other.

- **Handling errors** by the use of semantic information. Continuing with the previous example, if the robot knows it is in a bedroom and finds a shower it may reach the conclusion that it must be an error, since bedrooms are not supposed to contain showers. Also, this information could even help to correct the mistaken information, and the robot may realize in some way that what was identified as a shower is a fridge instead.

- **Increasing efficiency in task planning by reducing the search space**. One example of this could be that a robot is asked to retrieve a certain object, a jacket. It could have information that states that clothes are usually in a wardrobe and that the wardrobe is in the bedroom, so it would start its search there and would not try to look for it in any of the other rooms, reducing considerably the search space from the whole environment to a single room.

2.2.4 Integrating semantics in maps

In order to be able to process and use semantic information, it must be linked in some way to the metric and topological information of the map built by the robot.

A layered architecture has been suggested by several authors in order to handle this task.

For **Maio and Rizzi** [MR93] each layer corresponds to a meaningful environment abstraction. Here, an intelligent autonomous agent can obtain information from its environment via three different channels: (i) the *metric* channel, which is responsible for the current robot position, (ii) the *visual* channel, which involves all the images a robot can get from its surroundings (i.e. by camera or sonar), and (iii) the *symbolic* channel, which allows tagging points of interest with information that can be obtained for instance by reading a sign or text. This knowledge is then processed and organized in several layers, which include only significant information used for a family of tasks, reducing their processing complexity. Each layer is built from the layer below through a process of *clustering*, where nodes that belong to the same category are mapped to the same node in the upper layers, like it is shown in figure 2.3, where a building example is shown. The top building layer is built by clustering the floor layer, which relies on the room layer, that also is a basis for the clustering that yields the department layer. Every layer depends ultimately on the sensor measures.

Zender, Jensfelt, Mozos et al. [ZJMMB07] also suggest to structure the spatial knowledge in layers, based on a metric map. From it they build a navigation map, establishing a graph-based model of connectivity where nodes are doorways and rooms. Over this layer a topological map is placed, and the highest layer of the hierarchy corresponds to a conceptual map. Each element in the topological layer is linked to a category in the conceptual map.

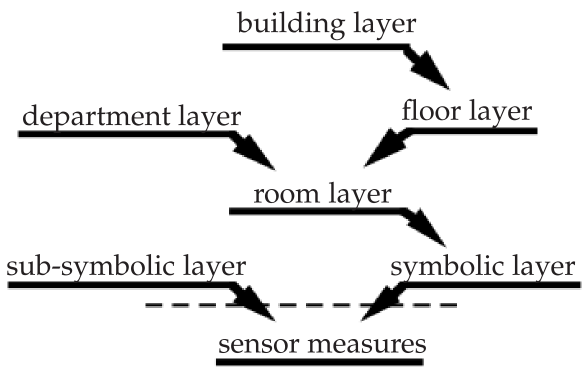


Figure 2.3: Example of a layered architecture proposed by Maio and Rizzi [MR93]. Each layer is built from the lower layers by a process of clustering.

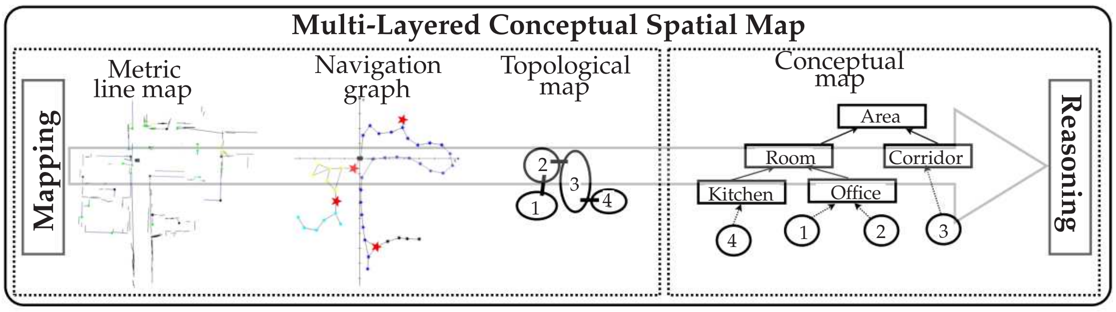
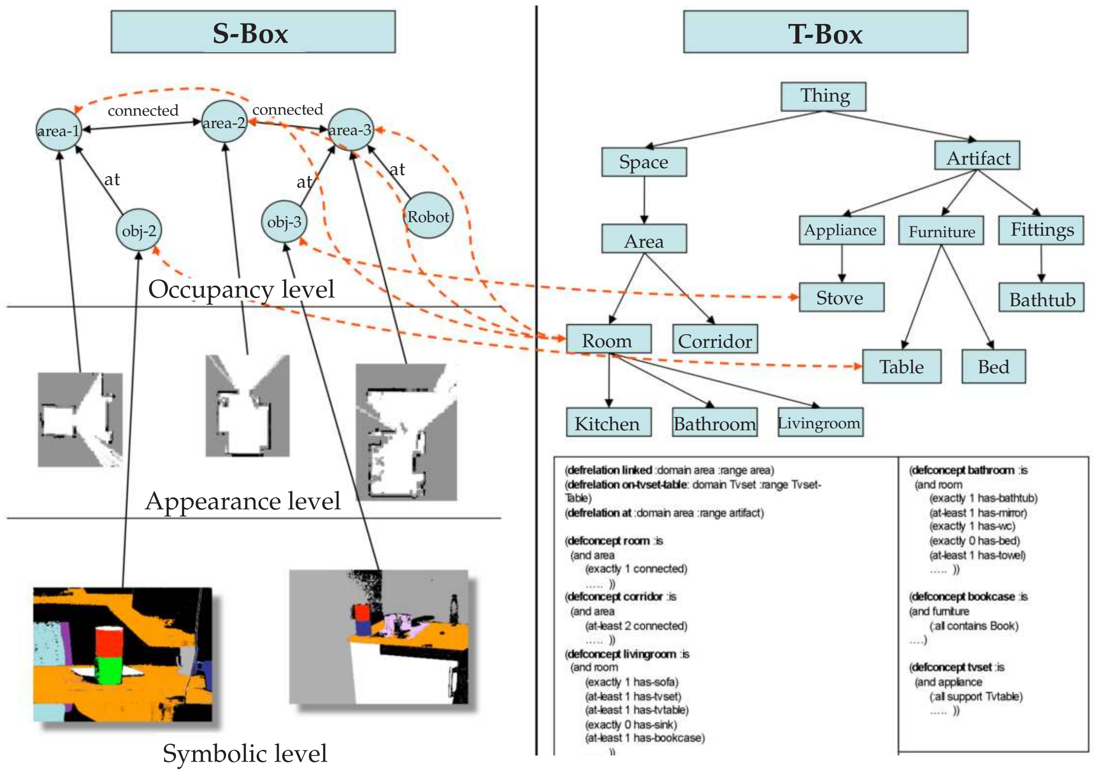


Figure 2.4: Layered architecture for knowledge representation proposed by Zender, Jensfelt, Mozos *et al.* [ZJMMB07]

Galindo et al. [GFMGS08] propose to separate the semantic contents in two highly related parts called *Spatial Box* (or *S-Box*) and *Terminological box* (or *T-Box*). The first part contains information about the actual state of the environment, connecting metric information (e.g. mapping and camera images) to symbolic representations that are part of a topological graph. These symbolic representations are associated to a category in the *T-Box*, which contains general knowledge in form of concepts and relations between them. On further work, Zender, Jensfelt, Mozos et al. also refer to S-Box and T-Box as a way of differentiation between general knowledge about the world and spatial information [ZMJ+08]. As it is shown in figure 2.5, the symbolic level captures information about the visual appearance of objects, that are linked to a node in the *S-Box*. This node, which also has a symbolic name (e.g. obj-2, obj-3) is situated in some space unit that also includes information about topology (i.e. obj-2 is in area-1, which is conected to area-2). These instances in the *S-Box* are connected to a category in the *T-Box*, from which semantic information can be obtained.



```
(defrelation linked :domain area :range area)
(defrelation on-tvset-table: domain Tvset :range Tvset-
Table)
(defrelation at :domain area :range artifact)

(defconcept room :is
  (and area
      (exactly 1 connected)
      ..... ))
(defconcept corridor :is
  (and area
      (at-least 2 connected)
      ..... ))
(defconcept livingroom :is
  (and room
      (exactly 1 has-sofa)
      (at-least 1 has-tvset)
      (at-least 1 has-tvtable)
      (exactly 0 has-sink)
      (at-least 1 has-bookcase)
      ..... ))
```

```
(defconcept bathroom :is
  (and room
      (exactly 1 has-bathtub)
      (at-least 1 has-mirror)
      (exactly 1 has-wc)
      (exactly 0 has-bed)
      (at-least 1 has-towel)
      ..... ))

(defconcept bookcase :is
(and furniture
      (:all contains Book)
....)

(defconcept tvset :is
  (and appliance
      (:all support Tvtable)
      ..... ))
```

Figure 2.5: Galindo *et al.* semantic representation [GFMGS08]. Each symbolic representation in the *Spatial Box* is linked to a category in the *Terminological Box*.

2.2.5 Knowledge structure

Whether it corresponds to a layer in an architecture or not, the semantic information itself must be structured in a certain fashion. Ontologies and description logic are broadly used in this topic [GFMGS08, MJZ$^+$07]. Ontologies are shared specifications of conceptualizations of a domain [Gru93]. They can be defined as a formally specified model that depicts a domain of knowledge and the relations between the concepts within it [CCPR02].

Further domain information can be added by description logics. This knowledge about a certain domain is therefore structured as an acyclic graph where concepts are related to their super- and sub- concepts, making it possible to infer new knowledge through a reasoner. OWL-DL has been used in [ZMJ$^+$08] as a language to implement this knowledge.

2.2.6 Acquisition of semantic knowledge

Once the knowledge is structured in a certain fashion there are several ways in which a robot can acquire this information. There are already numerous works about automated ways to obtain semantic information about the robot's environment that encompass fields of study such as scene interpretation or object recognition.

These techniques can be also mixed with human interaction in order to obtain a semi-automated mapping process, like the one described by Diosi, Taylor and Kleeman in [DTK05]. Here, an occupancy map is created at the same time the robot is following an operator around the environment. The operator gives the robot comments regarding the current location and the robot generates annotations in the map accordingly to these verbal sentences.

These automated and semi-automated approaches to the acquisition of knowledge are however out of the scope of this master thesis. Nevertheless, they can be added to the resulting system as further work. The current work focuses on the semantic structure itself and tries to make it as general as possible. Regarding this process of creation of semantic knowledge, human intelligence is necessary for it and it cannot be fully generated in an automatic way [SS09].

2.3 Natural Language Processing

This section describes briefly the work done in the field of human-robot interaction regarding natural language processing.

2.3.1 Introduction

The way humans interact with machines and in this case with robots is evolving everyday towards a more human-friendly communication. This is usually referred to as natural interaction and encompasses a broad set of different fields such as gesture recognition and natural language processing. A fluent dialogue that resembles the natural language as much as possible has been suggested [MJZ$^+$07] as a powerful tool to improve human-robot interaction.

2.3.2 Human-computer interaction cycle

Traditionally, human-computer interaction has been decomposed in several processing stages that, including the user, form what is called the *interaction cycle* [FM09]. As shown in figure 2.6, first the system captures the user input and processes it. In the case of a speech-based system it would be a microphone

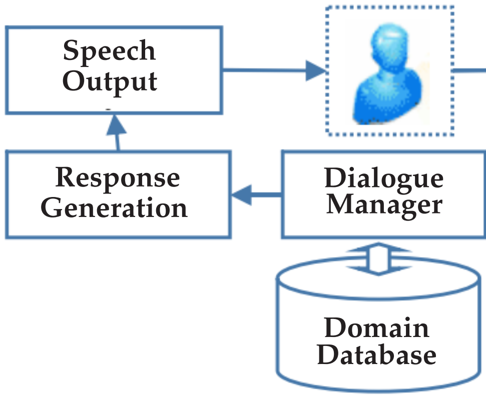


Figure 2.6: Human-Computer interaction cycle by Filipe and Mamede [FM09].

that records the user's voice, which would be processed afterwards with some *Speech Recognition* component. This component sends the recognized words to the *Language Understanding* component and it sends the related speech acts to the *Dialogue Manager*. With this information and some domain database information it calls the *Response Generator* and it uses a *Speech Output* module in order to generate sound.

2.3.3 Language parsing and understanding

Once the Speech Recognition module has done its work and parsed the user's speech, the result is a sentence that needs to be processed in order to produce an action or an answer. This problem belongs to the field of *natural language processing* and it has been addressed for many years.

One decision that needs to be made when addressing this problem is whether the goal is to produce human-like conversation or the user's input should be *restricted* in order to simplify the language processing. In the first case the choices that the user could make are *expanded* during the dialogue [ABD$^+$00]. This case has been often related to the concept of intelligent machine. Alan Turing formulated this point of view with his famous *Turing Test*, where a machine would be considered intelligent if an interrogator can mistake it for a human when communicating with the machine via teletype [Tur50]. Although this definition of intelligence is controversial and also is the Turing Test itself [Fre90], the fact is that whether a machine would be considered intelligent because of its language skills or not, it is true that having a good language understanding ability would improve human-robot interaction [MJZ$^+$07].

However, specially regarding speech-based dialogue systems, a restriction in the dialogue possibilities would enhance the performance of both speech recognition and language processing components. This does not mean that the accepted language should be too limited. For example, menu-based dialogue systems are usually frustrating for its users. By any means, this input language should be general enough to enable the user to express his goals in the most natural way possible [ABD+00].

2.3.4 Models and algorithms

Over the last 50 years, many theories have been developed to enable computers to work with natural language. Coming from the usual computer science tools one can found plenty of models and theories well-known for every computer scientist.

A set of tools including deterministic and non-deterministic state machines, formal rule based models, such as context-free and regular grammars and logics are the most spread and long-lived ones. These led to algorithms such as state space search and dynamic programming algorithms. Including probability these basic tools can be transformed into weighted automata, Markov models and hidden Markov models considering the first ones, and also stochastic or probabilistic context-free grammars, in which each production is augmented with a probability.

Usually these systems include a search through a space of possible states regarding an input, and to this end there are many well-known graph algorithms such as A* search, depth-first and best-first.

In addition, regarding logics there is also a wide set of familiar tools that, among others, include first order logic and predicate calculus [JM00].

Chapter 3

Integrating Task Planning and Semantics

This chapter depicts a solution proposal that encompasses both a theoretical approach and its practical development. It is structured in four main sections. Section 3.1 shows a system overview to give the reader an idea of the whole system. The following three sections describe the main parts of the software in detail. A solution to *task planning* is discussed in section 3.2. Section 3.3 shows the solution to *maps and semantics* and section 3.4 explains how the problem of *string command extraction* is addressed.

3.1 Overview of the system

This section describes the main parts involved in the system and how they interact in order to perform the intended task.

As it is shown in figure 3.1, the input to perform a task is expected to be in form of a string. It is a sentence that represents the sequence of tasks that need to be carried out. The structure of this sentence and the processing of it by the **Simple Command Extractor** is described in detail in section 3.4. Once this input has been processed, a `TaskInfoM` message is sent by this module to the **Task Planner**. This message contains the necessary information to describe these tasks. This will be performed and monitorized in a sequential way by the Task Planner, which uses simpler tasks that are properly wrapped and offer a generic message interface. This interface consists of messages that allow this Task Planner to start, stop, restart and check their status. Wrapping and task planning are described in detail in section 3.2. Sometimes, in order to perform some tasks, a certain knowledge about the environment is needed. This semantic information is represented in the **Domain Reasoner** and in the

Semantic Map Layers. These correspond to the T-Box and S-Box described in [GFMGS08], respectively. Section 3.3 describes the structure of both in detail. The semantic information that can be found in the map is added manually by an operator that interacts with the GUI.

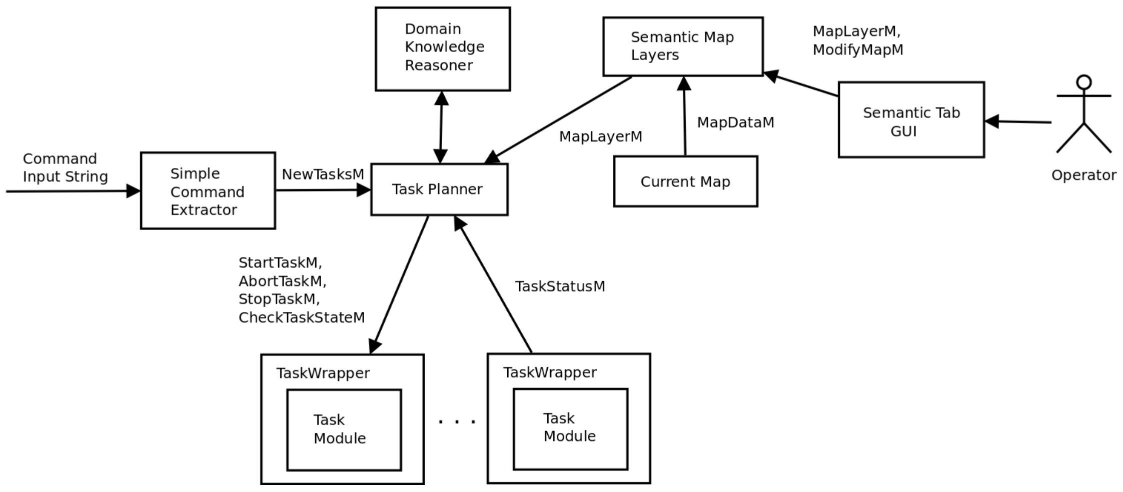


Figure 3.1: System overview. Interaction between all parts of the software.

3.2 Task planning

This section includes a description of the abilities Lisa already has regarding task planning and some discussion about their usability. Furthermore, an explanation of a theoretical approach to the proposed task planning and a description of its software development is provided.

3.2.1 Current state of task planning

At this moment, Lisa does not have task planning in a way that allows developers to freely use current implemented modules and develop new abilities that involve already acquired ones.

Everytime a new module has to be designed a considerable amount of new code, that most of the time is repeated somewhere in another module, is produced. As it is shown in figure 3.2, the reason for this is that modules are conceived as glue code. Thus, every module is handcrafted to hold together different parts of the project that are properly encapsulated. This course of action implies not only many hours of work to create new modules but also results that are difficult to maintain.

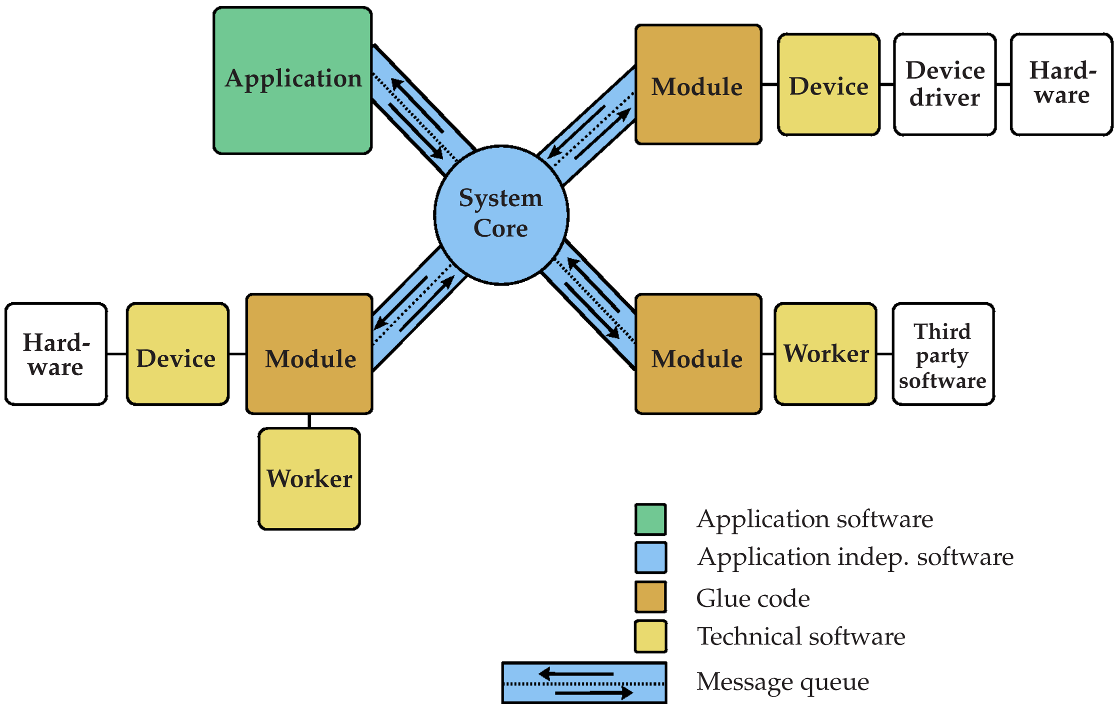


Figure 3.2: Lisa's software architecture.

Regarding **RoboCup@Home** games [NDFD+11] it entails several problems. Indeed, these games are usually a composition of already implemented tasks and they should be easy to manage as long as there is already a main set of abilities implemented. Unfortunately, this is not always the case, since there are many messages that need to be sent and states that need to be checked just to achieve a simple already implemented task (e.g. grabbing an object). This problem, together with a lack of documentation about how the modules themselves work, leads the programmer to multiple hours of reading and understanding unknown (but already working) code, copying what is needed to other modules and making the amount of code that needs heavy maintainance increase everyday.

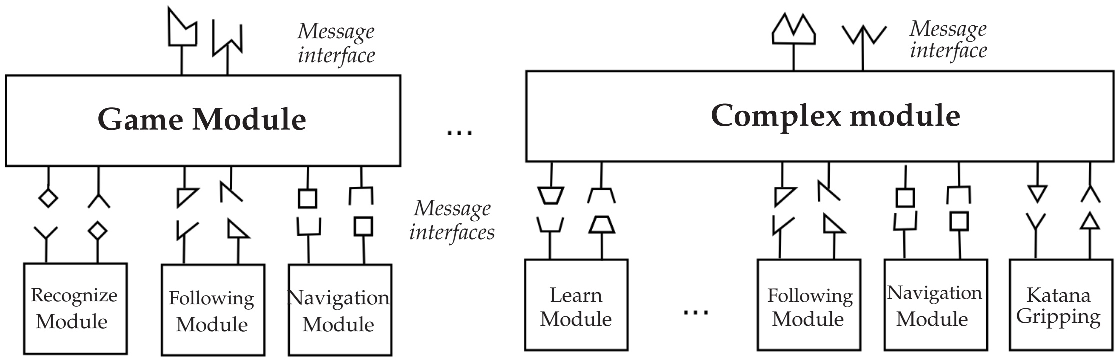


Figure 3.3: Representation of the current interaction between modules. Since every interface is different, developers need to handcraft the specific interaction for the needed modules.

It could be argued that this is not a real problem because @Home games are a fixed set of tests that do not have considerable changes from one year to the next one. Accordingly, after some time of development they would be ready and working so the need for maintainance would be low. Nevertheless, there are at least two games every year that need new development:

- **Open Challenge**. It is an open game where every team shows different features that make their robot worthier.
- **Demo Challenge**. A semi-open challenge where some topic is fixed and teams show their creativity applying the capabilities of their robot to this fixed topic.

In addition, when a lot of time is invested for these modules, there is not much time left for improving already existing capabilities or developing completely new ones.

The initial interaction between Lisa's modules can be seen in figure 3.3. There, every message interface is different (represented by different shapes in the offered and required interfaces). A rather relevant concern about this design is what happens if one of the basic modules needs to change its interface. If this ever happened, this interface would need to be changed in every single module that uses it, generating a considerable amount of work and probably a source of future errors.

Besides, a household robot's abilities should be general and it should be possible to combine them in order to enable a user to interact with it and obtain an efficient service. This is the aim of **General Purpose** game, where a robot is given a complex command (i.e. *Go to the kitchen, take the cup and bring it to Mike*) and must process it, break it down into simpler tasks and perform them.

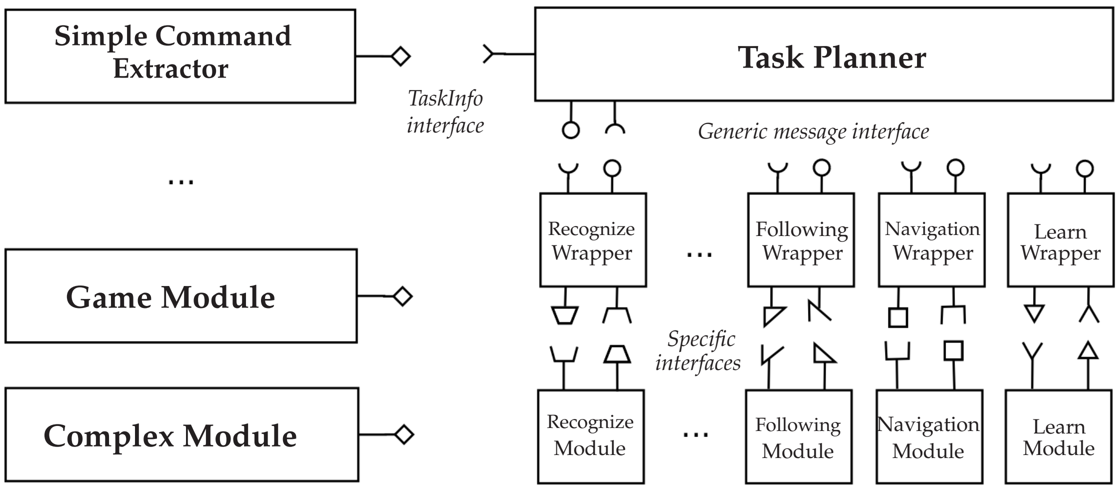


Figure 3.4: Representation of the proposed interaction between modules. Wrapper modules abstract the different message interfaces offered by the basic modules and offer the same interface to their external side. The Task Planner this way is only dealing with these messages and the TaskInfo messages sent by the rest of the modules.

According to the current state of the project, this means a module of considerable size where most of the code is copied from basic modules (i.e. navigation, grabbing). This module is therefore difficult to maintain and test. Until now, our @Home team has not earned any points at RoboCup@Home competition with this module.

Considering these reasons, an intermediate abstraction layer is proposed. As it is shown in figure 3.4, each basic module will interact with a *wrapper module*. These *wrappers* will be the ones that make an abstraction offering all the same generic message interface to upper layers. Moreover, there will be a task planner module that will have a control position over these modules, being able to perform complex tasks that involve arbitrary combinations of the basic ones. Ideally, any new module would only need to interact with this task planner through a defined interface that would not change. This would increase the system maintainability and scalability. In the case mentioned before, where one of the basic modules changed its interface, only its wrapper would need to be changed, making the impact on the whole system much slighter than before. Regarding scalability, whenever a new basic ability is implemented, it would only need a wrapper and some minor changes to be used with the task planner.

3.2.2 Task planning approach

The first choice that needs to be made is which kind of approach is most suitable for the problem that intends to be solved. According to this, the decision made has been to continue with the current finite-state machine behavior and add an abstraction level that enables developers to easily use implemented tasks as encapsulated modules.

One reason is an **incremental approach** point of view. Iterative and incremental approaches to software development have been used since long time ago [LB03]. Besides the fact that Lisa's project is already running software that grows naturally in an incremental way, there are several advantages that need to be taken into account in order to understand why it is indeed a good choice [LB03]:

- **Verification of the software**. When the development is done in an incremental way, each iteration is easier to verify and test than a whole system developed through a traditional waterfall model.
- **Learning from experience**. Developers have the opportunity to apply what they have learned while developing previous versions of the system. This makes the quality of the software and the effectiveness of the development increase along time.
- **Smalls steps at a time**. Taking small steps each time minimizes the risk. If the step taken is discovered to be wrong or not suitable it is possible to go back to the previous version without a considerable impact to the project.

It seems therefore reasonable to think that if it is possible for the software to grow in a consistent way, an incremental approach is a suitable one. To this end, the aim of this work has been focused on building a steady abstraction layer that enables further iterations to easily improve and enhance the system with new features.

In addition, household robots are not expected to do high-level reasoning about the world yet. They are rather needed to perform already well-defined tasks efficiently in a benevolent environment [Bro91b]. According to this, sometimes reasoning can make the system slower in a way that the robot is not able to give an answer in a reasonable amount of time. On the other hand, the finite-state machine approach seems to provide a faster solution [Bro91b].

This does not mean that robots will never be expected to do complex reasoning about the world. This is the reason why it is necessary that this work provides a steady structure on which further development can be added in the future. This basis is strengthened by adding semantic knowledge about the environment. However, this will be discussed in detail in the next section.

3.2.3 Abstraction of tasks and wrapping.

In order to encapsulate the current tasks, there are several things that need to be done. It is necessary to understand what a task exactly is and which information is needed in order to represent it.

First of all, a task consists, directly or indirectly, of different parts of the already running software that will be addressed in this work as *devices*, *workers* and *skills*[1].

Device. Devices are class abstractions that handle the necessary interaction with the hardware components from the robot and enable the rest of the software to interact with them through an interface.

Worker. Workers are classes that encapsulate basic computation and algorithms needed for higher level modules.

Skill. A skill is an implemented ability that allows a robot to achieve some goal but does not necessary have a meaning to the end user. A skill is implemented in a *module*, uses *devices* and *workers* when necessary and exchanges messages with other *modules*. Skills include abilities like:

- Path planning.
- Face detection.
- People detection.
- Speech recognition.
- Arm planning and gripping.
- Grippable object detection.

A repeated use of the software and further development of @Home games in the project has made it clear that there are certain sequences of skills that are frequently used together. This happens because they are meaningful whole units that encompass a feature that would most likely be a requirement for an end user. As a result the definition of Task appears naturally:

[1]Note that the definition of *skill* is made to help the reader to understand the concept of task. However, whereas *workers* and *devices* are explicitly defined as such in the current framework, the entities addressed as *skill* are just *modules* or *@Home games*.

Task. A task represents a meaningful action that is performed by a robot and solves an @Home problem. This includes any of the usual abilities that a household robot shows and that could be interesting as a unit for an end user. Therefore, there is here a higher level of abstraction from skills, and more than one of them could be included in one task. The currently available tasks are:

- **Navigation**. Driving to some Location.
- **Grabbing**. Taking an object.
- **Releasing**. Dropping or giving an object.
- **Bringing**. Carrying one object to a place.
- **Learning**. Learning the face of someone or the appearance of an object.
- **Recognizing**. Recognizing someone or something previously learned.
- **Talking**. Saying a sentence or sequence of them.
- **Waiting for an answer**. Waiting for a specific sentence.
- **Following**. Driving after an operator.

For instance, the task following includes several skills that are already implemented, like detecting persons, tracking them and path planning. These skills eventually use devices and workers as well.

This set of tasks can of course be extended with more that include new developed skills or that represent new needs discovered in the end user.

Since a task is an abstraction that will allow higher level software to interact with these actions in a general way, there is the need to define what information is required to perform a task, which should be common to every type.

Taking as a guide the currently identified set of tasks, it is clear that any action is performed *somewhere*. Therefore, a `Location` is always needed in order to represent a task. In addition, some of these tasks have a target. This target can be either a person or an object that is somewhere in the world. Additionally, regarding the case where there is speech involved, text is needed. In order to match the naming policy of the already implemented software, every class that represents a running task ends with the name `Module`.

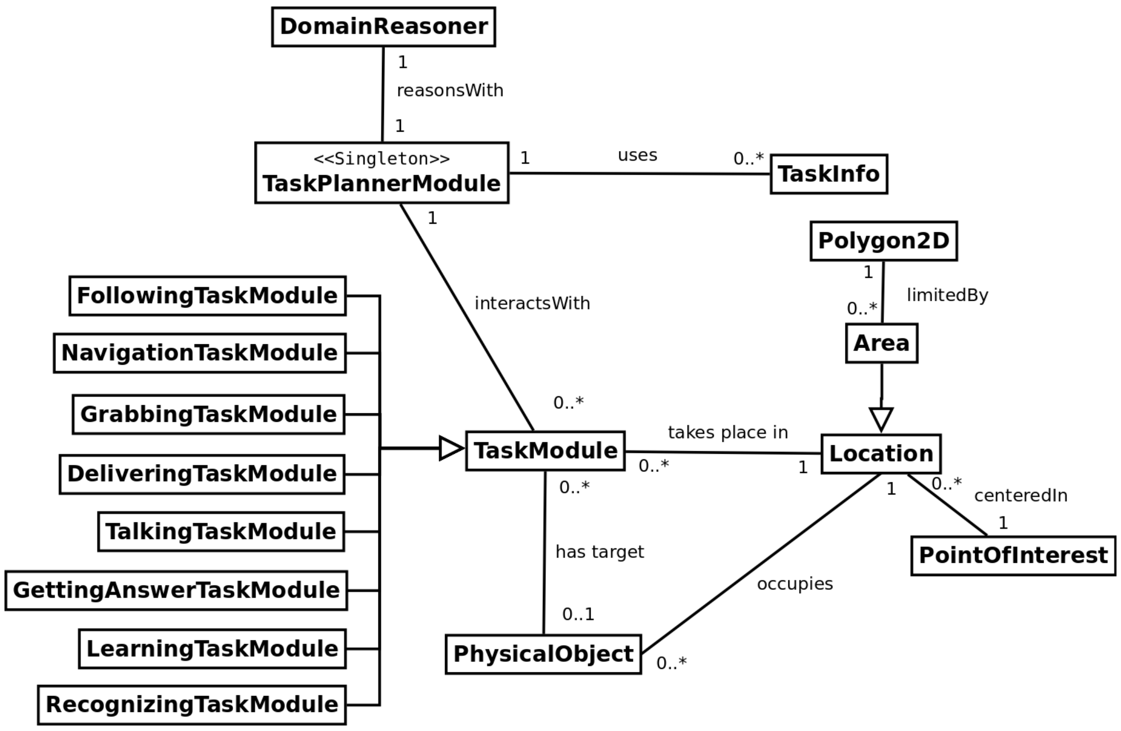


Figure 3.5: Task design class structure. Methods and attributes of each class are discussed in the text and a detailed implementation diagram is shown in section 3.2.6.

To be able to talk about `Locations` and `PhysicalObjects` it is also necessary to define what they are.

A `Location` is a specification of space within the map. As shown in figure 3.5, it can be just a point (the Worker class `PointOfInterest`, which is already in the project, is used) or an `Area`. This is one of the ideas that have been added to the mapping concepts. Subsequently, now it is possible to specify areas in the map and these will not be limited to points. One reason for this is that the @Home arena is always partitioned in rooms, being the identification of them in the map more useful when they are regions and not just points. One example of this is the case where a navigation task to a room is started, but the robot is already standing in it. At the moment there is not a possible way to know this, whereas it would be possible if rooms were specified as areas instead of points. Moreover, some objects that need to appear in the map occupy a significant amount of space that is obviously more than just a point.

The *point-in-polygon* algorithm

Regarding the situation described in the previous subsection, where it is necessary to know whether a `PointOfInterest` is part of a `Location` or not, the virtual method `belongsTo` has been implemented. When a `Location` is a point, it calculates whether the distance to it is less than a threshold that can be passed as a parameter and whose default value is 0. When the `Location` is an `Area`, it checks whether the point is within it or not. In order to ascertain this, the algorithm *point-in-polygon*, also known as *even-odd test* has been used. The algorithm is shown in algorithm 1. Roughly, a horizontal scanline through the target point is drawn and its intersections with the edges of the polygon until reaching the target point are counted. When this number is *odd*, the point is inside. There are some exceptions that are omitted in this count. These exceptions are that the current edge is horizontal (therefore parallel to the scanline) or that the intersection point is one end point of the edge (thus, would be counted twice and the result would not be the expected). A more detailed discussion about this broadly used algorithm can be found in [Hec94].

Algorithm 1 Point-in-Polygon algorithm.

```
1: Given: polygon $P$, point $X$
2: Output: boolean value. **true**if $X$ is inside $P$, **false**otherwise.
3:
4: $l \leftarrow 0$
5: **for all** $m$ edge in $P$ **do**
6:    $h \leftarrow$ horizontal through $X$
7:    $s \leftarrow$ intersection point between $h$ and $m$
8:    $sx \leftarrow$ x coordinate $s$
9:    $xx \leftarrow$ x coordinate $X$
10:   **if** $m$ is not horizontal **and** $h$ cuts $m$ **and** $s$ is not an end of $m$ **and** $sx \leq xx$
      **then**
11:      $l \leftarrow l + 1$
12:   **end if**
13: **end for**
14:
15: **if** $l$ is odd **then**
16:    **return true**
17: **else**
18:    **return false**
19: **end if**
```

Regarding `PhysicalObject` objects, they can be either persons or things that are target of the action defined in the task. The information needed to characterize a target object is therefore a name, the `Location` it occupies and the `Category` it belongs to. Semantics, categories and their relation with the class `PhysicalObject` are discussed in more detail in section 3.3.

3.2.4 Generalization of the messages

Although Tasks are an abstraction made to be able to work with actions in a general way, they are still threads running in the system and the way in which the communication with them is established is through messages. To this end, there is also the need to generalize the messages used.

Considering that a task is an action performed by a robot, the following messages along with their possible parameters[2] have been considered:

- **`StartTaskM`**. It makes a task start. Its parameters are a `Location`, a `PhysicalObject` and some text when necessary.
- **`RestartTaskM`**. Restarts the requested task. It is expected that a task that is requested to restart already has its parameters set and will try to repeat its action from the beginning.
- **`TaskStatusM`**. Contains information about the current state of the task. Its main parameter is an enumerate whose value can be: `NOT_DEFINED`, `NOT_STARTED`, `RUNNING`, `ERROR`, `SUCCESS`. In addition, an extra `info` field has been added to include some more information to enable the `TaskPlanner` to make decissions when necessary.
- **`CheckTaskStatusM`**. Request for a `TaskStatusM`. When the `TaskType` is not assigned, all the Tasks will answer even when they are not executing.
- **`StopTaskM`**. Stops the execution of the desired Task.

3.2.5 State machine specification

As it has been discussed before, the decision made to handle task planning is the further use of finite-state machines. Consequently, each task must have a defined set of states. Indeed, each task has the same set of states and all of them

[2]All of them include a `TaskType` parameter that indicates to which task it is related. For simplicity, this has been omitted. The assumption that only one task from one type is performed at a time has been made.

are defined in the parent class. This is what is included in the `TaskModule` and shown in figure 3.6.

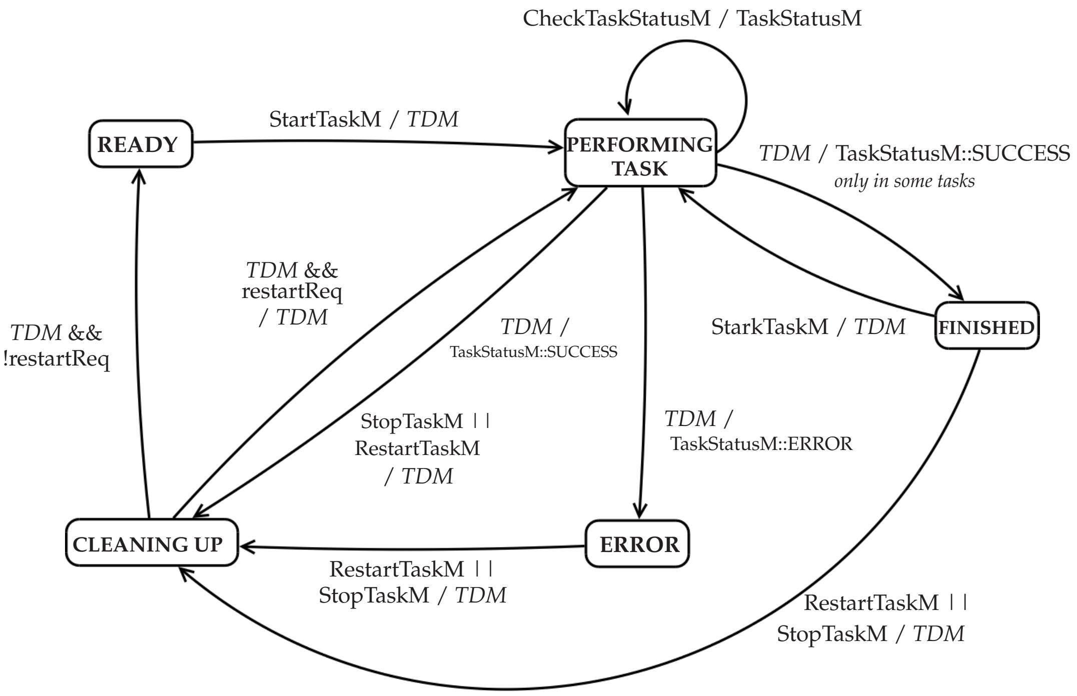

CheckTaskStatusM / TaskStatusM
READY
StartTaskM / TDM
PERFORMING TASK
TDM / TaskStatusM::SUCCESS
only in some tasks
TDM && restartReq / TDM
TDM && !restartReq
TDM / TaskStatusM::SUCCESS
StarkTaskM / TDM
FINISHED
StopTaskM || RestartTaskM / TDM
TDM / TaskStatusM::ERROR
CLEANING UP
ERROR
RestartTaskM || StopTaskM / TDM
RestartTaskM || StopTaskM / TDM

Figure 3.6: State diagram for the state machine present in each task. This set of states is shared by every task. The acronym *TDM* stands for Task-Dependent Message, which depends on the task that is wrapped.

There are five possible states:

- **`READY`**. A task is in this state when waiting for a `StartTaskM`.
- **`PERFORMING_TASK`**. The task is running normally.
- **`ERROR`**. Something is not working and the task cannot continue.
- **`CLEANING_UP`**. The cleaning up is being carried out. This process can include setting the robot devices to initial positions when necessary and also resetting inner attribute values.
- **`FINISHED`**. The task has finished successfully. Only tasks that depend on other tasks after them stay in this state. For instance, in the case that there has been a request to grab an object and bring it somewhere, the

second task depends on the success of the first, and the first cannot be ready again until the second has finished (i.e. the Katana arm cannot go to initial position until the other task is performed). This control is left to the `TaskPlannerModule`.

Regarding the transitions between states, a task goes from the `READY` state to the `PERFORMING_TASK` state when a `StartTaskM` message is received. It will stay in this state until the task is performed or an error occurs. In this transition, a `TDM`[3] is sent to activate the underlying modules that are encapsulated by this task. During the time that the task stays in this state, it can receive a `CheckTaskStatusM` message, that will be answered by a `TaskStatusM` to inform the task planner[4] with the current status of the task. At some moment during the execution of the task, it will receive a `TDM` informing from success or a result, depending on the type of task. In this case, it will go to the `CLEANING_UP` state (or the `FINISHED` state in some cases) while sending a `TaskStatusM::SUCCESS` message to inform the `TaskPlannerModule`. Whenever a task performs the transition to `CLEANING_UP` state, it performs a general procedure that includes resetting the values of its attributes to valid ones that allow it to start again when necessary. On the other hand, it is possible that something does not work properly in the underlying module and a `TDM` meaning error is received while the task is still running. In this case, it would go to `ERROR` state and send a `TaskStatusM::ERROR` message to the `TaskPlannerModule`. Another possibility is that no `TDM` is received at all because some underlying module is waiting as well, frozen or in a not-reported error state. A task will also go to a `ERROR` message in this case, only when it is defined as *timed* and the specified timeout is gone[5]. In addition, from any of the mentioned states it is possible to receive a `StopStaskM` that will make the task to perform its `CLEANING_UP` routine and be `READY` again, and also a `RestartTaskM` that will start over again with the performance of the task with the same parameters as it was started before.

3.2.6 Implementation of the tasks

This subsection explains in detail the key points in the implementation of each one of the tasks. First, there is a description of the general features shared by all of them and their implementation in the general `TaskModule` class. Then, the specific features in each task are shown. In figure 3.7 a detailed implementation class diagram is shown. For simplicity, setter and getter methods and methods

[3]Task-Dependent Message.
[4]In this text. task planner always refers to the `TaskPlannerModule`
[5]The timing of the tasks will be discussed in the implementation

that are inherited from the `ActiveMessageModule` class already in the project are omitted[6].

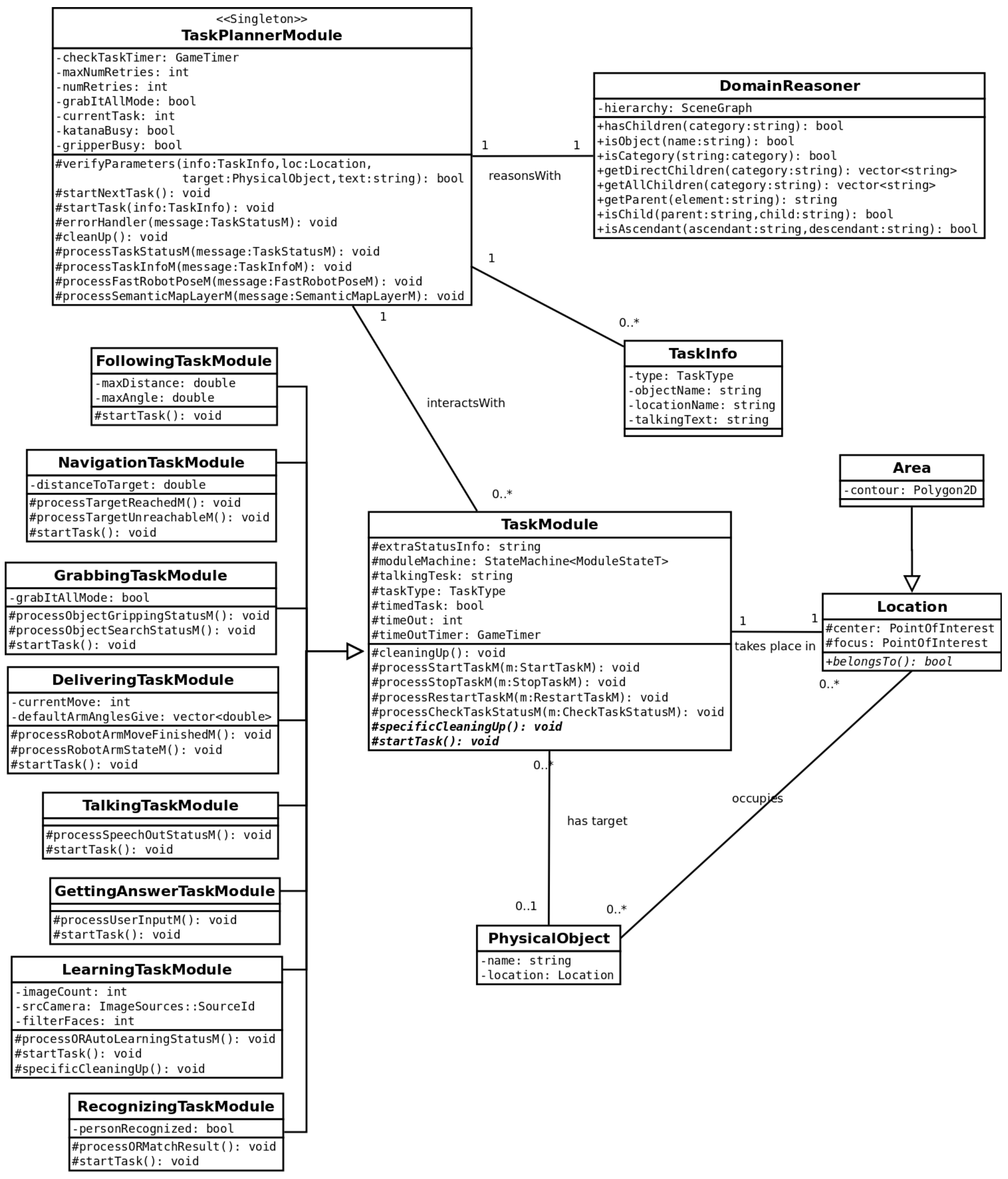

<<Singleton>>
TaskPlannerModule
-checkTaskTimer: GameTimer
-maxNumRetries: int
-numRetries: int
-grabItAllMode: bool
-currentTask: int
-katanaBusy: bool
-gripperBusy: bool
#verifyParameters(info:TaskInfo,loc:Location, target:PhysicalObject,text:string): bool
#startNextTask(): void
#startTask(info:TaskInfo): void
#errorHandler(message:TaskStatusM): void
#cleanUp(): void
#processTaskStatusM(message:TaskStatusM): void
#processTaskInfoM(message:TaskInfoM): void
#processFastRobotPoseM(message:FastRobotPoseM): void
#processSemanticMapLayerM(message:SemanticMapLayerM): void
DomainReasoner
-hierarchy: SceneGraph
+hasChildren(category:string): bool
+isObject(name:string): bool
+isCategory(string:category): bool
+getDirectChildren(category:string): vector<string>
+getAllChildren(category:string): vector<string>
+getParent(element:string): string
+isChild(parent:string,child:string): bool
+isAscendant(ascendant:string,descendant:string): bool
1
1
reasonsWith
1
0..*
TaskInfo
-type: TaskType
-objectName: string
-locationName: string
-talkingText: string
1
interactsWith
0..*
FollowingTaskModule
-maxDistance: double
-maxAngle: double
#startTask(): void
NavigationTaskModule
-distanceToTarget: double
#processTargetReachedM(): void
#processTargetUnreachableM(): void
#startTask(): void
GrabbingTaskModule
-grabItAllMode: bool
#processObjectGrippingStatusM(): void
#processObjectSearchStatusM(): void
#startTask(): void
DeliveringTaskModule
-currentMove: int
-defaultArmAnglesGive: vector<double>
#processRobotArmMoveFinishedM(): void
#processRobotArmStateM(): void
#startTask(): void
TalkingTaskModule
#processSpeechOutStatusM(): void
#startTask(): void
GettingAnswerTaskModule
#processUserInputM(): void
#startTask(): void
LearningTaskModule
-imageCount: int
-srcCamera: ImageSources::SourceId
-filterFaces: int
#processORAutoLearningStatusM(): void
#startTask(): void
#specificCleaningUp(): void
RecognizingTaskModule
-personRecognized: bool
#processORMatchResult(): void
#startTask(): void
TaskModule
#extraStatusInfo: string
#moduleMachine: StateMachine<ModuleStateT>
#talkingTesk: string
#taskType: TaskType
#timedTask: bool
#timeOut: int
#timeOutTimer: GameTimer
#cleaningUp(): void
#processStartTaskM(m:StartTaskM): void
#processStopTaskM(m:StopTaskM): void
#processRestartTaskM(m:RestartTaskM): void
#processCheckTaskStatusM(m:CheckTaskStatusM): void
#specificCleaningUp(): void
#startTask(): void
Area
-contour: Polygon2D
Location
#center: PointOfInterest
#focus: PointOfInterest
+belongsTo(): bool
1
1
takes place in
0..*
occupies
0..*
has target
0..1
0..*
PhysicalObject
-name: string
-location: Location

Figure 3.7: Implementation class diagram for the TaskModule related classes.

[6]For more information about the implementation details, see the doxygen documentation attached to this text.

`TaskModule`: The parent class

One of the main goals of the development of this part of the software during its whole design and implementation has been to keep the extensibility and maintainability to the maximum possible. In terms of implementation, keeping as much code as possible in the general `TaskModule` is good for both, since there is not repeated code and the amount of implementation needed to add a new `TaskModule` is reduced.

According to this, the transitions between the states explained in subsection 3.2.5 subsections as well as the subscriptions to the generic messages[7] are implemented here. Furthermore, a general cleaning up protocol is implemented here as well and called when necessary from any `TaskModule`. From this cleaning up protocol it is possible to add some specific cleaning up for the task dependent variables, but the general behavior is coded in the parent class.

There are only few things that need to be implemented when adding a new `TaskModule`, and these are related to the `TaskDependentMessages` mentioned before. Each of these tasks needs to subscribe to messages that depend on the underlying modules and handle them as necessary. The only `TDM` needed to be sent by each TaskModule when a `StartTaskM` is received is encapsulated in the virtual method `startTask` which is called by the `processStartTaskM` present in the parent class. Even in this case, there are some common things that are done that are also shared by every task. Therefore, the only part that is implemented is the sending of the message. The rest belongs also to `TaskModule` module.

Another relevant detail in the implementation is that although the subscription to and processing of this generic messages is done in the `TaskModule` class, the reception of them is done in each of the task wrappers.

The monitoring of the underlying modules' execution is also a job done by each task wrapper, which sends the necessary `TDM` messages and throws `TaskStatusM` messages containing the necessary information for the task planner to monitorize it.

Considering the possibility that some of these underlying modules become stuck at some point in their execution, subsquently stopping the execution of the task wrapper, which waits for a status message from them, there is the possibility of setting a timeout and establishing any type of task as timed. When a task is timed, a timer starts when transitioning to the state `PERFORMING_TASK` and checked in the idle process of the thread. When the timeout is reached, the task transitions to the `ERROR` state and sends a `TaskStatusM` message with `TaskSTatusM::ERROR` value to the `TaskPlannerModule`.

[7] `StartTaskM`, `StopTaskM`, `RestartTaskM`, `CheckTaskStatusM`

FollowingTaskModule

Description. Among all tasks, following is the only one that has not a predefined end, but waits until an operator asks the robot to stop following. Tasks are expected to monitor their own execution in the meaning that they control the underlying modules and receive status messages from them, but they are not designed to interact with the user or with other modules without the help of a `TaskPlannerModule`. The reason of this is avoiding conflicts between modules that subscribe to or broadcast the same kind of messages. Thus, the communication with the user and the sending of a required `StopTaskM` message by the `TaskPlannerModule` will be explained in detail in the next subsection.

Sent messages: `TaskStatusM`, `StartFollowingM`, `StopFollowingM`.

Subscribed to messages: None[8].

GettingAnswerTaskModule

Description. This module waits until it receives a specific input. Experience has shown that speech recognition is far from working perfectly because of many circumstances (e.g. environmental noise, microphone settings, speech recognition software) that can make the robot unable to understand what an operator said. Hence it is by default set as timed and there is a config value in the config file to set the waiting time until it sends a timer error message.

Sent messages: `TaskStatusM`.

Subscribed to messages: `UserInputM`.

Specific config values: `m_TimeOut`. Time to wait in seconds until a timer error is sent.

GrabbingTaskModule

Description. This module is responsible for grabbing an object with the Katana arm.

Sent messages: `TaskStatusM`, `SearchGrippableObjectsM`.

Subscribed to messages: `ObjectGrippingStatusM`, `ObjectSearchStatusM`.

Specific config values: `m_GrabItAllMode`. Determines what should be done if the desired object is not recognized. When set to true, any object, recognized or not, will be grabbed.

[8]As subclass of `TaskModule`, every Task is subscribed to the generic messages `CheckTaskStatusM`, `StopTaskM`, `StartTaskM`, `RestartTaskM`. For simplicity, these are omitted from all the descriptions.

`LearningTaskModule`

Description. This module learns automatically from a certain camera an object or a person standing right in front. Experience has shown that if there is some problem in the access to the source camera, the module could get frozen. In addition, many times learning tasks are relevant but not always essential for the performance of a sequence of tasks. Therefore this task has also been set timed by default.

Sent messages: `TaskStatusM`, `ORCommandM`.
Subscribed to messages: `ORAutoLearnStatusM`.
Specific config values:
`m_ImageCount`. Number of images used to learn a face or an object.
`m_SrcCamera`. Represents from which camera the images will be taken.
`m_FilterFaces`. Config value for the learning of people.
`m_TimeOut`. Time to wait in seconds until a timer error is sent.

`NavigationTaskModule`

Description. This task is responsible for navigation to a certain `Location`. Because of the new definition of `Location`, the `LookAt` option present before in the `StartNavigationM` is also encapsulated and only taken in account if there is a focus defined in the map (i.e. there is another point with the same name and Focus termination). This point is then taken as the default focus.

Sent messages: `TaskStatusM`, `StopNavigationM`, `StartNavigationM`.
Subscribed to messages: `TargetReachedM`, `TargetUnreachableM`.

`RecognizingTaskModule`

Description. This task encapsulated the recognition of a person or object in front of the source camera.

Sent messages: `TaskStatusM`, `ORCommandM`.
Subscribed to messages: `ORMatchResult`.

`ReleasingTaskModule`

Description. This task encapsulates the releasing of an object that is currently held by the Katana arm. One spetial feature of this task is that it has to interact at least twice with the underlying arm control modules, since it does two movements, one to put the katana in a reachable position for an operator to receive the object, and another one after some time to open the Katana gripper.

Sent messages: `TaskStatusM`, `RobotArmMoveM`.
Subscribed to messages: `RobotArmStateM`, `RobotArmMoveFinishedM`.

`TalkingTaskModule`

Description. Says a sentence or text through the speakers.
Sent messages: `TaskStatusM`, `SpeechOutM`.
Subscribed to messages: `SpeechOutStatusM`.

3.2.7 Executing the tasks: The `TaskPlannerModule`

The purpose of the module wrapping is adding a layer of abstraction over the message interface offered by every control module. Once all the tasks have been wrapped, all of them send the same generic messages and are also subscribed to the same generic control messages. This way, whereas before every module needed to be handcrafted in order to handle all the specific messages offered by the underlying needed modules (e.g. katana control modules, navigation), now a generic task planner can handle all the wrappers and perform arbitrary combinations of them in sequence.

State machine

The `TaskPlannerModule` is also a state machine. It is a spetial module in the sense that it does not offer the same interface as the rest of the wrapped modules. Therefore, it is separated although it has as set of states a subset from the states the other tasks have. If one takes an abstract perspective to look at the job performed by the `TaskPlannerModule` this is a reasonable idea, since it also performs a task. The reason why it does not have *exactly* the same set of states comes from the idea that this is a *control module*. Currently, there is no other module or thread that monitors this module. Hence the `ERROR` state is not used. The `FINISHED` state is not needed either, since it will go directly to `READY` after the `CLEANING_UP` phase. However, this state could be added in the future if higher layers of abstraction are built over the task planner in order to increase the complexity of its behavior.

As it is shown in figure 3.8, the state diagram is quite simple. The main cause for this is that the homogeneous interfaces of the wrappers simplify the complexity of the state machine, since there are no message conflicts. Indeed, the main execution process is rather simple: the `TaskPlannerModule` waits for a `TaskInfoM` containing all the needed information to perform a sequence of tasks. One received, performs them one after another, monitoring them via `TaskStatusM` messages. Once finished with the last task, performs the cleaning up and waits for further messages. Nevertheless, the diagram also shows

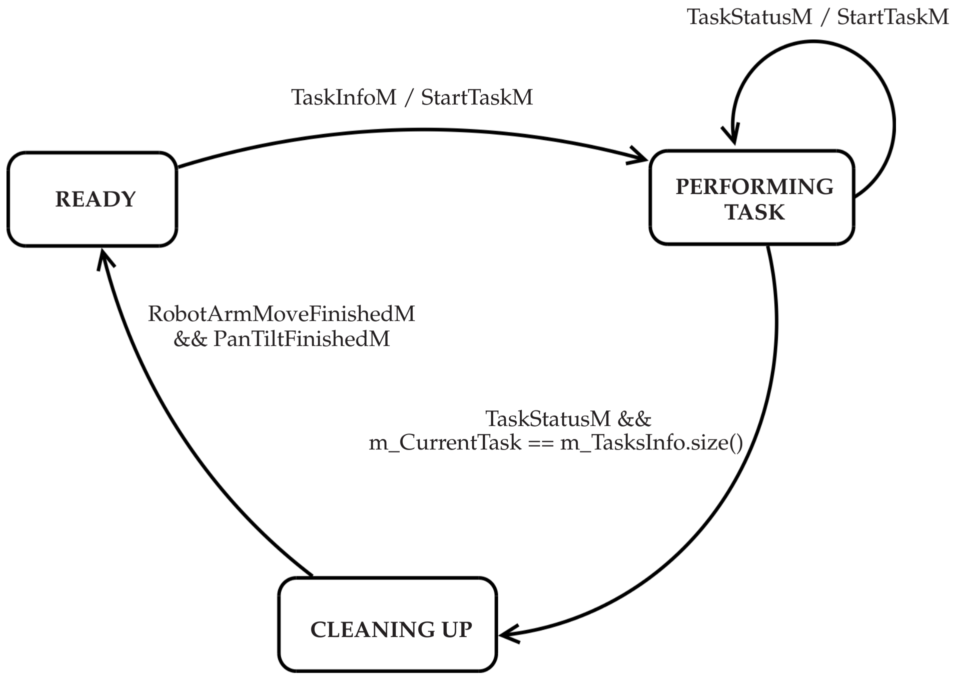

TaskStatusM / StartTaskM
TaskInfoM / StartTaskM
READY
PERFORMING TASK
RobotArmMoveFinishedM && PanTiltFinishedM
TaskStatusM && m_CurrentTask == m_TasksInfo.size()
CLEANING UP

Figure 3.8: State diagram for the `TaskPlannerModule`.

that there are some specific messages still in it. The reason for this is that there is not a `CleaningUpModule` that wraps this operation, abstracting these messages. Indeed, it could be a good feature to develop but it was not considered in time to include it in the main set of tasks.

The `CLEANING_UP` process in this case is different to the one made in the wrappers, which was mainly aimed to reset inner values of each module. In this case it not only does that, but also sends messages to the hardware in order to make every device go back to the initial state.

Regarding the `TaskInfoM`, it is send by the `SimpleCommandExtractor` module, whose job is extracting the necessary information from an input string and transforming it into a more usable structure to be handled later by the `TaskPlannerModule`[9]. This message contains a sequence of `TaskInfo` objects, which include each the `TaskType`, `Location`, `PhysicalObject` and text when needed. This information is verified by the `TaskPlannerModule`, completed when necessary and possible and used for the starting of the current task.

[9]The `SimpleCommandExtractor` is discussed in detail in subsection 3.4

Verifying the information received

Before starting any Task, the `TaskPlannerModule` verifies the parameters received in order to detect errors or missing information that could in some cases be completed.

This checking is made regarding the type of task required, field that, needless to say, must always have a value. The following comprobations are made for each kind of task:

- **Navigation**. First of all it is checked that there is a `Location` assigned to the task (i.e. the location name is in the map). Then it verifies that Lisa is not already there. As a remark, the change from points to locations allows Lisa to check whether she is inside a destination room already before trying to drive. This is done by the algorithm *Point-in-Polygon* described before in algorithm 1. In this case the verification throws a false result, which is interpreted right now as *skipping task*.
- **Talking**. The verifying method checks whether there is a given text. If not, this task is skipped.
- **Grabbing**. Checks if there is an object name. If not Lisa will still try to find an object in front of her. When it is not an object name it checks if it is a category via the `DomainReasoner`, which will be discussed later. A list of candidate objects is obtained and the information of the desired `PhysicalObjects` is updated.
- **Releasing**. Checks if it is holding anything from before. If not, will change the task into navigation and only drive to the target.
- **Bringing**. Same as releasing, checks if she is holding anyhthing and changes the task into navigation when necessary.
- **Learning**. Checks whether there is a name for the learned person. When not, assigns a default name and continues with the task.
- **Recognizing**. No verification added, anyone in front of the robot will be recognized.
- **Following**. No verification added, anyone next to the robot will be the followed operator.
- **Getting answer**. Checks whether there is an expected sentence. If not, it informs of it and skips the task.

Monitoring the execution of tasks

Following with the same principle that says that performing tasks can get stuck while waiting for a message, the `TaskPlannerModule` keeps track of the execution state of the currently performing task by periodically sending a `CheckTaskStatusM` to it. The frequency with which these messages are sent is defined by the user in the xml file, defining the timeout in seconds that will trigger each message. In case of error, it calls the `errorHandling` method in order to make a decision.

Besides the wrapped tasks described previously, two artificial types of task have been added in order to allow the user to interrupt the current sequence of tasks at any moment. These are the *interruption types*:

- **`STOP`**. It stops the currently performing task and aborts the execution of the whole sequence.
- **`CONTINUE`**. It stops the currently performing task and moves on to the next. This comes from the necessity to assign an arbitrary (i.e. decided by an operator) end to otherwise endless tasks, as following. In the current implementation of the software, a speech file has been added so the user can say *"Lisa continue"* in order to trigger the next task.

Error handling

The error handling is not one of the main goals of this work. Nonetheless, it has been considered in the task planning and some basic features have been implemented. Among the errors that are currently handled are:

- **Timeouts**. Each timed task that runs out of time is sending a Timer error. When this happens the `errorHandler` method sends a `RestartTaskM` as many times as allowed by the config value `m_MaxNumRetries` configurable in the xml file. After this number of retries the handler moves on to the next task.
- **Navigation errors**. When there is an error trying to reach destiny the task is also retried as many times as possible.
- **Grabbing errors**. When the problem is the recognition of the object because it is the wrong one or it was not recognized properly, it tries with the next candidate in the list, when there is more than one desired object in the list. This happens when the planner has been asked for a category and not for a specific object, then it fills a list with possible candidates and tries with all of them. If there are not more candidates, tries to restart as

many times as possible. If no objects were found it also tries to restart, because it will make Lisa drive around trying to get a better position. Another situation that handled is the error in grabbing that leads to errors in subsequent tasks, such as bringing or releasing. In both cases the navigation is still done but the arm releasing movement is omitted in the latter case[10]. Moreover, in the case where she is asked to take an object when she is already holding one, this second task is skipped.

3.2.8 Increasing the set of tasks

One of the main goals of this work is to enable the project to grow easily, adding one higher abstraction layer that save developers the time to deal with specific interfaces. Indeed, not only for using the already developed tasks, but also facilitate the increase of the number of tasks.

To this end, a template for wrapping a general task has been created, giving the developer the necessary information to include it in the task planning. A user guide to the framework, describing its structure in a developer-oriented fashion is described in detail in Appendix A.

3.3 Semantics in maps and data representation

This section describes the current features available in Lisa's mapping, discusses desirable information that should be added, how it has been structured and added and the key points in the process of development and implementation.

3.3.1 Current map representation

Lisa's current representation of her environment is only covered in a metric approach (gridmap or occupancy map) that allows her to know which parts of her surroundings are free and which are obstacles that she needs to avoid. In addition, there is some information that is added as a layer on top of this occupancy map, and covers obstacles in the form of lists of points that conform polygons. This is mainly used for manually adding information about obstacles that cannot be identified by Lisa's sensors but are known *a priori*.

At this moment, Lisa is only able to gather empirical measurements of her environment and build an occupancy map. This means that she is not really able to understand it but to measure it. Therefore a closed door is the same

[10]Note that this behavior is arbitrarily chosen as the best possibility. However, the reaction to this type of situations can be changed when required in the `error handler`

as a wall for her as long as they both are obstacles through which she cannot drive. Furthermore, she is able to grip objects does not *understand* whether a bottle can contain a drink or a fridge can contain a bottle. She is not aware that a table could have objects standing on it. She does not have any information about the fact that people are dynamically moving and usually stand or seat, or about the fact that walls cannot be moved but furniture can and usually is[11]. By any means she is not able to communicate with a user in a human-like fashion, since she does not store any semantic nor topological information about her environment.

Regarding to the three paradigms considered in section 2.2.2, Lisa's current implementation only addresses the metric approach. Therefore, in order to make Lisa's understanding of her environment more complete and useful, this master thesis proposes to add to her representation of the world some semantic information.

3.3.2 Structuring semantic knowledge

In order to be able to add and work with semantic knowledge, the first step to take is to decide which kind of knowledge will be used and how.

Considering the fact that the approach proposed in this work does not include logical reasoning about properties, its main contribution will be a hierarchical classification of places and artifacts that can be used by the task planner to make decisions. Hierarchical knowledge about the environment is a powerful tool for human-robot interaction and decision making. These categories will include properties in order to make it possible in the future to add reasoning without major changes in the structure of the software.

Once the decision of using a hierarchical classification is made, the next question that arises is which categories should be included. Although the proposed hierarchy is heavily based in the one proposed by Galindo *et al.* [GFMGS08], some changes are suggested.

Regarding the Space hierarchy that is shown in figure 3.9, some categories have been added. Historically *Points* have been the main space entity for navigation, and therefore it has been considered an important category to include. Furthermore, *Corridors* have been considered rooms whereas in the other hierarchy they are considered a different type of area. The reason for this is that mostly in @Home games and in any activity that has been performed so far, corridors

[11]Not all these types of information are covered by the approach described in this work, although most of them could be adressed using as a base the proposed structure.

have the same features as rooms and are treated the same way. In addition, the category *connector* has been added and includes the class[12] *Doorway*.

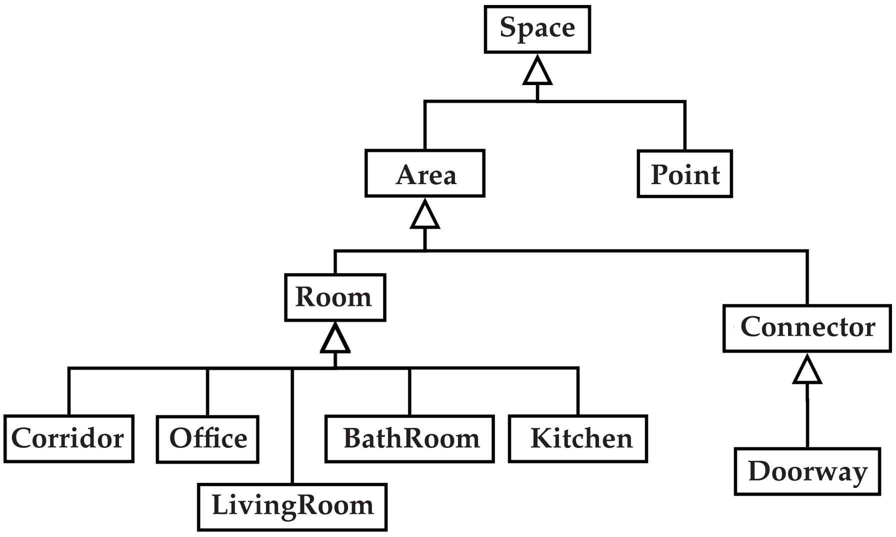


Figure 3.9: Proposed space knowledge hierarchy

Changes have also been proposed for the Artifact hierarchy in figure 3.10. The main and most important difference is the inclusion of *InteractiveArtifact* and *NonInteractiveArtifact* categories. An *InteractiveArtifact* is any object present in the @Home arena which Lisa is expected to interact with in a way that can change the object's state, being this its position, shape or any other feature like, for instance, whether it is open or closed, empty or full (for the case of *Containers*). This is not a hundred percent real, since Lisa is not currently able to open containers, but it would be logical to think that at some moment in the future she could be able to do so and this should be modelled in the general knowledge she has about her environment. Besides *Container*, the other main category in *InteractiveArtifact* is *GrippableObject*. This includes anything that can be grabbed by Lisa, either with her Katana arm or with her Gripper. On the other hand there is the *NonInteractiveArtifact*. This includes objects that can be found in the @Home arena that cannot change their state because of Lisa. Among these *SurfaceArtifacts* can be found. These are objects (usually furniture) that have any flat surface that can hold grippable objects on them. This distinction is made with the idea that there can be reasoning about it for searching objects. Lastly, there is the category *OtherFurniture*, which include other type of objects that are

[12]In this section, words *class* and *category* have the same meaning and are therefore exchangeable.

usually in the environment but with which Lisa cannot interact, either directly or indirectly.

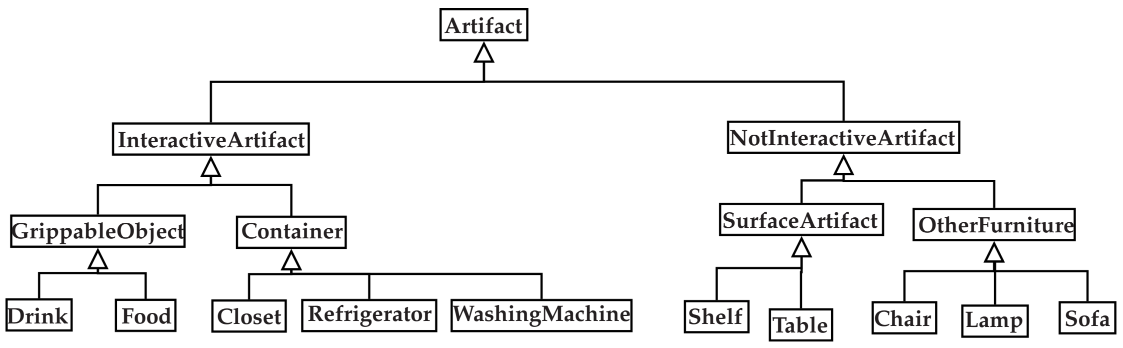


Figure 3.10: Proposed artifact knowledge hierarchy

3.3.3 Semantic knowledge representation

In order to be able to manipulate the previously defined knowledge, it is necessary to choose a representation.

What is clear is that this knowledge needs to be stored with some structure in a file that can after be loaded by the module that uses it. This enables the developer to easily change the structure when needed, which is a desirable feature considering this knowledge is stated from common assumptions about the environment and these assumptions can change with time because of culture, new interactions and tasks that are developed or, why not, new artifacts that become usual in home environments in the near future and need to be taken in account.

Since the type of information that is being handled is semantic, the first idea that is considered to represent it is OWL[13] in any of variants (OWL Lite, DL or Full). This would endow the developer with a broad semantic expressiveness. There are two main disadvantages in the use of OWL for the purpose of this work. The first is that although it is a powerful tool for expressing semantics, it can be difficult to read and understand and maybe for the purpose of classifying categories and adding some simple properties it is not necessary. The second is that the parsing of OWL files is mainly oriented for Java developers.

Considering the kind of information needed to be represented, XML seemed to be a good option. An advantage of this approach is that the project includes a XML parser. Indeed, there is a `SceneGraph` class that has a tree structure and

[13]Ontology Web Language - http://www.w3.org/TR/owl-features/

that can be initialized from a XML file with the proper structure. This simplifies the loading of the knowledge structure.

However, the consideration that maybe in the future a bigger capacity of semantic representation is needed and maybe some reasoning about properties is added should be also taken into account. For this purpose the knowledge about the environment is encapsulated by the class `DomainReasoner`, that includes the hierarchical classification and can answer some queries that can be extended along time.

Queries considered for the first stage of task planning are mainly related to ascendant/descendant relationship. It is possible to know, for a given category, who is its parent category, who are its children (direct or its whole set of descendants) and whether it has parent or children. Given two categories, whether they are related in these terms. This type of knowledge is used by the task planner to complete information and perform tasks when categories instead of objects are present in the tasks received.

`Categories` and `Properties` have been also added in order to represent the type of information that should be handled by a `DomainReasoner`. However, they are not fully used since the logical reasoning about them has been left out of the scope of this work. Nevertheless, further development in this class could use them for the addition of this logical reasoning.

The relationship between these classes along with their attributes and methods can be seen in figure 3.11, where an implementation class diagram for this part of the framework is shown. Here, every object belongs to a `Category` and is represented in a `MapLayer`[14]

3.3.4 New map layers for representing semantic annotations

General Knowledge about the environment is something that does not change during running time. However, this is not the case for the information contained on the map. This can of course be loaded from a file but it is very likely that is modified during execution time.

For this reason there is the need for the creation of a new type of message that includes all the semantic information included in the map. This way, when a user interacts with the map adding or removing information, the updated semantic layer will be sent as a message and the Task Planner will be aware of it.

[14]These map layers are contained in the `SemanticMapLayerM` and do not appear explicitly named as `MapLayer`.

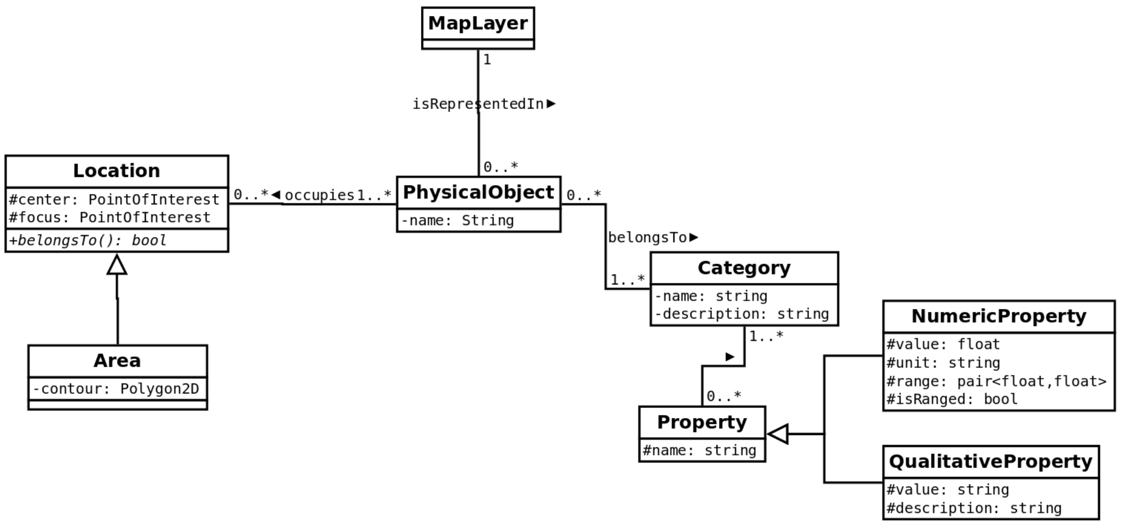


Figure 3.11: Implementation class diagram for the semantic information.

This is the case of the `SemanticMapLayerM`, which has been added to include information about the three considered layers, which are rooms, interactive artifacts and non-interactive artifacts. Each time a change is made in the GUI, a `SemanticMapLayerM` message is sent by it and received by the `TaskPlannerModule`, which updates its representation of the map.

3.3.5 Adding semantic information: GUI

In order to be able to add this new information to the map it is necessary to change the map GUI already existing and enable it to work with the new layers.

Since Lisa's GUI is already implemented as well and intending to keep the code clean and the result of this master thesis as cleanly separated as possible, the choice made was to add a inner tab inside the `MapTab` that contains the new interaction needed.

The result is shown in figure 3.12. Some limited code needed to be added to the `MapTab` class and also some new information needed to be added to the `MapDisplay` class in order to display the rooms as well.

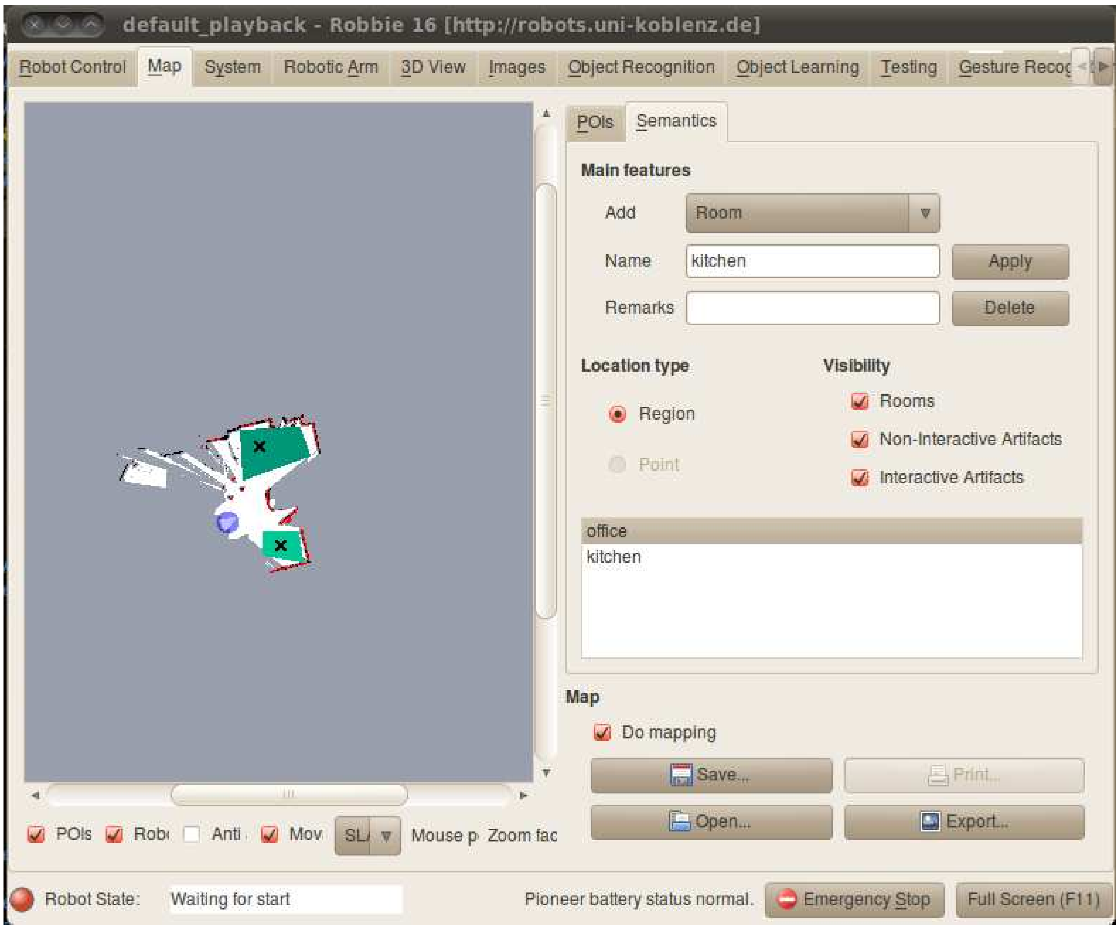


Figure 3.12: Screenshot of the MapTab with the new Semantics Tab and the representation of the information in the MapDisplay.

3.4 String command extraction

This section discusses what kind of input should be accepted for command extraction, describes it and explains how it is processed to generate the necessary information for the Task Planner to perform a sequence of tasks.

3.4.1 Current input processing

Every user input is currently processed in every module that receives a `UserInputM`. Therefore, whenever a module is added to the project, there are many steps that need to be done, including:

- Implementing a new module with a new set of states.

- Hard coding the expected user input within this new module.
- Adding a new Speech File and the subsequent config values to the xml files.

There are several difficulties with this solution. Most of them are related to the fact that speech and language are mixed together and scattered all over the project and jeopardize some of the design goals of Robbie and Lisa project, such as maintainability and extensibility [TSL$^+$11].

One of them is that it is possible that **different inputs are expected for the same action**. For instance, it could happen that in Game A a user could say *"go to the kitchen"* and be understood, and when running Game B, the user needed to say *"move to the kitchen"* to obtain the same result. In addition, this problem could lead to its counterpart: **same input could result in a different behavior** in different modules. Accordingly, a user in could say *"find the water"* and that could lead to Lisa navigating to where the water is and grabbing it in Game A and in Game B it could also include her bringing the water back to where the user is. This is not only confusing for users but also difficult to maintain.

The second problem that comes from this is that there is language processing in most of the modules and this leads to **repeated code**, which requires again high maintenance effort. Considering the possibility of language processing or speech recognition being improved in the future, it would be really difficult to undertake a project like this when there are a considerable amount of modules and the developer must work through all of them changing the required parts of the code that are related to his work.

3.4.2 Definition of a desirable valid input

An idea that comes through the problems stated in the previous subsection is that the accepted, valid input is not defined anywhere. There is some documentation about speech commands in Lisa's handbook but again only for games and not for each module. As a result, it is sometimes necessary for users and developers to read the code of a module and search the parts where the input is processed in order to know which is the exactly sentence or word that will trigger a certain action.

To this end, this work intends to define a desirable accepted valid input that could theoretically be used in every module. This way there would be an only documented definition that any user could read and it would be in one only module, so the extension, maintenance or improvement of it would be limited to the input parsing module.

In order to separate speech recognition and language processing, this input is expected to be already processed from speech. The format that has been chosen for it is a vector of words, to keep the compatibility with the `UserInputM` and also for its simplicity.

The following EBNF describes the first proposal for accepted input in the system:

Listing 3.1: Desired valid input EBNF

```
<input>            ::= <sentence> [{[","] <sentence>} <last_sentence>]

<last_sentence>    ::= "and" <sentence>

<sentence>         ::= <navigation_sent>
                     | <grabbing_sent>
                     | <delivering_sent>
                     | <talking_sent>
                     | <introducing_sent>
                     | <learning_sent>
                     | <recognizing_sent>
                     | <following_sent>

<navigation_sent>  ::= <navigation_verb> <direction_prep> ["the"] <location>

<grabbing_sent>    ::= <grabbing_verb> ["the"] <object>
                        [<placement_prep> ["the"] <location>]

<delivering_sent>  ::= <delivering_verb> <object>
                        <placement_prep> ["the"] <location>
                     | <delivering_verb> <object>
                        <direction_prep> ["the"] <location>
                     | <delivering_verb> <object>
                        <direction_prep> <person> [<placement_prep> <location>]

<talking_sent>     ::= <talking_verb> <string>
                     | <talking_verb> <direction_prep> <person>
                     | <talking_verb> <string> <direction_prep> <person>

<introducing_sent> ::= "introduce␣yourself"

<learning_sent>    ::= <learning_verb> <object> [<placement_prep> <location>]
                     | <learning_verb> <person> [<placement_prep> <location>]

<recognizing_sent> ::= <recognizing_verb> <generic_object_name>
                                <placement_prep> ["the"] <location>
                     | <recognizing_verb> <generic_person_name>
                                <placement_prep> ["the"] <location>

<following_sent>   ::= <following_verb> <person> [ <placement_prep> <location> ]

<location>         ::= <learned_location_name>
                     | "there" | "here" | "in␣front␣of␣you"

<generic_object_name>   ::= "object" | "thing"
<generic_person_name>   ::= "person"
<object>                ::= <object_name_in_dictionary> | "it"
```

```
<person>                ::= <person_name_in_dictionary> | "him" | "her"
                          | "the␣person" | "person" | "someone" | "somebody"

<placement_prep>        ::= "in" | "at" | "on" | "of"
<direction_prep>        ::= "to"

<navigation_verb>       ::= "go" | "navigate" | "drive" | "move"
<grabbing_verb>         ::= "grab" | "take" | "get"
<delivering_verb>       ::= "deliver" | ... | "put"
<talking_verb>          ::= "say" | "talk"
<learning_verb>         ::= "learn" | "memorize"
<recognizing_verb>      ::= "recognize" | "identify"
<following_verb>        ::= "follow"
```

This is a very strict syntactic definition of the input that should be accepted. Although speech recognition is out of the scope of this work, it should be kept in mind that eventually there is a speech recognition module that captures the user's command and converts it to a string or vector of strings that will be processed by this simple command extractor. Therefore and considering the current state of speech recognition, it would be unrealistic to expect that the command extractor receives a complete sentence like:

***Go** to the **kitchen**, **take** the **cup** and **give** it to **Michael**.*

Indeed, it is likely that many times the string parsed would be something more similar to:

***Go kitchen**, **take cup** and **give** it the **Michael**.*

The question here is whether it is necessary for the command extractor to really process every linking word and preposition so the sentence is syntactically correct. Furthermore, another question is whether all these linking words, prepositions and connectors add any necessary information to perform the tasks. The truth is that they do not add such information, at least in the current range of Lisa's skills. As long as the words printed in bold are understood, the requested tasks would be completely defined and they could be performed. However, they are necessary for the user to state a meaningful sentence. They are meaningful in terms of Human-Robot Interaction. It would also be unrealistic and not user-friendly at all to expect the user to talk using only the words that will finally be processed to execute the tasks:

Go kitchen take cup give Michael.

Therefore, the decision made was to allow all these words as input but only process the ones that contain semantic information. This way the set of valid sentences is broader and the robustness of the command extractor is increased, making it less vulnerable to errors in speech recognition.

3.4.3 Implementation of the Simple command extractor

In order to extract the information from this vector of strings received in the `UserInputM` message, some already implemented tools were considered.

The first idea was to use Flex[15] and Bison[16]. Flex is a tool used for generating scanners and Bison is a parser generator that can convert a free-context grammar into a LR parser. They are usually used together to generate compilers, where Flex generates a sequence of tokens that are after analyzed and processed by the software generated by Bison.

This solution entailed one problem, which is the integration with the current software. Parsers generated by Bison are C code and there was no easy way to include this code in a module that is loaded when starting Robbie server. In addition, the process needed to include the new generated parser anytime there was change did not seem easy, since it would include at least steps like:

- Change the grammar specification and/or the tokens in Flex.
- Run Bison and Flex again to generate the new parser.
- Modify the necessary part to be able to inlcude it in the command extractor module.
- Update the module.
- Compile again.

Moreover, the grammar could be specified by a regular expression and therefore it did not seem necessary to use a complex free-context grammar processor when the input is a much more simpler, regular grammar.

Consequently, the second solution considered was to use the Boost regular expression library[17]. Boost is a portable set of libraries with a wide range of applications, one of them regular expressions, very similar to the ones used in Perl language. Another advantage of this decision was that Boost is already included in the project, so it would be only using a library in the programming of the module. Nevertheless, analysing the input in more detail it can be seen that there is not a considerable use of regular expressions in the parsing. Instead, there are a defined set of key words that are known and they are just searched through the sentence.

On the other hand, one of the properties of English sentence structure is that its word order is very rigid [VS00], spetially regarding to imperative sentences,

[15]http://flex.sourceforge.net
[16]http://www.gnu.org/software/bison
[17]http://www.boost.org/doc/libs/1_47_0/libs/regex/doc/html/index.html

where the verb is always first. In addition, although there might be some very polite users that would like to communicate their wishes in another way, commands given to a robot are expected to be in an imperative form. This makes the sentence analysis very simple. The parser only needs to look for known verbs that are previously known and classified by task types and analyse the following words, which will relate to the current verb, comparing them with the also known sets of locations and objects. All known words are grouped in synonym sets that are defined in a config file. The `CommandExtractorModule` maintains a dictionary that pairs every verb to its task type and also lists of known objects, people and locations.

Considering this information, the final implementation considered the input as a stream of words where each known verb is a token that marks the beginning of a new sentence. The next words are analysed in terms of whether they are objects or locations and they are associated to the current verb in a `TaskInfo` object that will be part of the final `TaskInfoM` message this module will send to the `TaskPlannerModule`.

The two type of tasks that involve text are limited in their possible input when the `CommandExtractorModule` is used, since this text cannot contain as words verbs that as used as tasks. In order to avoid this problem some marking of the sentence should be added, but this is not realistic in a speech command and was not considered relevant in the scope of this work.

Chapter 4

Evaluation

This chapter intends to measure the quality of the work done. This is done through some experiments that have as a goal obtaining a measure of the complexity and usability of the new system against the old approach.

First of all, the first section discusses which values have been chosen to measure the quality of this work. The second section describes an application of the proposed system to the **General purpose** game comparing the previous state with the new approach. The third section does a similar comparison with **RIPS**[1] game. Finally, the last section summarizes the results and discusses their quality.

4.1 Measuring quality

Whenever a work like this is done there is always the need to measure how good the results are in order to be able to reach some conclusions and learn from the process. Considering the fact that one of the aims of this work is to guarantee the extensibility and maintainability of Lisa's project, these features should be taken in account in order to choose appropriate measures.

In order to measure whether the result is really more extensible than before, the most straightforward way to confirm it is to add a new module and analyse the effort needed. This effort will be measured in terms of complexity. A state diagram is shown for each experiment in order to let the reader judge this complexity in an intuitive way. However, since impressions are vague and cannot reflect in an objective way the quality of the result, some numerical values have been suggested as well. These values that are measured to reflect the complexity of adding a module are lines of code, messages sent and messages received.

[1]Robot Inspection Poster Session.

However, it could be irrelevant how good these results are if the modules added do not work or perform the tasks in the expected way. In order to measure the performance of these modules, several tests have been made. For General Purpose, a random set of sentences has been tested and the percentage of success is the choosen value.

4.2 General Purpose

General purpose has been the main application considered while working on this master thesis. The reason for this is that both General purpose and task planning have the same aim of making robots able to perform a generic set of task in an arbitrary fashion.

According to this, evaluating the quality of this work through a comparison with the previous state of General purpose game seems appropriate.

As it is shown in figures 4.1 and 4.2, the reduction in complexity is considerable. In this case all the processing work is done in the task planner. The only step that needs to be taken by the module is send a `TaskInfoM` that includes driving to some point inside the arena, as specified in the RoboCup@Home rulebook [NDFD+11]. After that the sentence is processed and the tasks performed.

Regarding success rate, the previous state does not seem very difficult to improve, since the old module did not really worked. Currently, no points have been earned with it by our @Home team. Sometimes it was because of problems with the speech recognition, but mainly it was because the module was not working. It took a long time to develop this module and the difficulties to handle all the possible situations in just one module made it very difficult to debug. Indeed, not all the possibilities are considered in it. At this point, only navigation, grabbing, releasing and following are handled and not completely. As an example, the module is subscribed to the message `TargetUnreachableM`, sent by the navigation module when there is an error reaching the target, but it does not handle it.

However, in order to ascertain whether this resulting system is better, there is the need to perform experiments and tests to check if the results obtained are the expected. To this end, a small sentence generator was implemented. It creates a small sentence that includes known names, objects, places and verbs. The procedure was therefore to generate a set of sentences that include a number of tasks between 1 and 5[2] and note the differences between the expected

[2]Although the number of tasks is restricted to three in the general purpose game, an arbitrary number between 1 and 5 is chosen to show the increased generality of the module. Note that in addition, the task planner can handle any number of tasks, but for simplicity of the results' analysis this number has been restricted to 5.

result and the obtained one. Only when the expected result matched perfectly the obtained result the task is considered as successful[3].

It is also important to describe the conditions under which these tests were made. In the config file the valid values introduced for locations were kitchen, office, table and exit. The known objects were the juice and the pringles, which belong to categories Drink and Food respectively. And the valid person names were Carmen, Susi and Viktor. This means that any other object, location and name is not understood by the command extractor. This is done to approach as much as possible the behavior expected in general purpose games.

4.2.1 Test cases

Here are explicitly enumerated all the sentences that were used for the evaluation. These were generated by the small sentence generator and the ones that were not successful are written in boldface. Following there is a discussion about the situation in which these errors occurred and the reason found for them. As it can be seen, the result is promising although there are some errors, and some cases that have not been properly considered in the implementation of the task planner, like the case where a person is also a location. This ambiguity is not properly handled by the simple command extractor and leads to errors. These specific cases where the behavior was not exactly the expected are discussed in subsection 4.2.2. Nevertheless, the success rate is 81.8%.

1. *go to the kitchen and take the drink.*
2. ***follow the person in front of you learn Carmen say hello my name is lisa and grab the drink.***
3. *move to the exit.*
4. *talk hello my name is lisa and hear next task.*
5. *follow the person in front of you talk hello my name is lisa and recognize the person in front of you.*
6. *drive to the table grab the drink learn Susi bring it to the kitchen and talk hello my name is lisa.*
7. *follow the person in front of you get the food and drive to the office.*

[3]Note that the expected behavior can be an expected handling of an error situation, e.g. when trying to grab an object Lisa does not success, but goes on with the rest of the task in a reasonable way. This expected behavior was arbitrarily decided and is the one described in section 3.2.7.

8. *recognize the person in front of you learn Viktor follow the person in front of you hear next task and identify the person in front of you.*

9. ***drive to the office grip the food bring it to the exit follow the person in front of you and hear next task.***

10. *follow the person in front of you.*

11. *recognize the person in front of you introduce yourself and drive to the office.*

12. *talk hello my name is lisa go to the kitchen hear next task and recognize the person in front of you.*

13. *grasp the food and identify the person in front of you.*

14. ***talk hello my name is lisa get the food give it to Susi talk hello my name is lisa and hear next task.***

15. *move to the table grasp the food introduce yourself and put it in the table.*

16. *say hello my name is lisa recognize the person in front of you and hear next task.*

17. ***get the juice hear next task go to the table and identify the person in front of you.***

18. *talk hello my name is lisa and talk hello my name is lisa.*

19. *follow the person in front of you follow the person in front of you take the food and navigate to the table.*

20. *learn Susi grab the food and hear next task.*

21. *learn Alyssa and follow the person in front of you.*

22. *take the drink talk hello my name is lisa and release it in the kitchen.*

23. *identify the person in front of you.*

24. *drive to the office and hear next task.*

4.2.2 Errors discussion

The following set of sentences yielded errors during execution:

1. *follow the person in front of you learn Carmen say hello my name is lisa and grab the drink.* Here, the error was that the task planner got stucked in the ″learn Carmen″ task. Apparently did not get the name but that should not been a problem, because in that case a generic ″User″ name is used. One reason can be the `ORAutoLearningModule` or maybe a connection loss with the kinect.

2. *drive to the office grip the food bring it to the exit follow the person in front of you and hear next task.*. Here, the driving to exit was skipped after not being able to grab the food, that was not in the office. The expected behavior is that Lisa informs that she could not grip but she drives to the destination anyway.

3. *get the juice hear next task go to the table and identify the person in front of you.* In this case the hear next task got also stucked after one time out triggered and went to the error state. It is likely that there is a bug in the error handler but it was not found.

4. *talk hello my name is lisa get the food give it to Susi talk hello my name is lisa and hear next task*. Here, the problem is that when Susi is not right in front of lisa when she gets the food, she is not going around to find her. This is in part an error of previous design, because this type of situation was not considering when abstracting the task wrappers. The solution to this could be adding a "find person" task, which would include navigation and people detecting, and add it to the task planner.

4.3 RIPS

As a second example to test the complexity of adding a new module, RIPS has been chosen. RIPS is a suitable candidate for the task planner because it has a fixed course of action. It does not include conditions that need to be handled in a certain way by the task planner and can therefore send directly a `TaskInfoM` with a sequence of tasks that need to be performed. This can lead to a discussion on whether the task planner is as general as it seems or it has limitations. Indeed, the current design of the task planner has some limitations, since it is only able to perform tasks in a fixed sequence. This means that neither conditions can be modelled in a very easy way, nor alternative execution paths. This remains to the task of the developer, who has now to interact with the task planner.

However, although there are cases and games for which the task planner cannot be used directly (it can always be adapted to more specific tasks) in its most general fashion, it can be used as a base on which build further reasoning and represent conditions.

Going back to RIPS, a smaller version of RIPS in the old system has been implemented to perform this test. The reason for this is that the use of GiGo is not addressed in this approach to the task planning and it would not be trustworthy to count the lines of code of a module that performs different actions than the module that is going to be implemented. Therefore, the smaller version of RIPS only handles Lisa's own introduction and registration, being it a sequence of actions that could be expressed in a sentence, understandable by the command extractor: *drive to the table introduce yourself give the paper to the jury say i am ready for my inspection hear exit now and drive to exit*. Nevertheless, the module sends a `TaskInfoM` message instead, as there is some ambiguity with the sentences that can be included in a "say" task. The reason for this is that when the sentence includes a verb that is already a task verb, it is not possible to tell whether the user wants Lisa to actually *say* this verb or *perform* it. Some kind of text marking would be necessary, and this is not user friendly in a speech command interface.

Once implemented both old and new version of the small RIPS game, they were tested and both working. The difference in complexity was considerable and is shown in section 4.4.

4.4 Results overview

Once implemented and tested the evaluation modules, it is possible to analyse whether the quality is good enough. As tables 4.1 and 4.2 show, the difference in complexity between the new modules and the old one is considerable. It seems that the reduction in work and maintainance is big. In the case of the General Purpose module it could be said that the maintainance of it is the same as the maintainance of the task planner itself, because the only action performed by the module is driving inside the arena once the game starts and then wait until the task planner and the command extractor module do their work. As a remark, if the number of lines of code from the old General Purpose mode is compared to the task planner module, General Purpose still has slightly more code than the task planner and considerably less generality. Furthermore, whereas the old general purpose module is subscribed to 13 different types of messages, the task planner only subscribes to 5. This means that only the wrapping of the modules has helped reducing by more than a 50% the number of messages that need to be handled in the module.

However, since one of the aims of this work was general purpose, this is not surprising. It is also interesting therefore to try to apply the task planner to another module and see if it can be used for anything else. The limitations of generality have been already discussed in the previous section, but it is also promising that one module for which the task planner was not adapted can also be beneficiated by the use of the new system.

All in all, the results prove that the intermediate abstraction layer can really facilitate the further building of new tasks and modules. Although the task planner has been shown to have its limitations as well as the semantic representation of the environment that sometimes makes it difficult to handle some ambiguity, the test results have shown that some of the main goals, such as extensibility and maintainability have been met.

Game	**Old**	**New**	**% Reduction**
GP	857	90	89.50
RIPS	396	106	73.25

Table 4.1: Lines of code in the old and new system in the two evaluated modules. The reduction value is relative to the old number of lines of code.

	Subscribed		**Sent**		% **Reduction**	
Game	old	new	old	new	Subscribed	Sent
GP	13	2	10	1	84.6	90.0
RIPS	12	2	7	1	83.3	85.7

Table 4.2: Number of subscribed and sent messages per module in the old and new system, in absolute numbers.

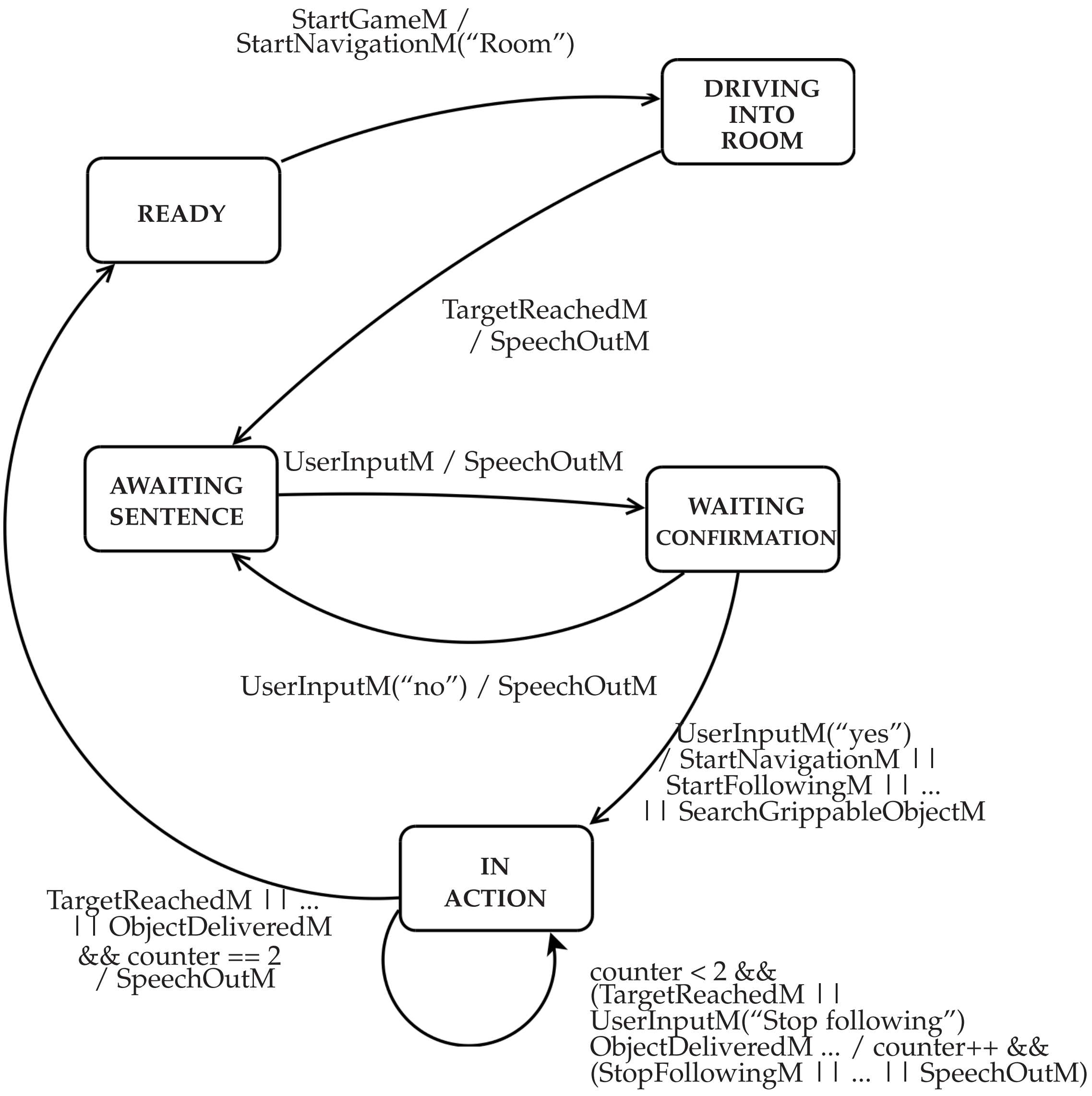

StartGameM / StartNavigationM("Room")
DRIVING INTO ROOM
READY
TargetReachedM / SpeechOutM
AWAITING SENTENCE
UserInputM / SpeechOutM
WAITING CONFIRMATION
UserInputM("no") / SpeechOutM
UserInputM("yes") / StartNavigationM || StartFollowingM || ... || SearchGrippableObjectM
IN ACTION
TargetReachedM || ... || ObjectDeliveredM && counter == 2 / SpeechOutM
counter < 2 && (TargetReachedM || UserInputM("Stop following") ObjectDeliveredM ... / counter++ && (StopFollowingM || ... || SpeechOutM)

Figure 4.1: State diagram of the old `GeneralPurposeModule`. For simplicity, not every task dependent message is shown. The counter represents the current sentence.

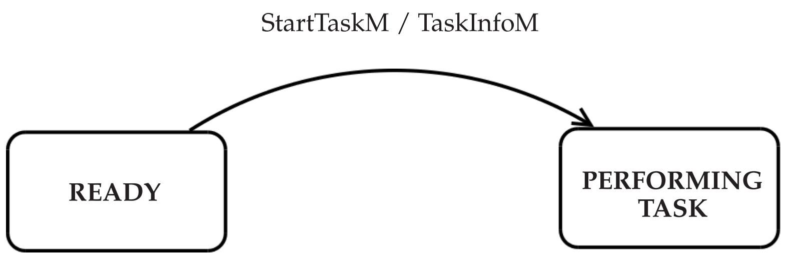


Figure 4.2: State diagram of the new `GeneralPurposeModule`. It sends a `TaskInfoM` to the task planner so Lisa drives into the room and after lets it and the `CommandExtractorModule` do the rest of the work.

Chapter 5

Further work

This master thesis is in part conceived as a basis structure or reengineering work. Accordingly, after its completion there are many features that can be added or improved opening several doors to the development and growth of Lisa's project. The structure of the software is modular enough to improve any of these parts independently with a very slight impact on the rest.

Considering the area of **Semantics**, there are many ways in which this work could be improved and broadened. The addition of properties to the categories in the general knowledge about the environment would allow Lisa to reason about her perception of the world. This includes developing logical reasoning that could be used for correction of wrong information or completion of vague information through the perception of the environment. In addition, the acquisition of any knowledge, which is now manually added, could be completed by automated or semi-automated acquisition of knowledge (i.e. scene interpretation, object recognition).

Regarding **Task Planning**, a more structured handling of errors coming from any of the tasks, making decisions when the information perceived and the general knowledge about the world do not match are possible. Furthermore, including alternative courses of action that are triggered by conditions would be an interesting improvement of the task performance, which is sequential at the moment.

In the area of **Human-Robot Interaction**, the valid input could be enhanced so users can express a broader set of goals and wishes. However, this last improvement relies heavily on the effectiveness of the speech-recognition, whose improvement would also have a positive impact in the usability of this work.

Chapter 6

Summary

During the development of this work, the purpose has always been to be consistent to the goal stated at the beginning. Overall, this means providing the necessary basis for Lisa to develop in a more general way as a human companion. More in detail, this also means generalizing the use of the tasks she is able to perform and improving the representation she has of the world along with her general knowledge about it. This needed to be done in a way that not only produces results now, but also allows further tasks and knowledge to be included in the framework.

Since this goal encompasses different fields of computer science, such as task planning, semantics, logics, mapping and natural language processing, a broad initial research phase was needed. Considering the information gathered during this phase and analysing which of the solutions already proposed by the scientific community suited best the problem adressed, some relevant choices were made. State-machine behavior instead logics was chosen as the most suitable approach to the task planner, leaving to further logic reasoning an open door in the future. Semantics were added as a new set of layers in the map and these separated parts were integrated into the system and with one another through a deep study of the current running software. To this end, some reengineering of the existing basic modules was done and also the user interface needed to be enhanced.

The diverse nature of this work made it necessary to design every part taking generality and extensibility as an essential priority. The fact that the result obtained should be an improvement in the framework, and that this improvement should be useful for further features developed by the @Home team made it also necessary that maintainability and reduction of complexity were also important priorities. There would be no use for this project if the result was too complex to be used by anyone.

Taking these priorities in account, once the software was designed and developed, the terms that should be used to verify and evaluate the result came naturally. Some measures of complexity were taken and the overall description of the complexity reduction was shown. This was done by adressing two existing @Home games and reimplementing them with the new framework, in order to analyse the improvements obtained.

The impression obtained from the evaluation is that the results are promising, although it is also true that this work has been proven to have some limitations, such as alternative courses of action and limited reasoning about the world. However, it has also been proven to be an effective solution with a high rate of success. The expectations at this moment are that the offered solution is general enough to evolve with new modules and overcome this limitations, including new features without increasing its complexity.

Taking a look into the future, some improvements seem possible after the completion of this work. These include fields as diverse as logical reasoning, increasing of semantic knowledge in an automated or semi-automated way and non-fixed or conditional planning of tasks. All in all, the perspective obtained is that the result of this work is a start point for further development of Lisa's project. Nevertheless, only the use of these results and the time will show whether the quality of the proposed solution enables the project to grow this way.

Appendix A

Framework user guide

This appendix is a user guide for current developers. Section A.1 is a brief overview of how the system works. Then, in section A.2, wrappers are described and also a small how-to guide for the adding of new wrappers is explained in section A.3. The next section includes a how-to guide for creating modules that just interact with the task planner sending to it a fixed sequence of tasks. Section A.5 explains the work done in the Domain Reasoner and section A.6 gives the user some information about the new part of the map GUI. Finally, the last section includes some FAQ and troubleshooting while in use of this framework.

A.1 How does the system work?

The main idea of the system is to add an abstraction layer to the modules so it is possible to interact with them in a general way. Before, every module had its own message interface and it was necessary to look up the necessary messages to send and handle, repeating a lot of code and reading also a lot of it.

Basically, every module that is intended to be used as a general task is covered with a wrapper module that shows a general interface that is always the same. This way, the task planner does not have to deal with every specific different message, and it can monitorize and control their execution by sending general messages that can *start*, *stop*, *restart* and *check status* [1].

As it is shown in figure A.1, if any new task is added, the only thing that is necessary for it to work with the task planner is a wrapper module. When the

[1]These messages are described in detail in the documentation of the code

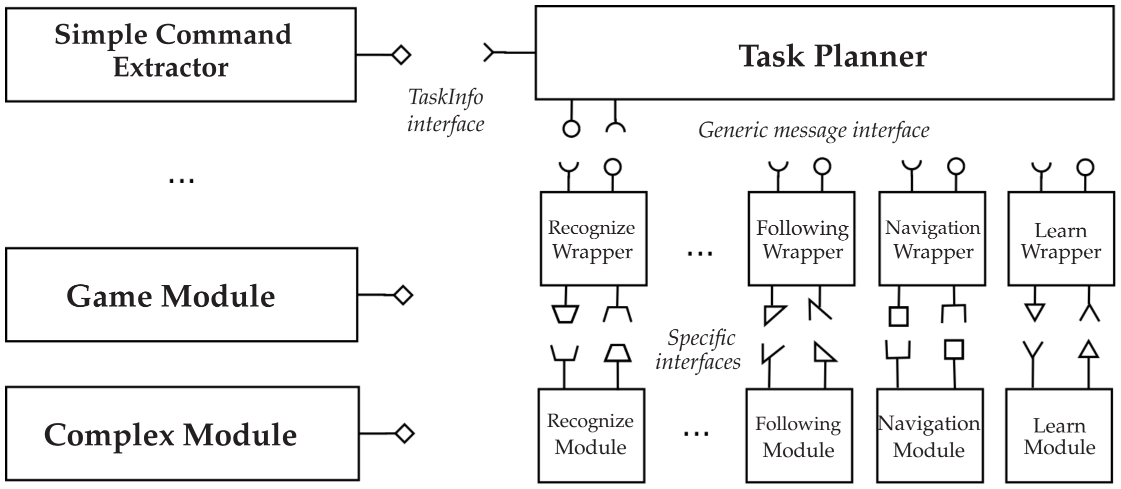


Figure A.1: Representation of the interaction between modules. Wrapper modules abstract the different message interfaces offered by the basic modules and offer the same interface to their external side. The Task Planner this way is only dealing with these messages and the TaskInfo messages sent by the rest of the modules.

task added is a fully developed new one, it could be easier just to match with this interface instead of wrapping. This option is left to the developer's choice.

A.2 How do wrappers work?

Wrappers are modules like any other in the project. They are threads that communicate through messages with other modules, and that behave like state machines according to the messages they receive. The special feature about them is that they all have the same set of states and they represent the performance of a general task.

The parent class of all the wrappers is called `TaskModule` and abstracts all the operations that are common to them. Since every wrapper is a module, the main operations represent processing of messages. On one side, the messages from the task planner[2] are processed. On the other side, the underlying modules communicate with these wrappers with *TDMs*[3].

Let's take the very simple `TalkingTaskModule` as an example. It communicates with the underlying module through a `SpeechOutM` message, which is sent when the task planner sends a `StartTaskM` to it. When the talking is finished, the wrapper also receives a `SpeechOutStatusM` indicating that it has

[2]*start*, *stop*, *restart* and *check status*
[3]*Task-Dependent Messages*

finished, and then the wrapper goes to the `READY` state and informs the task planner that everything went OK with a `TaskStatusM`.

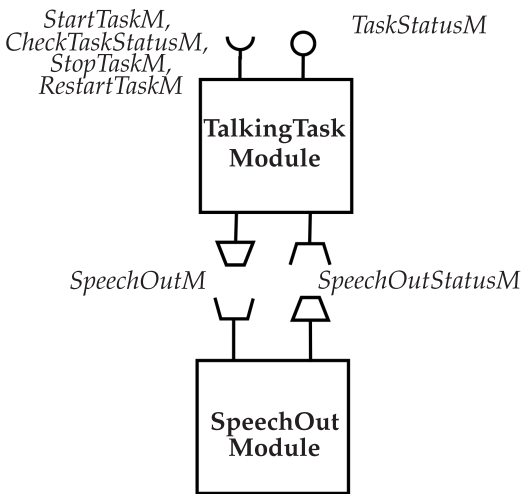


Figure A.2: Representation of the interaction between a wrapper and an underlying module. Here, the `TalkingTaskModule` abstracts the speaking interface with the messages `SpeechOutM` and `SpeechOutStatusM`.

Basically, all the processing is done in the `TaskModule` wrapper, so if you want to add a new module you do not really have to worry about the processing of any of the messages. The only thing that needs to be specified is the interaction with the underlying module, because this is the only thing that changes from one wrapper to another. Therefore, you only need to subscribe to the necessary messages, process them approppriately and inform the task planner of it with necessary. On the other hand, you also need to specify which messages will be sent to these underlying messages when a `StartTaskM` message is received. This is done in a virtual method called `startTask()` and that needs to be implemented in every new module.

The detailed processing the `TaskModule` does of these messages is depicted in detail in figure A.3.

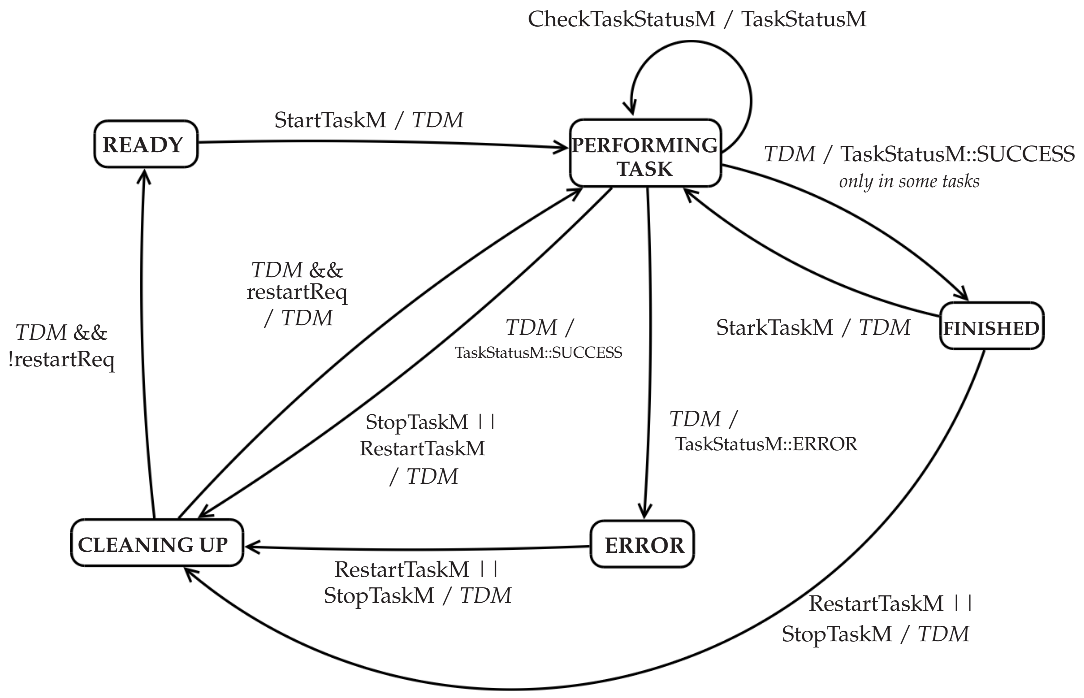

CheckTaskStatusM / TaskStatusM
READY
StartTaskM / *TDM*
PERFORMING TASK
TDM / TaskStatusM::SUCCESS *only in some tasks*
TDM && restartReq / *TDM*
TDM && !restartReq
TDM / TaskStatusM::SUCCESS
StarkTaskM / *TDM*
FINISHED
StopTaskM || RestartTaskM / *TDM*
TDM / TaskStatusM::ERROR
CLEANING UP
ERROR
RestartTaskM || StopTaskM / *TDM*
RestartTaskM || StopTaskM / *TDM*

Figure A.3: State diagram for the state machine present in each task. This set of states is shared by every task. The acronym *TDM* stands for Task-Dependent Message, which depends on the task that is wrapped.

A.3 How to create a new wrapper

Once you know about how the wrappers work and want to add one new to the possibilities of the task planning, there is a sequence of steps that you need to take, including the usual steps for adding a new module to the project:

1. Add new TaskType to: `Modules/TaskPlanning/TaskTypes.h`.
2. Copy the files `MyTaskWrapperModule.h`, `MyTaskWrapperModule.cpp` into `TaskPlanning/TaskWrappers` with the new name.
3. Replace all `MyTaskWrapper` in both with the new name.
4. Add files to `CMakeLists.txt` in `TaskPlanning/TaskWrappers`.
5. Add module to `ModuleObjects.cpp`.
6. Subscribe to underlying messages in the constructor, and also assign the new TaskType value to the attribute `m_TaskType`[4]

[4]This works like an ID for the wrapper, who can know with it if the messages received are for it or for other modules. This way this can be done in the `TaskModule` class.

7. Process the underlying subscribed methods properly.

8. Implement `startTask()` method, which should just send the necessary messages to the underlying module (usually it does not take more than one or two messages).

9. When there are some specific things you want to do when cleaning up, you can add some `specificCleaningUp().`

10. Add the module to the `taskPlanning` profile in thefile `AtHome.xml`. Otherwise it will not be loaded.

If you want also that this task you added has a verb assigned and is parsed by the `CommandExtractorModule`, which is the usual expected behavior, you will also need to do the next few steps:

11. Add the synonym verbs to the `<Synonyms>` in the taskPlanning profile.

12. Add the line `addVerbType(...)` in the method `initVerbTypes` in the `CommandExtractorModule`. This is done so it can parse them and send the appropriate `TaskInfoM`.

When the new task you added needs some specific error handling or parameter verifying, you might need to check this in the task planner module. This is done in the methods `errorHandler` and `verifyParameters`[5].

A.4 How to use tasks that are already wrapped

This is probably the easiest thing to do with this framework. The only thing that you need to do in your module is to send a `TaskInfoM` to the task planner.

In addition, you need to add to your module's profile in the `xml` file the `taskPlanning` profile as a parent. It would then look like:

```
<yourProfileName parents=”taskPlanning”>
 set your config values here
 ...
</yourProfileName>
```

A `TaskInfoM` only contains a vector of objects of the class `TaskInfo`, and this is a very simple class that contains only 4 things:

[5]**NOTE:** Only do this if you are already familiar with the behavior of the task planner. Otherwise the behavior of the task performing could change and some errors could appear while performing other type of already existing tasks

- Type of the task to perform.
- Name of the location where the task is performed.
- Name of the object that is target of the task. This can be an object or a person.
- Text necessary to perform the task (e.g. when the task involves talking or hearing).

The task planner will then perform every task sequentially, and will handle the possible errors in the way defined in the `errorHandler`.

A.5 Semantic Knowledge structure

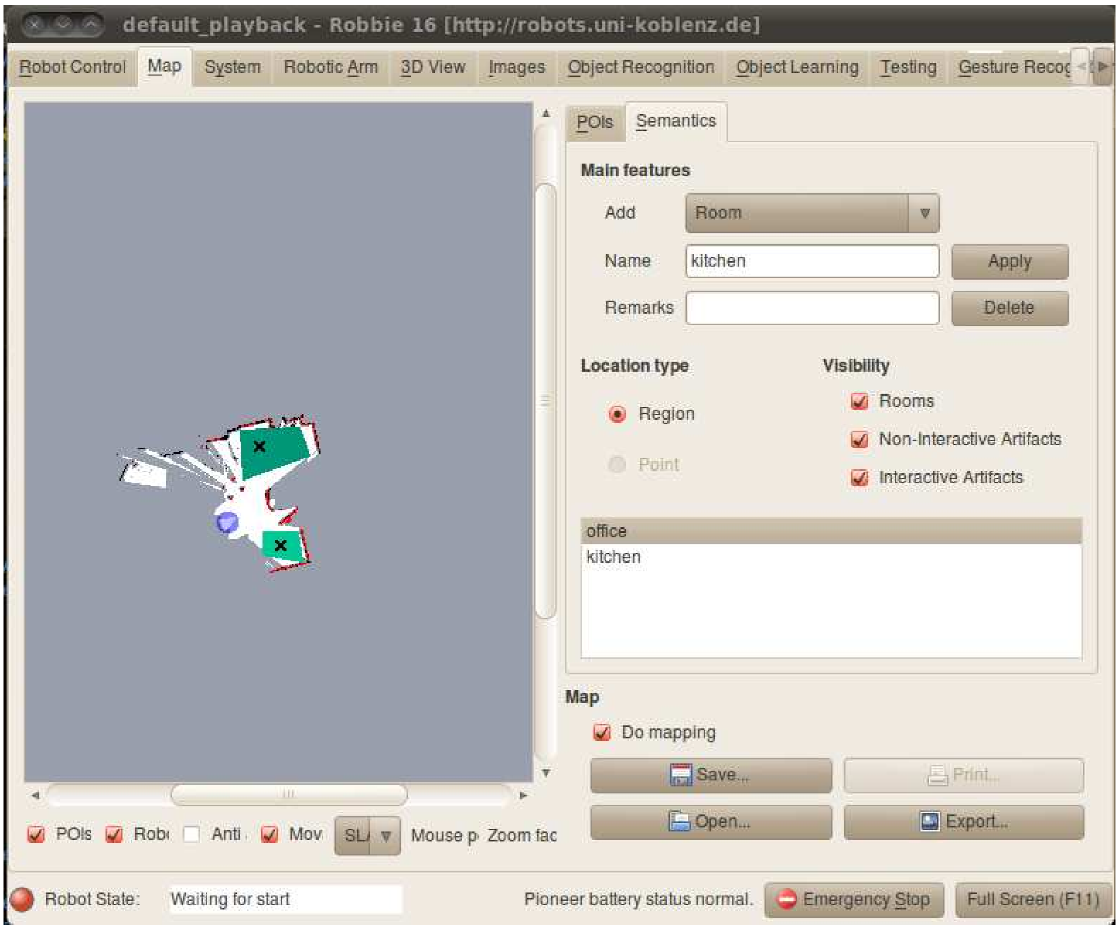
Right now, some reasoning is done about the general knowledge about the environment. The task planner has a domain reasoner[6], which basically abstracts a semantic hierarchy that is represented in the xml file `config/SemanticHierarchy.xml`. This including knowing whether a object belongs to one category, or which objects belong to it, finding parents, children, etc. If any improvement to this is made, this class is the place where it should go. There are also categories implemented, although they are not used yet. They are also described in the software documentation.

A.6 GUI general information

The Map tab has now a new sub tab that allows the user to include information like artifacts, interactive artifacts and rooms. This is all done in a very similar way to the way in the POI tab. When adding a room, the center is the first point that is clicked on and the area is defined in the usual way in the map widget. This is, by pressing shift while clicking. Then, a rectangle appears that can be modified by dragging and dropping the corners.

When clicking apply the map is updated with the new room. Note that if the center is not inside this area the GUI will not allow you to add the room.

[6]See class `DomainReasoner` in the software documentation.



Add: Room
Name: kitchen
Remarks:
Location type: [✓ Region] Point
Visibility: [✓ Rooms] [✓ Non-Interactive Artifacts] [✓ Interactive Artifacts]
office
kitchen
[✓ Do mapping]
Robot State: Waiting for start
Pioneer battery status normal.

Figure A.4: Screenshot of the MapTab with the Semantics Tab.

A.7 FAQ and Troubleshooting

Here are some answers to some FAQ and errors that can occur while using this framework. If you have any problem that does not appear here and you solve it successfully, please feel free to update this FAQ. Also, if you have any problem in general with the software or you find some bug and you do not find the solution, please feel free to contact me and I will try my best to help you[7].

I added my module and compiled it, but when I run the software I cannot make Lisa perform the new task.

This can happen for many reasons:

[7]Email:`carmennavarro@gmail.com`.

- Is the module loaded? You can check this in the System tab in the GUI. If not, it could be that you did not add the module to the Modules in the `taskPlanning` profile. Another reason could be that you did not add the module to the `ModuleObjects.cpp` file, or that it is not in the `CMakeLists.txt`
- Are you using the `CommandExtractorModule`? If so, and when you type your verb nothing happens, it could be that this verb is not added in the synonyms in the config file. Another reason could be that you forgot to add the verb type in the init in the `CommandExtractorModule`.
- Did you implement the `startTask` method in your module? When not there will not be interaction with it.
- Did you set the value for the `m_TaskType` value? When it is not set, the wrapper will not receive any message from the task planner, because first it checks if this value matches the parameter in the message.

There are some parameters specific to the new task I want to add, where do I add them?

If they are technical parameters that do not need to be changed in execution time, you can add them in the config file and load them. If not, they should be possible to reflect in terms of target locations, objects and text.

I compiled my module but when I run the software I get a ”Config value not found” error and the system breaks down.

This looks like you did not define the `taskPlanning` profile as parent of your profile. If you did not, all the paths to the config value in the task planning are different and the system does not find it. Check if your profile looks like this in the xml config:

```
<yourProfileName parents=”taskPlanning”>
 random config values handling
 ...
</yourProfileName>
```

I defined the task as timed but the timer does not seem to run.

Did you implement a `idleProcess` in your module? The timers are implemented in the `TaskModule`. To do so, the `idleProcess` is defined there, so if you did implement it, you need probably to handle your timer in your new wrapper as well.

My task enters into a repeating loop while executing, and Lisa keeps on doing the same thing again and again.

One of the reason this can happen is that in the processing of a message you are not checking in which state your wrapper is. In Lisa's framework there are many modules running at the same time and everyone receives the messages a module sends. So if your wrapper is subscribed to an underlying message to which other modules are subscribed as well, it could happen that you are handling a message that *"is not for you"*. This can be avoided by checking if you are *expecting* this message. And this is usually depending on your task's state. If it is an underlying message the one you wait for, most of the time you have to check whether you are in the state `PERFORMING_TASK`.

Bibliography

[ABD+00] Allen, J. ; Byron, D. ; Dzikovska, M. ; Ferguson, G. ; Galescu, L. ; Stent, A.: An architecture for a generic dialogue shell. In: *Natural Language Engineering* (2000)

[BBC+95] Buhmann, J. ; Burgard, W. ; Cremers, A.B. ; Fox, D. ; Hofmann, T. ; Schneider, F.E. ; Strikos, J. ; Thrun, S.: The mobile robot Rhino. In: *AI Magazine* 16 (1995), Nr. 2, S. 31

[BCF+98] Burgard, Wolfram ; Cremers, Armin B. ; Fox, Dieter ; Hähnel, Dirk ; Lakemeyer, Gerhard ; Schulz, Dirk ; Steiner, Walter ; Thrun, Sebastian: The interactive museum tour-guide robot. In: *Proceedings of the National Conference on Artificial Intelligence*, 1998

[BNK09] Becker, Thomas ; Nagel, Claus ; Kolbe, Thomas H.: A multi-layered space-event model for navigation in indoor spaces. In: *3D Geo-Information Sciences* (2009), S. 61–77

[Bro86] Brooks, Rodney A.: A robust layered control system for a mobile robot. In: *Robotics and Automation, IEEE Journal of* 2 (1986), Nr. 1, S. 14–23

[Bro90] Brooks, Rodney A.: Elephants Don't Play Chess. In: *Robotics and Autonomous Systems* (1990), S. 3–15

[Bro91a] Brooks, A.: Intelligence without representation. In: *Artificial Intelligence* 47 (1991), Nr. 1-3, S. 139–159

[Bro91b] Brooks, Rodney A.: New Approaches to Robotics. In: *Science* (1991)

[CCPR02] Chella, Antonio ; Cossentino, Massimo ; Pirrone, Roberto ; Ruisi, Andrea: Modeling Ontologies for Robotic Environments. In: *Proceedings of the 14th international conference on Software engineering and knowledge engineering* (2002), Nr. 2

[CL85] CHATILA, Raja ; LAUMOND, J.P.: Position referencing and consistent world modeling for mobile robots. In: *Robotics and Automation. Proceedings. 1985 IEEE International Conference on*, 1985, S. 138–145

[Coh99] COHN, A. G.: *Qualitative Spatial Representations*. 1999

[CR05] COLLINS, Steven H. ; RUINA, Andy: A Bipedal Walking Robot with Efficient and Human-Like Gait. In: *Proceedings of the 2005 IEEE International Conference on Robotics and Automation* (2005), S. 1983–1988

[Cro85] CROWLEY, James L.: Navigation for an intelligent mobile robot. In: *Robotics and Automation, IEEE Journal of* 1 (1985), Nr. 1, S. 31–41

[DDWB00] DISSANAYAKE, G. ; DURRANT-WHYTE, Hugh ; BAILEY, T.: A computationally efficient solution to the simultaneous localisation and map building (SLAM) problem. In: *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on* Bd. 2 IEEE, 2000, S. 1009–1014

[DGLL00] DE GIACOMO, Giuseppe ; LESPÉRANCE, Yves ; LEVESQUE, H.J.: ConGolog, a concurrent programming language based on the situation calculus. In: *Artificial Intelligence* 121 (2000), Nr. 1-2, S. 109–169

[DTK05] DIOSI, Albert ; TAYLOR, Geoffrey ; KLEEMAN, Lindsay: Interactive SLAM using Laser and Advanced Sonar. In: *Proceedings of the 2005 IEEE International Conference on Robotics and Automation* (2005)

[Elf89] ELFES, Alberto: *Occupancy grids: A probabilistic framework for robot perception and navigation*, Carnegie Mellon University, Diss., 1989

[FM09] FILIPE, Porfírio ; MAMEDE, Nuno: Indoor Domain Model for Dialogue Systems. In: *Universal Access in Human-Computer Interaction* (2009), S. 512–520

[FN71] FIKES, Richard ; NILSSON, N.: STRIPS: A new approach to the application of theorem proving to problem solving. In: *Artificial Intelligence* (1971)

[Fre90] FRENCH, R.M.: Subcognition and the Limits of the Turing Test. In: *Mind* 99 (1990), Nr. 393, S. 53–65

[GFMGS07] GALINDO, Cipriano ; FERNANDEZ-MADRIGAL, J.A. ; GONZÁLEZ, J. ; SAFFIOTTI, Alessandro: Using semantic information for improving efficiency of robot task planning. In: *ICRA Workshop: Semantic Information in Robotics*, 2007

[GFMGS08] GALINDO, Cipriano ; FERNÁNDEZ-MADRIGAL, Juan-Antonio ; GONZÁLEZ, Javier ; SAFFIOTTI, Alessandro: Robot Task Planning Using Semantic Maps. In: *Robotics and Autonomous Systems* (2008)

[Gre69] GREEN, Cordell: Application of Theorem Proving to Problem Solving. In: *Artificial Intelligence* (1969)

[Gru93] GRUBER, T.R.: Towards Principles for the Design of Ontologies Used for Knowledge Sharing. In: *Formal Ontology in Conceptual Analysis and Knowledge Representation*. Deventer, The Netherlands : Kluwer Academic Pubishers, 1993

[GSC$^+$05] GALINDO, C. ; SAFFIOTTI, A. ; CORADESCHI, S. ; BUSCHKA, P. ; FERNANDEZ-MADRIGAL, J.A. ; GONZALEZ, J.: Multi-Hierarchical Semantic Maps for Mobile Robotics. In: *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS-05)*, 2005

[Hec94] HECKBERT, P.S.: *Graphics gems IV*. AP Professional, 1994

[Hew70] HEWITT, Carl: PLANNER: A Language for Manipulating Models and Proving Theorems in a Robot. In: *CSAIL* (1970), Nr. 168

[JM00] JURAFSKY, Daniel ; MARTIN, James H.: *Speech and language processing: an introduction to natural language processing, computational linguistics, and speech recognition*. Prentice Hall, 2000 (Prentice Hall series in artificial intelligence). – ISBN 9780130950697

[Kui00] KUIPERS, Benjamin: The Spatial Semantic Hierarchy* 1. In: *Artificial Intelligence* 119 (2000), Nr. 1-2, S. 191–233

[LB03] LARMAN, C. ; BASILI, V.R.: Iterative and incremental developments. a brief history. In: *Computer* 36 (2003), Nr. 6, S. 47–56

[LF00] LEONARD, J.J. ; FEDER, H.J.S.: A computationally efficient method for large-scale concurrent mapping and localization. In: *ROBOTICS RESEARCH-INTERNATIONAL SYMPOSIUM-* Bd. 9 Citeseer, 2000, S. 169–178

[LRL$^+$94] LEVESQUE, Hector J. ; REITER, Raymond ; LESPÉRANCE, Yves ; LIN, Fangzhen ; SCHERL, Richard B.: GOLOG: A Logic Programming Language for Dynamic Domains. In: *Logic Programming* (1994)

[McD91] MCDERMOTT, Drew: A reactive plan language / Citeseer. 1991. – Forschungsbericht

[MH69] MCCARTHY, John ; HAYES, Patrick J. ; MELZER, B. (Hrsg.) ; MICHIE, D. (Hrsg.): *Some Philosophical Problems from the Standpoint of Artificial Intelligence*. Edinburgh University Press, 1969

[MJZ$^+$07] MOZOS, Óscar M. ; JENSFELT, Patric ; ZENDER, Hendrik ; KRUIJFF, Geert-Jan M. ; BURGARD, Wolfram: From Labels to Semantics: An Integrated System for Conceptual Spatial Representations of Indoor Environments for Mobile Robots. In: *Proceedings of the IEEE ICRA 2007 Workshop: Semantic information in robotics (ICRA)*, 2007

[Mor88] MORAVEC, H.P.: Sensor fusion in certainty grids for mobile robots. In: *AI Magazine* 9 (1988), Nr. 2, S. 61

[MR93] MAIO, Dario ; RIZZI, Stefano: Knowledge architecture for environment representation in autonomous agents. In: *Proc. ISCIS VIII, Istanbul* (1993)

[MS02] MCILRAITH, Sheila A. ; SON, T.C.: Adapting Golog for composition of Semantic Web Services. In: *Principles of Knowledge Representation and Reasoning - International Conference* Citeseer, 2002, S. 482–496

[NDFD$^+$11] NARDI, Daniele ; DESSIMOZ, Jean-Daniel ; FORD DOMINEY, Peter ; IOCCHI, Luca ; RYBSKI, Paul E. ; SAVAGE, Jesus ; SCHIFFER, Stefan ; SUGIURA, Komei ; WISSPEINTNER, Thomas ; ZANT, Tijn van d. ; YAZDANI, Amin ; HOLZ, Dirk ; KRAETZSCHMAR, Gerhard ; GOSSOW, David ; OLUFS, Sven: RoboCup@Home Rules and Regulations / RoboCup@Home, www.robocupathome.org. 2011. – Forschungsbericht

[Nil84] NILSSON, N.: Shakey the robot / DTIC Document. 1984. – Forschungsbericht

[Pra95] PRATT, Jerry E.: *Virtual Model Control of a Biped Walking Robot*, Massachusetts Institute of Technology, Diplomarbeit, 1995

[RK95] ROSENSCHEIN, Stanley J. ; KAEBLING, Leslie P.: A situated view of representation and control. In: *Artificial Intelligence* 73 (1995), Nr. 1-2, S. 149–173

[Ros85] ROSENSCHEIN, Stanley J.: Formal theories of knowledge in AI and robotics. In: *New Generation Computing* 3 (1985), Nr. 4, S. 345–357

[SS09] SIORPAES, Katharina ; SIMPERL, Elena: Human Intelligence in the Process of Semantic Content Creation. In: *World Wide Web* (2009)

[TBB+99] THRUN, Sebastian ; BENNEWITZ, Maren ; BURGARD, Wolfram ; CREMERS, Armin B. ; DELLAERT, Frank ; FOX, Dieter ; HÄHNEL, Dirk ; ROSENBERG, Charles ; ROY, Nicholas ; SCHULTE, Jamieson ; SCHULZ, Dirk: MINERVA: A second-generation museum tour-guide robot. In: *In Proceedings of IEEE International Conference on Robotics and Automation (ICRA99*, 1999

[TGBK98] THRUN, Sebastian ; GUTMANN, Steffen ; BURGARD, Wolfram ; KUIPERS, Benjamin J.: Integrating Topological and Metric Maps for Mobile Robot Navigation: A Statistical Approach. In: *Proceedings of the National Conference on Artificial Intelligence*, 1998, S. 989–996

[Thr98] THRUN, Sebastian: Learning Metric-Topological Maps for Indoor Mobile Robot Navigation. In: *Artificial Intelligence* 99 (1998), Nr. 1, S. 21–71

[Thr02] THRUN, S.: Robotic mapping: A survey / Carnegie Mellon University, Computer Science Department. 2002. – Forschungsbericht

[TSL+11] THIERFELDER, Susanne ; SEIB, Viktor ; LANG, Dagmar ; HÄSELICH, Marcel ; PELLENZ, Johannes ; PAULUS, Dietrich: Robbie: A Message-based Robot Architecture for Autonomous Mobile Systems. In: *INFORMATIK 2011*, 2011

[Tur50] TURING, A.M.: Computing machinery and intelligence. In: *Mind* 59 (1950), Nr. 236, S. 433–460

[VGNS07] VASUDEVAN, Shrihari ; GÄCHTER, Stefan ; NGUYEN, Viet ; SIEGWART, Roland: Cognitive maps for mobile robots - an object based approach. In: *Robotics and Autonomous Systems* (2007)

[VS00] VERSPOOR, Marjolijn ; SAUTER, Kim: *English sentence analysis: an introductory course*. J. Benjamins, 2000. – ISBN 9781556196614

[WKBH00] WERNER, S. ; KRIEG-BRÜCKNER, B. ; HERRMANN, T.: Modelling navigational knowledge by route graphs. In: *Spatial cognition II* (2000), S. 295–316

[ZJMMB07] ZENDER, Hendrik ; JENSFELT, Patric ; MARTINEZ MOZOS, Oscar ; BURGARD, Wolfram: An integrated robotic system for spatial understanding and situated interaction in indoor environments. In: *Proceedings of the National Conference on Artificial Intelligence*, 2007, S. 1584–1589

[ZMJ$^+$08] ZENDER, Hendrik ; MOZOS, Óscar M. ; JENSFELT, Patric ; KRUIJFF, Geert-Jan M. ; BURGARD, W.: Conceptual spatial representations for indoor mobile robots. In: *Robotics and Autonomous Systems* (2008)