Universidad Politécnica de Madrid

Facultad de Informática

Departamento de Inteligencia Artificial

European Masters in Computational Logic

Masters Thesis

Enhancing A Dialogue System Through Dynamic Planning

Luciana Benotti

Ana García Serrano

Madrid Spain

July 2006

a UNComa, *en estos momentos difíciles*

Contents

My Thesis

It is widely agreed that inference plays an important role in communication through natural language. In previous years, though it was believed that its role was too complex for it to be clearly delineated and, in particular, handled computationally. Nowadays, the focus is on investigating the role of inference in concrete phenomena within the general topics of Natural Language Processing and Human-Computer Interaction. In this thesis, I am going to focus on the role that certain types of inference have in a particular task within a dialogue system.

Thesis Description

Through this thesis I investigate how to add dynamic planning capabilities to a dialogue system. In particular, I will work with the dialogue system FrOz [39] (a "text adventure" computer game) developed at the University of Saarbrücken. FrOz has been developed as a state-of-the-art natural language dialogue system. The game can understand commands the player presents as simple English sentences, execute the specified actions in the game world, and report back to the player (by automatically generating natural sounding English text) on the changes introduced in the present situation. In this way, the player explores the game environment executing actions that can result in changes on the game world as well as system feedback. The testbed provided by FrOz is ideal because it is highly flexible in two ways. On one hand, it is possible to restrict or extend the interactions the player can perform with the game modifying the game scenarios. On the other hand, thanks to FrOz modular architecture I can replace a simple module by a more sophisticated one without this causing a strong impact in other modules. The result is a controlled and flexible environment where to investigate interesting dialogue phenomena. For instance, FrOz was used to investigate how to resolve and generate complex referring expressions, i.e., bridging [54].

The addition of planning capabilities to this system –the concrete topic and main goal of this thesis– would greatly increase the flexibility and effectiveness of the game when dealing with user commands. In its present version, FrOz breaks down whenever the player tries to execute an action whose preconditions do not

hold in the game world. This degrades the usability of the system because in many situations the player specifies an action assuming that either all preconditions are satisfied, or that they can be easily satisfied by performing other actions which the player presupposes as part of the action she wants to execute; I will call such actions "implicit actions".

Let us consider, for example, the situation where the game has just described that a key is lying on a table in front of the player in a room with a locked door. Then the player might input the command, "Unlock the door with the key", which the current version of FrOz will fail to execute because the player is not actually holding the key (the `unlock-with(Object Key)` action has the precondition `instance(Key inventory-object)`). That is, the player is obliged to enter a command like "Take the key and unlock the door with it" for the action to succeed. In such cases, a collaborative dialogue system would try to fill the gaps between the input received from the player and the current state of the world, in order to free her from the nuisance of specifying simple actions which need to be executed to fulfil the preconditions of more complex tasks. To determine which are these auxiliary actions, which are left unspecified by the player, a planning step is needed.

Concrete Thesis Goals: In order to solve the problem previously described, I will integrate a generic planner to do planning in the context of the game. The achievement of this goal involves:

Task 1 Identify the essential planner characteristics, choose a state-of-the-art planner and evaluate its performance.

Task 2 Design the planner interface with FrOz architecture.

Task 3 Implement the interface and integrate it into FrOz code.

Task 4 Evaluate the integration at two levels: planner performance for the given task, and performance of the capabilities added to the game.

Work-plan

Task 1: It involved the identification of the essential characteristics required for the planner, as well as the revision of one of the possible candidates for the task: PaDoK. This task began as a research project at Bolzano University [8]. After the project I determined that PaDoK was not suitable for the task, but the work clearly pointed out to an appropriate choice: Blackbox [37], the planer I am currently using in this thesis. Blackbox is an extension of SatPlan (the winner of the 2006 Planning Competition) which excels in the computation of short, optimal plans in complex domains.

Task Output: The suitable planner was chosen. Its behaviour and situation in the state of the art are described in Chapter 3.

Task 2: The design of the interface between Blackbox and FrOz involves analysing FrOz architecture as well as the input required by Blackbox. Crucially, the interface between Blackbox and FrOz is dynamic, as it should model the fact that the player's knowledge about the game world changes while the player explores its environment. Hence, the input to the planner needs to be computed during game execution querying the player's knowledge base. The more the player knows about the world, the more complex the plans that Blackbox should be able to produce. It turns out, then, that the kind of planning needed in this application is embedded in a complex framework because the initial state, goal and the available objects need to be recalculated each time an action fails during game execution. This involves interfacing with the inference system RACER [29], a highly optimised Description Logic prover, which is used in FrOz to dynamically model the changing knowledge of the player.

Hence, this task required to study Description Logics together with the inference services provided by RACER, and to determine which services are appropriate for obtaining the specification required by Blackbox. The final outcome of the task then is the definition of a generic architecture that integrates dynamic planning in order to approach a concrete phenomena of dialogue systems. The architecture profits from the strengths of RACER and Blackbox, two generic inference tools available today.

Task Output: The state of the art of Description Logics and a detailed explanation of FrOz architecture are discussed in the Chapter 2 of this thesis. The design of the integration is described in Chapter 4 and was published in [9].

Task 3: The implementation of the interface requires to set in place the interaction with Blackbox as an external tool. This involves, on one hand, the dynamic generation of appropriate planning problems, which needs to be computed on the fly representing the player's state. On the other hand, the plans returned by Blackbox need to be parsed and transformed into the representation understood by FrOz. And crucially, as I discussed above, this interface needs to be synchronised with the information handled by RACER. In concrete, a layer that coordinates RACER and Blackbox was implemented. Moreover, in order to complete this task, FrOz code (more than 10000 lines of Oz code) was debugged and analysed in depth to understand where the interface should be integrated.

Task Output: The implemented interface integrated into FrOz architecture is the output of this task. This new version of FrOz is called *FrOz Advanced (FrOzA)*, which is the functional output of this thesis.

Task 4: The evaluation of Blackbox performance as a component of our dialogue system involves the actual execution of the planner on the kinds of planning tasks

that FrOzA needs to handle. The evaluation of FrOzA involves the anlysis of its added capabilities inspecting the dynamically generated planning problems, and the study of the computed plans (for predefined test-cases).

Task Output: FrOzA evaluation results are summarised in Chapter 5.

Relevance of the Topic

The most general goal of this thesis is to advance research on the integration of state of the art inference tools to enhance the abilities of modular and robust dialogue systems. From the beginning, FrOz was developed under this premise (it integrates a general purpose Dependency Grammar parser for English, the RACER reasoner for inference, an optimised generator for realisation, etc) in order to exhibit domain adaptability and generality. This work is a further step in the same direction. It shows how to integrate planning capabilities in a dialogue system profiting from an off-the-shelf general purpose planner. In short, my challenge is to show how this planner can be used to tackle an specific dialogue phenomena (see related work in [57]).

Links with Dr. Ana García Serrano's work: Dr. Ana García Serrano's work on human language technology follows a language engineering approach: maintainability, time optimisation, robustness, flexibility, domain adaptability and generality are key features for system development. In order to achieve this goals, her approach is to profit from existing linguistic resources [A5], which match exactly this thesis premise. On the other hand, her work on dialogue systems [A1,A2,A3,A4] makes special emphasis on the importance of cooperative interaction with the user (allowing flexibility in the input and friendliness in the output). Clearly, our specific task also pursuit this goal.

Relevant Bibliography

[A1] J. Hernández, A. García Serrano, and J. Calle Gómez. Dialoguing with an online assistant in a financial domain: The VIP-advisor approach. In Proceedings of AIAI, pages 305-314. Kluwer, 2004.

[A2] A. García Serrano, J. Martínez Fernandez, and P. Martínez. Experiences in evaluating multilingual information retrieval processes. International Journal of Intelligent Systems, 21(7):655-677, 2006.

[A3] A. García Serrano and J. Calle Gómez. A cognitive architecture for the design of an interaction. Lectures Notes in Computer Science, 2446:82-89, 2002.

[A4] A. García Serrano, P. Martínez, and J. Hernández. Using AI techniques to support advanced interaction capabilities in a virtual assistant for e-commerce. Expert Systems with Applications, 26(3):413-426, 2004.

[A5] A. García Serrano, P. Martínez, and L. Rodrigo. Adapting and extending lexical resources in a dialogue system. In Proceedings of the workshop on Human Language Technology and Know ledge Management, pages 1-7. Association for Computational Linguistics, 2001

Other Links: This thesis has motivated several interactions with researchers whose work is related (in different ways) with this work:

- The initial motivation for this work aroused after the development of FrOz in the University of Saarbrücken. In [39] the authors already mention that the integration of a planner into the game would be an interesting extension. Kristina Striegnitz, one of the authors introduced me to FrOz in the initial stages of my work.

- I also got in touch with Kristina Striegnitz PhD director, Claire Gardent. A fundamental research interest of her is the integration of reasoning capabilities and inference in Natural Language Processing tasks. She suggested bibliography about 'collaborative dialogue systems' and she read a version of this thesis.

- This work evolved under the interdisciplinarity direction of Carlos Areces, whose main research interest is Logic, and Raffaella Bernardi, whose main research interest is Computational Linguistics.

- At the beginning I worked with the planner called PaDoK developed in the University of Rome by Marta Cialdea Mayer. I got in touch with her several times to report problems with PaDoK and to ask for advice. During this interaction, we discussed some planning characteristics that were very necessary for my work and the last version of PaDoK includes them.

- When I changed to the planner Blackbox I got in touch with one of its authors, Henry Kautz to report problems with Blackbox.

- This thesis includes several enhancements suggested by the ESSLLI Student Session 2006 reviewers.

- Recently I got in touch with Matthew Stone, at the University of Rutgers, and he suggested some interesting bibliography that is related with this thesis.

In addition to my director, I thank all this people for their help and interest in my work.

Chapter 1

Dialogue Systems

The belief that humans will be able to interact with computers in conversational speech has long been a favourite subject in science fiction. This reflects the persistent belief that spoken dialogue would be the most natural and powerful user interface to computers. With recent improvements in computer technology, and in speech and language processing, such systems are starting to appear feasible. However, there are significant problems that still need to be solved before this can be achieved.

We will begin this chapter explaining what we mean with *Dialogue System (DS)* and motivating their development. Moreover, we present a short historical background for such systems. Furthermore, in Section 1.3 we will analyse the abilities that are essential for a dialogue system and we will present two approaches to achieve them. TRIPS [22] and ADVICE [50] are full fledged dialogue systems that implement these approaches and we will use them to put FrOz (the DS we are interested in) in perspective.

We will close this chapter with a discussion of how planning has been used to handle different aspects of dialogue systems, the major topic of this thesis.

1.1 What is a Dialogue System?

In Computational Linguistic, the word *dialogue* is used in different ways by different people. Many researchers in the speech recognition community see dialogue methods as a way of controlling and restricting the interaction with the user. For them, systems that implement such methods will engage the user in a sort of dialogue by asking her questions. By controlling the interaction through these dialogues, speech is much more predictable leading to better recognition and language processing. This is fine, but to achieve this aim, such systems needs to severely limit dialogue interaction making it less flexible.

A different perspective on dialogue involves modelling human-computer interaction after human conversation. For the proponents of this approach, the goal is to design and build systems that approach human performance in conversational interaction, and hence language understanding becomes more complicated. In this view, dialogue allows for more complex information to be conveyed than is possible in a single utterance. Given the current technological state of the art, the second, much richer perspective, seems feasible in a foreseeable future and we believe it will lead to much more effective user interfaces to complex systems.

Some people argue that spoken language interfaces will never be as effective as graphical user interfaces (GUI). This view underestimates the potential of dialogue-based interfaces. First, there will continue to be more and more applications for which a GUI is not feasible because of the size of the device one is interacting with, or because the task one is doing requires using one's eyes or hands or moving around. In these cases, speech provides a very natural additional modality.

Even if a GUI is available, spoken dialogue can be a valuable additional modality as it adds considerable flexibility and reduces the amount of training required. Conversational interfaces would provide the opportunity for the users to state *what* they wants to do in their own terms, just as they would do with another person, while the system takes care of the complexity of *how* to carry out this task.

A very promising approach are dialogue systems that allow for a mixed-initiative interaction. Such systems try to model the human-machine interaction after human collaborative task execution. Dialogue systems may become a revolutionary paradigm for human-computer interaction if they can properly face this challenge.

1.2 A Glance into the History of Dialogue Systems

The research area of dialogue systems was born in the 60s and we can observe different eras during its evolution:

Classical Systems (1966-1972): The earlier works are characterised by a lack of theories about human-human dialogue, and this influenced directly the design of dialogue systems which were developed in an 'ad-hoc' manner. One of the earliest dialogue systems was ELIZA [61] which had a trivial pattern-matching method for dialogue production. The dialogue manager for the paranoid agent PARRY [19] was slightly more complex as it considered some global variables that registered its mood. Such variables were modified by the inputs received from the user and also affected PARRY's choice of answers. PARRY and ELIZA are considered the first *chat-bots*, predecessors of current dialogue systems. As a rule, such systems usually did not take into account more than one dialogue turn in order to generate their responses.

First Dialogue Systems (1978-1986): More sophisticated dialogue managers appeared when theories about human-human dialogue were developed. Grosz' work [28] influenced significantly the computational study of dialogue. He developed theories about sub-dialogues and task oriented dialogues. GUS [12], the dialogue system developed by Xerox PARC, was one of the first systems of this era. Its goal consisted to provide a cooperative dialogue over a restricted domain (booking of flight tickets). The approach used for its implementation was based on *frames*; in such approach the system collects information until it has enough details in order to perform certain task. OSCAR [18] also belongs to this era and is the first dialogue system to implement the *speech act theory* [6, 49], which revolutionised the area.

Towards collaborative Dialogue Systems (1988-1999): In the late 80s the DS community started to focus in the development of collaborative dialogue systems. The first examples of dialogue systems that try to provide mixed initiative interaction belong to this era. One of the precursors was the Unix Consultant [63], an agent that predicts the level of experience of the user. A traditional example is TRAINS [5], the Rochester interactive planning assistant that works on a transportation domain.

Practical Dialogue (2000-now): During the previous four decades several lessons were learnt in the Dialogue System research field. One of the most important was to realise the high complexity such systems need to handle. But also to notice that most of the potential applications for human-computer interaction do not need to capture the extent of full human conversation. In these applications, the dialogue is focused in accomplishing a concrete task. Such dialogues are called *practical dialogues* and are the kind of dialogues handled by TRIPS and ADVICE, which deal with complex enviroments (multimodal interaction, several knowledge sources, etc.).

FrOz also belongs to this last era, but works over a simplified enviroment in order to offer a testbed for current technologies. FrOz approaches dialogue phenomena profiting from generic components that belong to the state of the art of artificial intelligence and computational linguistics. This thesis is another step in the same direction.

1.3 Four Challenges for Dialogue Systems

In this section we introduce what Allen describes in [3] as the four major problems in building practical dialogue systems today.

- The first is handling the level of complexity of the language associated with the task.

- The second is integrating a dialogue system with a complex "back-end reasoning system" (e.g., a factory scheduler, a map server, an expert system for kitchen design, etc.).
- The third is the need for intention recognition as a key part of the understanding process.
- The fourth is enabling mixed-initiative interaction, in which either the system or the user may control the dialogue at different times in order to make the interaction most effective.

We will consider each of these problems in turn. After describing each of them we analyse how they are tackled by two state-of-the-art dialogue systems. One of these systems is TRIPS, a problem solving assistant developed at the University of Rochester. The other system is ADVICE, a virtual assistant for e-commerce developed in the framework of the European Commission IST Programme. Both dialogue systems implement an agent-based architecture and their goal is to assist the user in the accomplishment of a task. Finally, we will discuss to what extent the challenge has been met in FrOz.

Building Semantic Representations in Practical Dialogues: In order to deal with practical dialogue a "deep" semantic representation of what was said has to be produced. Most existing dialogue systems use hand tailored grammars. There have been many efforts over the years to develop broad coverage grammars for natural languages. The problem that arises when using such grammars in dialogue systems is that sentences have several interpretations due to the vast ambiguity inherent in natural languages. One of the mainstay techniques for dealing with this problem has been to use semantic restrictions in the grammar to enforce semantic as well as syntactic constraints. Furthermore, the general grammar can be refined further by specifying domain-specific restrictions for the current task. This can significantly reduce the possible interpretations allowed by the grammar.

TRIPS uses a feature-based augmented context free grammar with semantic restrictions to process an utterance. It looks for all possible speech acts anywhere in the utterance and then searches for the shortest sequence of acts that covers the input (or as much of the input as can be covered) as described in [2].

ADVICE uses two complementary approaches [51]. First, a deep analysis is performed using a semantic grammar based on ontological knowledge and a lexicon structured in three layers (general purpose, task oriented and domain specific). If this analysis is not successful, then a more shallow one is performed, which adapts and integrates existing linguistic resources (Brill tagger and EuroWordNet) [52].

Since FrOz is a computer game, the utterances entered by the user are naturally restricted to the game scenario. Hence, this task is simpler for FrOz, who handle

it using a TAG grammar (as explained in Chapter 2). In spite of this, we have noticed that FrOz fails too often due to parsing reasons, so it is clear that its grammar and lexicon should broaden its coverage.

Integrating Dialogue and Task Performance: The second problem is how to build a dialogue system that can be adapted easily to almost any practical task. Given the range of applications that might be used, from information retrieval, to emergency relief management, to tutoring, to e-commerce, no strong constraints on what the application program looks like can be accepted. The challenges here lie in designing a generic system for practical dialogue together with a framework in which new tasks can be defined relatively easily.

Both dialogue systems, TRIPS and ADVICE choose to approach this problem using an agent-based framework, where a generic agent or group of agents serves as the underlying structure for the interaction. This generic system, which originally is applicable across different practical domains, is specialised to the particular domain by integrating domain-specific information. However, due to their different development goals, TRIPS approach is more problem oriented while ADVICE approach is more knowledge oriented.

TRIPS implements an *abstract-problem solving model* that defines the key concepts: objectives (goals, sub-goals and constraints), solutions (courses of action towards the objectives), resources (objects and abstractions available for solutions), and situations (representations of the state of the world). Utterances in practical dialogue are interpreted as manipulations (creating, modifying, etc.) of these concepts. A *domain specific task model* provides mapping from the abstract problem solving model to concepts and operations in the particular domain.

ADVICE implements a *knowledge-area organisation model* that organises the knowledge in a hierarchy of knowledge area, each one defining a body of expertise that encapsulates both task and domain knowledge. This approach is supported by a methodology and a tool (Knowledge Structure Manager) [51]. The instantiation of this generic model for a particular domain (such as the professional craft domain) produces the *domain model*. Building a domain model implies the specification of the content of the different knowledge bases identified in the generic model for every primary knowledge area.

FrOz is a dialogue system that is limited to processing natural language commands. While inside this simple practical dialogue subclass, the game can be easily adapted to new commands redefining the action database and to new game worlds redefining the scenario. But FrOz has a lot to learn from other practical dialogue subclasses (such as information-seeking dialogues and advice and tutoring dialogues) if it is to achieve a more natural interaction with the player.

Intention Recognition: Dialogue systems need to implement mechanisms for intention recognition, i.e., determine what the user is trying to *do* with a particular utterance. This requires reasoning about the task, to identify which interpretation is the most appropriate given the current situation. Some kind of reasoning is unavoidable if the system is to react in the proper way to the user's input. The statistical techniques, which currently are so popular for Natural Language Processing (NLP), while useful for certain sub-problems such as parsing, do not suggest a solution to deal with problems that require reasoning in context. Even if the system only had to answer questions, it would be necessary to perform intention recognition in order to know what question is truly being asked. This reveals the complexity, and the power, of natural language. Approaches that are based purely on the form of language rather than on its content and context will always remain extremely limited. There are some good theoretical models of intention recognition [34], but these have proved to be too computationally expensive for real-time systems.

In TRIPS, a two-stage process first suggested in [31] is used. The first stage uses a set of rules that match against the form of the incoming speech acts and generate possible underlying intentions. These candidate intentions are then evaluated with respect to the current problem solving state. Specifically, interpretations that would not make sense for a rational agent to do in the current situation are eliminated (e.g.,it is unlikely that the user would want to reintroduce a goal that is already accomplished).

In ADVICE an agent called *interaction agent* is responsible for the recognition of the intention of utterances. This agent includes two components in order to perform this task. On one hand, it includes the *session model* that represents all the static information about the context of the dialogue. On the other hand, it includes a *dialogue manager* that is responsible for keeping the dialogue state up to date, as well as maintaining the dialogue coherence according to valid dialogue-patterns and thread management.

Due to the simple setup of FrOz, it does not implement a complex intention recognition framework. Instead, it aims to just handle certain simple intentional ambiguities such as reference resolution using the information stored in its knowledge bases. For example, if the user enters the command "take the apple" and there are two apples in the game scenario, but the player only knows the existence of one of them, then FrOz will be able to identify this apple as the one the player wants to take.

Mixed-Initiative Dialogue: Human practical dialogue involves *mixed-initiative* interaction, i.e., it involves the dynamic exchange of control of dialogue flow, increasing dialogue effectiveness and efficiency and enabling both participant's needs

to be met. In contrast, in fixed initiative dialogue one participant controls the interaction throughout. If the dialogue is *system-initiative*, the user must answer the specific question the system asks at each point in the dialogue. This can work well for very simple tasks like long-distance dialling. It does not work well, however, when you need to do something a little out of the normal path. On the other extreme, a dialogue system can be *user-initiative*. The system would do nothing but interpret the user input until sufficient information is obtained to perform the task. This has the problem that the user may not know what information he still needs to supply. Because of these problems, current dialogue systems need to offer mixed-initiative interaction.

Both in TRIPS and ADVICE, rather than viewing the interaction as a series of commands, the interaction involves defining and discussing tasks, exploring ways to perform the task and collaborating to get it done. Most importantly, all interactions are contextually interpreted with respect to the interactions performed so far, allowing the system to anticipate the users needs and provide responses that best further the user's goals.

In particular, TRIPS architecture includes a component called the Behavioural Agent that determines the system behaviour. When responding to an utterance, the Behavioural Agent considers its own private goals in addition to the obligations it has due to the current state of the dialogue.

A distinguishing characteristic of ADVICE is that user interactions are not only interpreted in the context of the current conversation but other conversations with the same user are also taken into account. User preferences are registered and then used to personalise ADVICE behaviour according to the particular user in order to achieve a more effective interaction.

FrOz is mainly ab user-initiative dialogue system, however it provides some simple forms of initiative. For instance, FrOz provides unrequested (but useful) descriptions each time the player access a new game location. Also, FrOz describes the problem and asks for clarification each time it is not able to execute the requested action. This thesis goal is to add a new form of initiative to FrOz. We want FrOz to be able to execute autonomously actions that are implicit in the player's input. To determine which are these implicit actions, the game needs planning capabilities.

1.4 Planning in Dialogue Systems

Planning has been used to handle different aspects of a dialogue system for several decades now [4]. For example, plan-based models are suitable for recognising speech acts performed by the user of a dialogue system, for inferring her goals, and for cooperating in their achievement [44]. Planning techniques also have been used

in natural language generation: when the system needs to convey large amounts of information using multiple utterances, the organisation of the information into utterance-size units is often viewed as a planning process [47].

The planning assistant TRAINS is a classical example of the use of planning in a dialogue system. The aim of TRAINS is to help the user to solve routing problems in a transportation domain. In such environment, the human and the computer must work together in a tightly coupled way to solve problems that neither of them could manage alone. System and user collaborate in order to build a suitable plan for solving the problem at hand. Then, planning techniques are used to validate the feasibility of the plan that is being constructed, infer user goals and cooperate in their achievement. However the authors report in [23] that, according to their experience in TRAINS, and other dialogue systems, traditional planning (finding courses of action from an initial situation to a goal) turns out not to be suitable in this setting. To start with, the initial state is usually not completely specified because of changing conditions, or simply because it is too big to represent. Similarly, the goals of the plan are also poorly specified. Not only they change over time as the user's concerns and preferences change, but also they typically cannot be extracted and codified in a way suitable for automatic processing. Hence the planning process needs to be tailored and specially implemented for each particular dialogue system.

In this thesis we identify a phenomena that can be directly supported by traditional planning, and we show how it is possible to take advantage of a state-of-the-art planning system to solve it. In the task we are going to tackle, initial states, goals and available actions can be completely specified and an off-the-shelf planner can be used to enhance the dialogue system capabilities.

Chapter 2

Inference in a Dialogue System

This chapter introduces FrOz, the dialogue system we work with during this thesis. We describe a complete cycle of input-output in this game, as well as the knowledge bases it uses for representing the game state. Moreover, we define those elements that constitute a particular game scenario. But before delving into these details, we give an introduction to Description Logics, the formalism used by FrOz for knowledge representation and inference.

2.1 Using Description Logics in a Dialogue System

During this section we will Analise the family of knowledge representation formalisms known as *Description Logics (DL)*. In particular, we will focus our discussion in the Description Logic called $\mathcal{ALCIF}$, the DL chosen by FrOz designers and used in this dialogue system to represent the game scenario knowledge bases and to perform automatic inferences over them. In Sections 2.1.1 and 2.1.2 we will discuss how and why Descriptions Logics were born and we will explain why nowadays they are among the most used representation languages in applications that require knowledge representation and inference services. Section 2.1.3 includes the basic notions and the formal definitions of the DL $\mathcal{ALCIF}$. Finally, Section 2.1.4 discusses DL inference systems focusing on RACER, the DL theorem prover used by FrOz.

2.1.1 Knowledge Representation: From Networks to Description Logics

Knowledge Representation main research goal is to design formalisms that are epistemologically correct, flexible enough in order to express information about a given domain and suitable for automatic processing. In other words, the final aim

of this artificial intelligence research community is to provide world descriptions that can be used efficient and effectively in order to build *intelligent applications*. Here *intelligent* refers to the ability of finding implicit consequences in explicitly represented knowledge. Considering this goal, the knowledge representation approaches proposed so far can be broadly classified in:

***Cognitive* Formalisms.** They try to mimic the processes used by people to perform certain intellectual tasks. In general, these representations were originally developed for particular applications, but many of them evolved into general purpose tools. Usually, the representation language used by them are ad-hoc data structures and the reasoning tasks they provide are also ad hoc procedures that manipulate such structures.

***Logic* Formalisms.** They describe the world in a non ambiguous way using logic. In general, their representation language is equivalent to some fragment of the predicate calculus, and the reasoning tasks consist in verifying logical consequences. Hence from the beginning, these formalisms were general purpose tools.

Two classic examples of cognitive formalisms are *Semantic Networks* [46] and the *Frames Paradigm* [43]. Both tools were born from the idea of developing systems based in a structured representation of knowledge. And, although there are important differences between them, they share a common basis. In fact, they can be both considered as *Network-based Structures*. Network-based structures regard the notions of concepts, individuals and relationships between them, as the cornerstones for knowledge representation. Under this approach, a domain expert should specify the key concepts in the domain as well as characterise the relationships between them. Moreover, peculiar individuals should be pointed out as examples or counterexamples of certain concepts or relationships. Afterwards, the concepts could be organised in a hierarchy, which is not only a compact representation of the information, but also allows for an efficient performance of certain reasoning tasks (such as "which concepts are more specific than others" or "which attributes are inherited").

Owing to their more human-centred origins, the network-based systems were often considered more appealing and more effective from a practical point of view than the logical systems. Unfortunately they were not fully satisfactory because of their lack of precise semantic characterisations. The end result of this lack of formal semantics was that every network based system behaved differently from the others, in many cases despite virtually identical components and relationships. The question then arose as to how to provide semantics to representation structures.

Consider the following example:

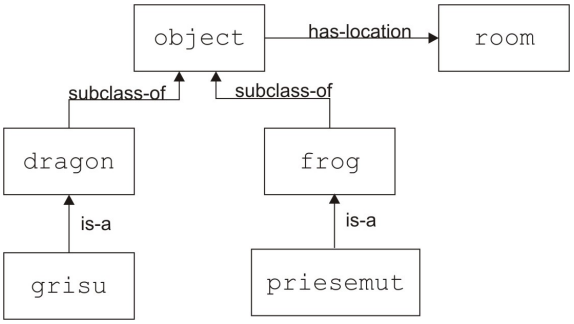

Figure 2.1: A semantic network

The semantic network in Figure 2.1 is meant to encode the following information: Grisu is a dragon, Priesemut is a frog, dragons and frogs are objects, and each object has location in some room.

As pointed out by Wood in [64] with respect to semantic networks, the arcs between the nodes and the nodes themselves can represent different kinds of information, and making the difference explicit can be crucial. Woods defines two main types of arcs, that he identifies as either encoding *intentional* or *extensional* information. An arc that contributes to the definition of a concept carries intentional knowledge. The arc labelled `has-location` between `room` and `object` is of this kind. On the other hand, the arc from `grisu` to `dragon`, labelled `is_a`, asserts the fact that Grisu is a dragon, which Wood classifies as extensional knowledge. But things are more subtle, because also the nodes carry different information: `dragon` is a class, while `grisu` is a distinguished individual. And furthermore, the intentional relation between `dragon` and `object` is of a different kind that the intentional relation represented by `has-location`. The distinction between extensional and intentional links, neither exhausts nor characterises all the possibilities. The need for a formal semantics was clear.

Logic formalisms developed for knowledge representation are characterised for their precisely defined semantics, while cognitive formalisms are considered more intuitive and user friendly. Hence, in order to take advantage of the strong points of both formalisms, network structures were provided with a semantic based on first order logic. This approach gave rise to theoretical research on what were first named *Terminological Languages*, and which are today called *Description Logics*.

The ancestor of today DL systems is KL-ONE [15], which signalled the transition from semantic networks to more well-founded terminological (description) logics. The next generation of DL systems resulted from a detailed research on the trade-off between the expressiveness of DL languages and the computational complexity of reasoning with them (e.g., [14]). As a result, it turned out that frames

and semantic networks did not require all the machinery of first-order logic, but could be regarded as fragments of it. The most important consequence of this fact is the recognition that the typical forms of reasoning used in network-based representations could be accomplished by specialised reasoning techniques, without necessarily requiring all the machinery of first-order logic theorem provers. Moreover, reasoning in different fragments of first-order logic leads to computational problems of differing complexity. Another important consequence of this work is that it became evident that different modelling tasks can differ in their expressivity requirements and that it is crucial to use the description language suitable for the given task. That is why DL researchers study not a single language but a *family of languages* defined in terms of the operators supported by each language.

This is how Description Logics were born as a family of formal languages with a clearly specified semantics. The languages in this family are characterised by a good balance between expressive power and efficiency when computing inference tasks.

2.1.2 Description Logics in Natural Language Processing

Description Logics were born as general purpose languages for knowledge representation, and hence they are appropriate for different kinds of applications. Moreover, they are particularly well suited for those domains where it is useful to organise concepts in a hierarchy or *ontology*.

The way in which DL are used in complex applications has gone through a radical evolution during the last decades. The first DL systems main characteristic was their scarce interaction with other systems or technologies. During this period, DL was used in *Software Engineering* to assist the developers in the management of the internal structure of very big systems. Other area where DL was successfully used was for *Configuration Tasks* that consist in finding a component set that can be combined to satisfy a given problem. In these systems, DL was used to codify the whole problem.

In more recent applications, DL is used to model only part of the problem, the DL system is a component of a bigger environment while other functionalities of the application are implemented more effectively by other technologies. In such an architecture, the DL component is in charge of providing some kind of reasoning services. This is the case of most applications in the relatively new area known as *Semantic Web* where the DL language called *OWL* is used for knowledge representation. Moreover, in the area of *Natural Language Processing (NLP)*, several applications use DL systems at different levels (from the representation of background knowledge up to the codification of the lexical semantics). The dialogue system FrOz is an example of such an application.

NLP is a classical DL application area. In fact, semantic networks and frames already had NLP as a main application. Several NLP projects that use DL have been developed and some of them even reached the status of commercial products (such as The Ford's Direct Labor Management System [48]). Originally, DL was used in order to build the semantic representations of utterances. DL languages were used to codify both the syntactic and semantic knowledge needed to guide the construction of the semantic representations. On one hand, lexical information was stored in the knowledge base, relating words and syntactic properties with basic abstract concepts. On the other hand, background information needed to be codified as well in order to disambiguate the lexical information.

This kind of system as implemented during the 80s and the early 90s (e.g. [53, 1]). Later, the linguistic community interest focused in statistic methods and the work in purely symbolic methods decreased. Recently, however, there has been a resurgence of the use of DL in NLP due to the great technological advances in DL theorem proving (nowadays, there are DL automatic theorem provers that can efficiently deal with extremely expressive DL languages). Currently, the approach is to use DL as a representation language for some particular tasks in NLP (e.g. [26, 54, 39]) in complex systems that can also use statistic techniques for other tasks. Today, one of the main challenges in NLP is to understand how to establish a suitable interface between symbolic and statistic methods.

2.1.3 The $\mathcal{ALCIF}$ Language

As already noted, the DL research community studies a family of languages for knowledge representation. We will now formally introduce the syntax and semantics of $\mathcal{ALCIF}$, which is the formalism used by FrOz to codify its knowledge bases. We will also define the inference tasks that are required by FrOz natural language processing modules.

$\mathcal{ALCIF}$ Syntax

The syntax of $\mathcal{ALCIF}$ is defined in terms of three infinite countable disjoint alphabets. Let CON be a countable set of *atomic concepts*, ROL a countable set of *atomic roles* and IND a set of individuals. Moreover, $\mathsf{FUN} \subseteq \mathsf{ROL}$ is the set of *functional atomic roles*.

We will define the language in three steps specifying the complex concepts and roles that can be defined in $\mathcal{ALCIF}$ as well as the kind of definitions and assertions that are allowed in an $\mathcal{ALCIF}$ knowledge base.

To begin with, we define the $\mathcal{ALCIF}$ operators that let us construct complex concepts and roles from atomic ones.

Definition 1. A *role* can be:
- An atomic role R such that $R \in \mathsf{ROL}$
- The inverse of an atomic role: R^{-1} such that $R \in \mathsf{ROL}$

A *concept* can be:
- An atomic concept C such that $C \in \mathsf{CON}$
- $\top$, the trivial concept called *top*
- Concepts defined using Boolean operators: Let C and D be two concepts then the following expressions are also concepts: $\neg C$, $C \sqcap D$, $C \sqcup D$
- Concepts defined using existential and universal quantified roles: Let C be a concept and R a role then the following expressions are also concepts: $\exists R.C$, $\forall R.C$

We can see that $\mathcal{ALCIF}$ is not very expressive with respect to complex roles. The language offer a richer operator variety when defining complex concepts.

Example 1. Examples of concepts in $\mathcal{ALCIF}$ are:

1. $\exists \texttt{has-location}^{-1}.\texttt{player}$ such that $\texttt{has-location} \in \mathsf{FUN}$ and $\texttt{player} \in \mathsf{CON}$
2. $\texttt{generic-container} \sqcap \forall \texttt{has-location}^{-1}.\neg\top$ such that $\texttt{has-location} \in \mathsf{FUN}$ and $\texttt{generic-container} \in \mathsf{CON}$

Now we can specify which are the kinds of definitions that can be included in an $\mathcal{ALCIF}$ knowledge base.

Definition 2. Given two concepts C and D there are two kinds of *definitions*:

1. *Partial Definitions*: $C \sqsubseteq D$. The conditions specified in C are sufficient in order to qualify C elements as D members, but they are not necessary conditions.
2. *Total Definitions*: $C \equiv D$. The conditions specified in C are necessary and sufficient to qualify C elements as D members, and viceversa. The concepts C and D are equivalent.

Notice that total definitions can be written in terms of partial definitions ($C \equiv D$ iff $C \sqsubseteq D$ and $D \sqsubseteq C$), and viceversa ($C \sqsubseteq D$ iff $C \equiv D \sqcap E$ such that E is a new concept).

The set of definitions in a knowledge base K is called the *TBox* (*Terminological Box*) and it contains definitions of the primitive and derived notions, and how they relate among them. This constitutes, knowledge about the domain that holds in all situations. Formally, an $\mathcal{ALCIF}$ TBox is a finite set of $\mathcal{ALCIF}$ definitions.

Example 2. Examples of $\mathcal{ALCIF}$ definitions are:

1. $\texttt{here} \equiv \exists \texttt{has-location}^{-1}.\texttt{player}$ such that $\texttt{has-location} \in \mathsf{FUN}$ and $\{\texttt{here}, \texttt{player}\} \subseteq \mathsf{CON}$
2. $\texttt{empty} \sqsubseteq \texttt{generic-container} \sqcap \forall \texttt{has-location}^{-1}.\neg\top$ such that $\texttt{has-location} \in \mathsf{FUN}$ and $\{\texttt{empty}, \texttt{generic-container}\} \subseteq \mathsf{CON}$

Finally, we will define the kinds of assertions that can be included in an $\mathcal{ALCIF}$ knowledge base.

Definition 3. *Assertions* let us *assign properties* to particular elements in the domain. Suppose that $a, b \in \mathsf{IND}$ are two individuals, C is a concept and R is a role, there exists two kinds of assertions:

1. Assign elements to concepts: the assertion $a{:}C$ specifies that the concept C is applicable to the element a. I.e., all the conditions specified by C are applicable to a.
2. Assign relationships between elements: the assertion $(a, b){:}R$ specifies that the elements a and b are related via the role R.

The set of assertions in a knowledge base K is called the *ABox* (*Assertional Box*) and it contains specific information about certain distinguished individuals in the modelled domain. Formally, an ABox is a finite set of $\mathcal{ALCIF}$ assertions.

Example 3. Examples of $\mathcal{ALCIF}$ assertions are:

1. $\texttt{myself:player} \sqcap \texttt{not-so-easy-to-kill}$ such that $\texttt{not-so-easy-to-kill} \in \mathsf{CON}$, $\texttt{player} \in \mathsf{CON}$ and $\texttt{myself} \in \mathsf{IND}$
2. $\texttt{(myself,couch1):has-location}$ such that $\texttt{has-location} \in \mathsf{FUN}$, $\texttt{myself} \in \mathsf{IND}$ and $\texttt{couch1} \in \mathsf{IND}$

$\mathcal{ALCIF}$ Semantics

Definition 4. An interpretation (or model) for the $\mathcal{ALCIF}$ syntax is a pair $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$. $\Delta^{\mathcal{I}}$ is the domain, an arbitrary non empty set that can be infinite, while $\cdot^{\mathcal{I}}$ is an interpretation function of atomic concepts, atomic roles and individuals such that:

- Atomic concepts are interpreted as subsets of the domain: Let $C \in \mathsf{CON}$ then $C^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$.
- Atomic roles are interpreted as sets of pairs of elements in the domain: Let $R \in \mathsf{ROL}$ then $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. Moreover, if $R \in \mathsf{FUN}$ then $R^{\mathcal{I}}$ is a partial function.
- Each individual $a \in \mathsf{IND}$ is interpreted as an element in the domain: Let $a \in \mathsf{IND}$ then $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$.

Given an interpretation $\mathcal{I}$, the concepts C and D, and a role R we can define the semantic of the three language levels introduced in the previous section. We will begin by extending $\mathcal{I}$ to complex roles and concepts:

An inverse role is interpreted recursively: $(R^{-1})^{\mathcal{I}} := \{(b,a) \mid (a,b) \in R^{\mathcal{I}}\}$

An arbitrary concept is interpreted recursively as follows:

- $(\top)^{\mathcal{I}} := \Delta^{\mathcal{I}}$
- Boolean operators:

$$(\neg C)^{\mathcal{I}} := \Delta^{\mathcal{I}} \backslash C^{\mathcal{I}}$$
$$(C \sqcap D)^{\mathcal{I}} := C^{\mathcal{I}} \cap D^{\mathcal{I}}$$
$$(C \sqcup D)^{\mathcal{I}} := (\neg(\neg C \sqcap \neg D))^{\mathcal{I}}$$

- Relational operators:

$$(\exists R.C)^{\mathcal{I}} := \{a \mid \exists b.(a,b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\}$$
$$(\forall R.C)^{\mathcal{I}} := (\neg(\exists R.\neg C))^{\mathcal{I}}$$

Now we can define when an interpretation:

- $\mathcal{I}$ satisfies a partial definition $C \sqsubseteq D$ iff $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$
- $\mathcal{I}$ satisfies a total definition $C \equiv D$ iff $C^{\mathcal{I}} = D^{\mathcal{I}}$
- $\mathcal{I}$ satisfies an assertion $a{:}C$ iff $a^{\mathcal{I}} \in C^{\mathcal{I}}$
- $\mathcal{I}$ satisfies an assertion $(a,b){:}R$ iff $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}}$

An interpretation $\mathcal{I}$ satisfies a knowledge base $K = \langle T, A\rangle$, such that T is the TBox and A is the ABox (and we write $\mathcal{I} \models K$) iff $\mathcal{I}$ satisfies all the definitions in T and all the assertions in A. K is satisfiable iff there exists an interpretation $\mathcal{I}$ such that $\mathcal{I} \models K$.

Example 4. The previous definitions are illustrated in the following knowledge base that captures the situation in Figure 2.1
Let $K = \langle T, A\rangle$ be a knowledge base such that $\mathsf{CON} = \{\texttt{dragon}, \texttt{frog}, \texttt{object}, \texttt{room}\}$, $\mathsf{FUN} = \{\texttt{has-location}\}$, $\mathsf{IND} = \{\texttt{grisu}, \texttt{priesemut}\}$ and:

- $T = \{\texttt{dragon} \sqsubseteq \texttt{object}, \texttt{frog} \sqsubseteq \texttt{object}, \texttt{object} \sqsubseteq \exists\texttt{has-location.room}\}$
- $A = \{\texttt{grisu} : \texttt{dragon}, \texttt{priesemut} : \texttt{frog}\}$

Under the formal semantics we have just introduced we can verify that K is satisfiable, i.e. it has at least one model $\mathcal{I} = \langle\{\texttt{grisu,priesemut,x,y}\}, \cdot^{\mathcal{I}}\rangle$ such that:
$\texttt{(dragon)}^{\mathcal{I}} = \texttt{\{grisu\}}$,
$\texttt{(frog)}^{\mathcal{I}} = \texttt{\{priesemut\}}$,
$\texttt{(object)}^{\mathcal{I}} = \texttt{\{grisu,priesemut\}}$,
$\texttt{(grisu)}^{\mathcal{I}} = \texttt{grisu}$,
$\texttt{(priesemut)}^{\mathcal{I}} = \texttt{priesemut}$, and
$\texttt{(has-location)}^{\mathcal{I}} = \texttt{\{(grisu,x),(priesemut,y)\}}$.

Reasoning with $\mathcal{ALCIF}$

The notion of satisfiability of a knowledge base that we defined in the previous section is one of the basic reasoning tasks in DL. Given the fact that the knowledge base codifies the information that we know about certain domain, it is at least required that this information is consistent, i.e. satisfiable by at least one model. But apart from checking whether the knowledge base is consistent, we need methods that let us query the information that is implicit in the knowledge base.

A typical inference task of this kind is *subsumption* (usually written $C \sqsubseteq D$). The problem of determining whether $C \sqsubseteq D$ for a given knowledge base K amounts to verifying whether the concept D is more general than the concept C given the definitions and assertions in K. In other words, it involves checking whether the interpretation of C is a subset of the interpretation of D for every interpretation that satisfies K. Formally, $C \sqsubseteq D$ for a given K iff for every interpretation $\mathcal{I}$ that satisfies K it is true that $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$.

Determining this kind of relationship is directly related with the task of information classification. The subsumption relationship implicitly defines a concept taxonomy, an `is_a` hierarchy that can be used for other inference tasks such as concept satisfiability.

Formally, we can define the following standard inference tasks. Let K be a knowledge base, C and D two concepts, R a role and $a, b \in \mathsf{IND}$, we can define the following inference tasks with respect to a knowledge base:

- *Subsumption*, $K \models C \sqsubseteq D$. Verifies whether for all interpretation $\mathcal{I}$ such that $\mathcal{I} \models K$ we have that $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$.
- *Instance checking*, $K \models a : C$. Verifies whether for all interpretations $\mathcal{I}$ such that $\mathcal{I} \models K$ we have that $a^{\mathcal{I}} \in C^{\mathcal{I}}$.
- *Relation checking*, $K \models (a, b) : R$. Verifies whether for all interpretation $\mathcal{I}$ such that $\mathcal{I} \models K$ we have that $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}}$.
- *Concept Consistency*, $K \not\models C = \neg\top$. Verifies whether for some interpretation $\mathcal{I}$ such that $\mathcal{I} \models K$ we have that $C^{\mathcal{I}} \neq \emptyset$.

Example 5. Given the knowledge base K defined in the Example 4, it is possible to infer further information. For instance, we can infer that the concept `object` is consistent with respect to K: there exists some interpretation that satisfies K and that assigns a non empty interpretation for `object` ($K \not\models$ `object` $= \neg\top$).

The basic reasoning tasks can be used to define more complex ones that are useful for implementing applications, such as:

- *Retrieval*, for a given concept C, find all the individuals mentioned in the knowledge base that are instances of C.

- *Most specific concepts*, for a given individual a mentioned in the knowledge base, find the most specific concepts in the knowledge base such that a is a member of them.
- *Immediate ancestor (descendant) concepts* for a given concept C, find all the concept immediately above (under) C in the hierarchy defined by the knowledge base.

For the DL $\mathcal{ALCIF}$, the inference tasks we defined in this section are very complex. For example, subsumption checking of two concepts with respect to an arbitrary knowledge base is a complete problem for the complexity class ExpTime (exponential time).

2.1.4 Reasoners: Services vs. Complexity

The DL research area is characterised by a tight interaction between theory and practise. The detailed theoretical work on the expressivity and complexity of the different languages, is complemented with implementation and testing of inference systems that are capable of providing concrete inference services for those languages.

Nowadays, DL theorem provers include the most important achievements in optimisation techniques and search heuristics for logic languages that are both decidable and show a high complexity.

Historically, we can characterise three different approaches in the implementation of DL theorem provers:

- *Limited and complete* [1]*:* The language constructor set is restricted in such a way that the reasoning can be efficiently handled. The system CLASSIC [13] is the most traditional example in this category.

- *Expressive and incomplete:* These systems try to provide a very expressive language while maintaining the efficiency of the reasoning tasks but not their completeness. Significant examples in this category are the reasoning systems LOOM [41], and BACK [45].

- *Expressive and complete:* These systems have complete reasoning algorithms for highly expressive languages. Example systems in this category are KRIS [7],

[1] An inference system is *complete* for a given task when it always returns an answer (usually the system is assumed to be also correct, i.e., the answers it returns are exact). On the contrary, a system is *incomplete* when it can fail to return some answer even if it exists. Incomplete systems usually implement heuristics that decrease the response time but they cannot explore all the search space.

FACT [32] and RACER [59, 30]. Even if, for these systems, the worst case complexity is very high, they have shown to be extremely efficient in concrete applications.

Although the selection of the kind of reasoning system will be based on the application that is being developed, nowadays the most advanced systems belong to the third category. The text game adventure analysed during this thesis, FrOz, uses the theorem prover RACER that is, nowadays, the most efficient complete reasoner for the logic called $\mathcal{ALCIFQH}_{\mathcal{R}^+}(\mathcal{D}^-)$, an extremely expressive language that extends the previously described language $\mathcal{ALCIF}$.

RACER

RACER was developed at the University of Hamburg by Haarslev and Möller. It is implemented in COMMON LISP and it is available for research purposes as a server program that can be used both in Windows and Linux. RACER offer inference services to client applications through an http interface. Recently, RACER has became a commercial product that sells its services through the RACER SYSTEMS GmbH & Co company. Moreover, there are implementations of graphical interfaces for taxonomy edition that can connect with RACER. RICE [60] is an example of such interfaces.

RACER offers different inference services including the ones described in Section 2.1.3. Moreover, this inference engine supports multiple TBoxes and ABoxes. Furthermore, it allows for the addition and retraction of ABox assertions even after queries have been performed (some DL reasoning systems forbid this due to optimisation reasons). All these characteristics are crucial for FrOz performance and motivate the choice of RACER as inference service provider.

2.2 The innards of FrOz

The text game adventure FrOz is a dialogue system developed at the University of Saarbrücken [39]. Why is FrOz a dialogue system? Because all the interaction between player and game is done in natural language. The game can understand commands the player presents as English sentences which verbalise actions that she (the player) wants to execute in the game world (such as "Open the door with the key"). FrOz executes the specified actions in the game world, and reports back to the player (by automatically generating natural sounding English text) on the changes introduced in the current situation (for example "The door is open").

As a dialogue system, FrOz general architecture follows a standard pipeline [10] composed by six modules. In such architecture, depicted in Figure 2.2, a cycle of input-output in the game can be described briefly as follows. First, the player's

input is parsed by the *parsing module*. This yields a semantic representation specifying the action that the player wants to execute, and also describing the objects that this action involves. Next, the object descriptions are resolved to individuals of the game world by the *reference resolution module* obtaining a ground term that specifies the action intended by the player. During the third step, the *actions module* looks up this action in an *action database*, checks whether its preconditions are met in the world, and, if so, updates the world state with the effects of the action. The changes introduced in the current situation are then reported to the player through natural sounding English text, which is automatically generated by the remaining three modules: *content determination*, *reference generation* and *realisation*. FrOz implements state-of-the-art techniques from computational linguistics for each one of these modules. We will describe each of them in detail in Section 2.2.2.

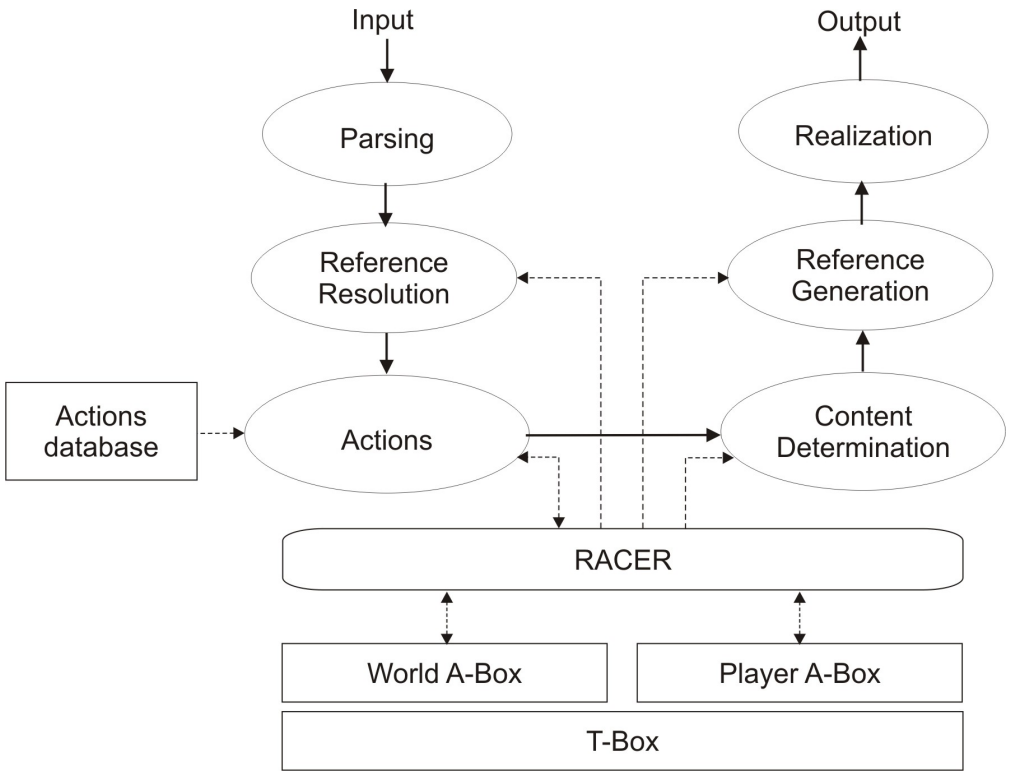

Figure 2.2: FrOz original architecture

The game can be instantiated with different *scenarios*. We describe the elements that constitute a scenario in Section 2.2.3.

FrOz uses Description Logic knowledge bases to codify the information concerning the state of the game. FrOz knowledge bases are accessed by almost all

modules in the pipeline (see Figure 2.2) via queries sent to the RACER reasoner. In the next section we will describe how FrOz codifies the knowledge it needs.

2.2.1 FrOz Knowledge Bases

As we mentioned before, most of the information defining a particular FrOz scenario is encoded as DL knowledge bases. In fact, underlying the system there are two knowledge bases, which share a set of common definitions: the T-Box; and differ only in their set of assertions: the A-Boxes. The common T-box defines the key notions in the world and how they are interrelated. Some of these notions are basic concepts (such as *object*) or properties (such as *alive*), directly describing the game world, while others define more abstract notions like the set of all the individuals a player can interact with.

One of the knowledge bases (the *world A-Box*) represents the *true state* of the world, while the other (the *player A-Box*) keeps track of the player's *beliefs* about the world. The assertions listed in the player A-Box will typically be a strict subset of the assertions in the world A-Box because the player will not have explored the world completely and therefore will not know about all the individuals and their properties. It may happen, however, that some effects of an action are deliberately hidden from the player; for example, if pressing some button in a room has some effect in another room which the player cannot notice. In this case, the player A-Box may actually contain information that is inconsistent with the world A-Box.

The A-Boxes specify the kind of an individual (for example, an individual can be an *apple* or a *player*) detailing to which concept an individual belongs to. Relationships between individuals in the world are also represented here (such as the relationship between an object and its location). A fragment of an example A-Box describing a possible state of the world in the FairyTaleCastle scenario is:

```
room(empfang)                          alive(myself)
player(myself)                         alive(worm1)
frog(priesemut)                        alive(priesemut)
crown(crown2)                          has-location(myself,couch1)
apple(apple1)                          has-location(priesemut,table1)
apple(apple2)                          has-location(apple1,couch1)
worm(worm1)                            has-location(apple2,couch1)
couch(couch1)                          has-location(couch1,empfang)
table(table1)                          has-location(table1,empfang)
exit(drawing2treasure)                 part-of(crown2,priesemut)
has-exit(empfang,drawing2treasure)     part-of(worm1,apple1)
green(apple1)
```

A graphical representation of the relations represented in this A-Box fragment is given in Figure 2.3. Individuals are connected to their locations via the `has-location` role. Objects are connected with things they are part of via the role `part-of` (e.g., `part-of(worm1,apple1)`).

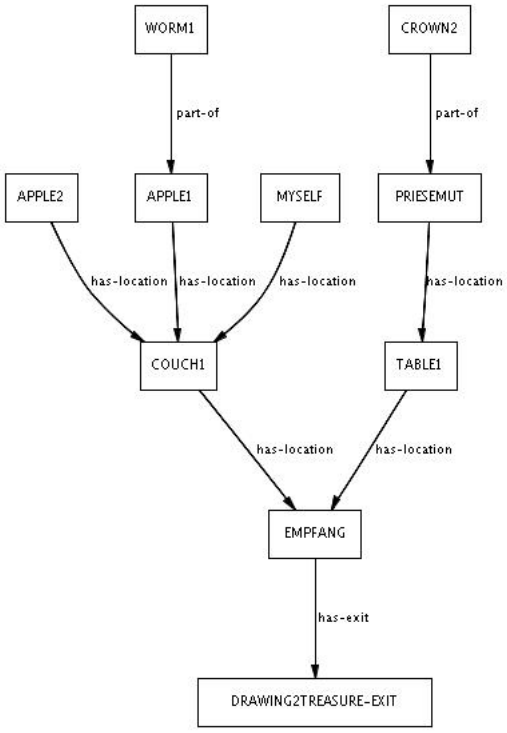

Figure 2.3: Graphical representation for roles in the game A-Box

The T-Box specify that the world is partitioned into three main concepts: generic containers, objects and things that can be opened and closed. Properties that can change in the world such as `alive` are also defined as concepts. The T-Box contains as well axioms that establish a taxonomy between concepts such as:

`takeable` $\sqsubseteq$ `object`
`apple` $\sqsubseteq$ `takeable`
`exit` $\sqsubseteq$ `open-closed`
`room` $\sqsubseteq$ `generic-container`
`player` $\sqsubseteq$ `generic-container`

A graphical representation of a T-Box fragment for FrOz FairyTaleCastle scenario is given in Figure 2.4.

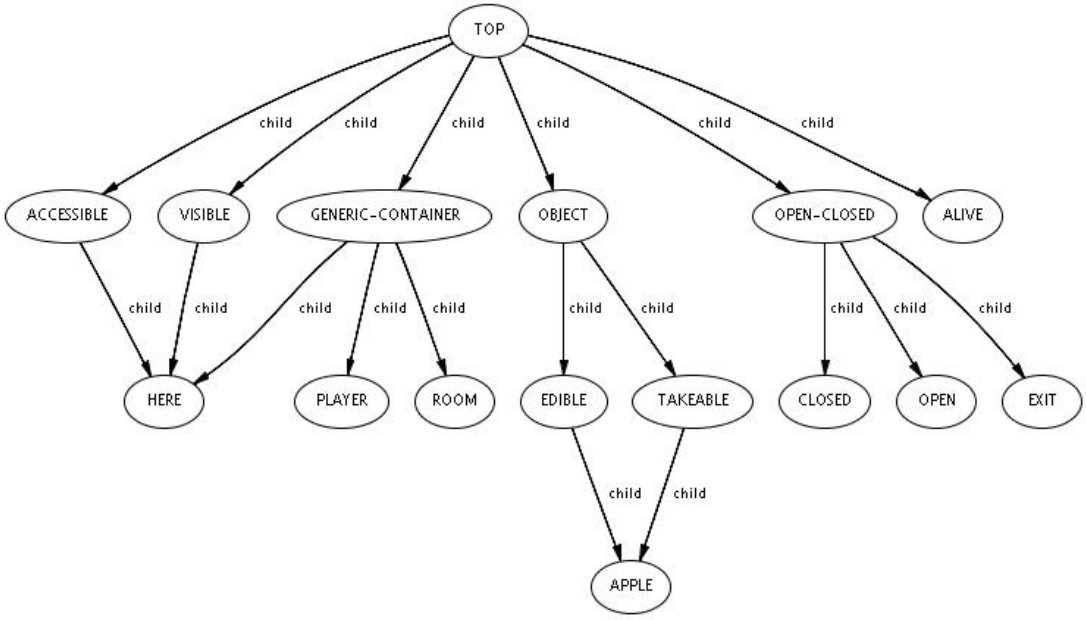

Figure 2.4: Graphical representation of a fragment of the game T-Box

On top of the basic definitions, the T-Box specifies more abstract concepts that are useful in the game context. For example, the concept `here`, which contains the room in which the player is currently located, is defined as:

$$\texttt{here} \equiv \exists \texttt{has-location}^{-1}.\texttt{player}$$

In the example A-Box we introduced for the FairyTaleCastle scenario, `here` denotes the singleton set `couch1`. It is the only individual to which an instance of `player` is related to via the role `has-location`. Another important concept in the game is `accessible`, which contains all individuals that the player can manipulate.

$$\begin{aligned}\texttt{accessible} \equiv\ & \texttt{here} \sqcup \\ & \exists \texttt{has-location.here} \sqcup \\ & \exists \texttt{has-location.(accessible} \sqcap \texttt{open)} \sqcup \\ & \exists \texttt{part-of.accessible}\end{aligned}$$

This DL definition means that the location where the player is currently standing is accessible to him, as well as the individuals that are in the same location.

If such individual is some kind of container and it is open then its contents are also accessible; and if it has parts, its parts are accessible as well. In the FairyTaleCastle A-Box we introduced, the denotation of `accessible` will include the set `{couch1,myself,apple1,apple2,worm1}`.

Finally, the concept `visible` can be defined in a similar way as `accessible`. The definition is a bit more complex, including more individuals and is intended to denote all individuals that the player can see from his position in the game world. In the FairyTaleCastle A-Box we introduced, `visible` denotes the set of all the objects in the world A-Box.

2.2.2 FrOz Step by Step: a Complete Example

Let us now analyse in detail a complete cycle of input-output in the game, based on the architecture sketched in Figure 2.2. We will describe in more detail the Actions module since it is the most involved in the enhancement proposed in this thesis.

Suppose that the user inputs to the game the natural language instruction: "Take the green apple". The first step the game performs is to parse this input.

The Parsing module uses a parser for Topological Dependency Grammar (TDG) to perform the syntactic analysis (see [20] for further details on TDG). The output of the TDG parser is a syntactic dependency tree which represents the syntactic analysis of the sentence. The syntactic dependency tree for "Take the green apple" would be:

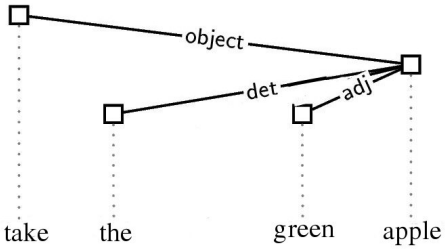

Figure 2.5: A syntactic dependency tree

From the syntactic dependency tree, the desired semantic representation of the input sentence is computed. During its computation, whenever a noun node is encountered some further information is recorded (agreement, linear position within the sentence, and (in)definiteness) for the purposes of reference resolution (the next module in the cycle). The final representation of the semantic construction for our example is the following:

```
take(patient:[apple(agr:[unit(gender:[neut]
                              number:[sing]
                              spec:[def])]
                    nmod:[green])
```

This semantic representation specifies the action that the user wants to execute, and also describes the objects that this action involves. However, it still refers to an individual in the world through a formalised version of the noun phrase (NP) in the input sentence. The next module will be responsible of mapping this representations to an individual in the DL knowledge base, so that it can be used internally.

If the input sentence is syntactically ambiguous then the output of this module will contain one entry in the above notation for each one of the readings.

The Reference Resolution module is responsible for resolving nouns and pronouns into concrete objects in the game world. It makes use of RACER inference system to retrieve individuals that match the player's descriptions and employs a simple discourse model which keeps track of available referents to resolve pronouns and ambiguous definite NPs. In our example, the resolution of a *definite* description is needed (the apple); this amounts to finding a unique entity which, according to the player knowledge, is visible and matches the description. To compute such an entity, a DL concept expression corresponding to the description is constructed and then a query is send to RACER asking for all instances of this concept. In the case of our example, all instances of the concept

$$\texttt{apple} \sqcap \texttt{visible} \sqcap \texttt{green}$$

would be retrieved from the player knowledge base. If such a query yields only one entity ($\{\texttt{apple1}\}$ for "the green apple" from the knowledge base in the example A-Box we introduced in the previous section), the reference has been unambiguous and succeeds. It may be the case, however, that more than an entity is returned. For instance, (given the same knowledge base) the query for 'the apple' would return the set $\{\texttt{apple1, apple2}\}$.

In such a situation, referents which are not *salient* according to the actual discourse model[2] are filtered out. If this narrows the candidate set to a singleton, we are done. Otherwise, we assume that the definite description was not unique and return an error message to the player indicating the ambiguity.

To resolve *indefinite* NPs, such as 'an apple', the player knowledge base is queried as described above. However, unlike the definite case, a unique referent is not required in this case. Instead it is assumed that the player did not have any

[2]The discourse model used in the game was inspired by Strube's salience list approach, further information about it can be found in [55].

particular object in mind and one of the possible referents is arbitrarily chosen (and the choice will be informed to the player by the generation module).

After all references has been resolved, a ground term (or sequence of ground terms) that specifies the action (or actions) intended by the player is passed to the Actions module. In our example, this representation would be:

```
[[take(patient:apple1)]]
```

The Actions module receives this list of lists of instantiated action descriptions. The outer list contains one entry for each reading of a syntactically ambiguous input sentence. These entries are themselves lists, which represents sequencing of actions which are to be performed one after the other (for example, if the input contains more than one action to be executed, as in: 'Take the green apple and eat it').

When the list contains just one reading, such as in our original example, then the entry is matched against a list of action representations in a database which specifies the action preconditions and effects. This database can be seen as the codification of the 'instructions' that guides the actions module in fulfilling its task. The actions module uses RACER to perform the necessary inferences in order to follow these instructions. The database is specified in a STRIPS-like format [24] and it divides the effects of an action into those that modify the world A-Box (*effects*) and those that modify the player A-Box (*player beliefs*) when the action is executed.

An example of an entry in the actions database is given below:

`take(patient:X)`	

preconditions	`instance(X accessible),`
	`instance(X takeable),`
	`not(instance(X inventory-object))`

effects	*add:*
	`instance(X inventory-object)`
	delete:
	`related(X`
	`individual-filler(X haslocation)`
	`haslocation)`

player beliefs	*add:*
	`instance(X inventory-object)`
	delete:
	`related(X`
	`individual-filler(X haslocation)`
	`haslocation)`

The term `X` in the action representation shown in the figure is a variable that gets bound to the actual argument that the resolution module computed. In our example, `X` would be bound to the constant `apple1`, and thus the preconditions and effects of the operators will become ground terms.

Let us see in detail how to read the specification of the action `take` when applied to our example. The command "Take the green apple" issued by the player requires that the apple is accessible to the player (`instance(apple1 accessible)`), that it is small enough to be taken (`instance(apple1 takeable)`) and that it is not carried by the player already (`not(instance(apple1 inventory-object))`). When this command is executed, the apple becomes an object in the player's inventory (`instance(X inventory-object)`) and it is no longer located where it used to be. This last effect includes an expression that requests RACER to return the individual in the world that represents the location of the apple (`individual-filler(apple1 has-location)`). A RACER expression is embedded in the action specification when the action cannot be specified completely in advance because it depends on the current state of the game (as is the case for most interesting actions). Finally, for this action, the effects on the player beliefs are identical to the effects on the world state.

If the player input requires the execution of more than a single action, we need to take some extra care. Let's consider the example we mentioned before, where the player's input is "Take the green apple and eat it". The input to the actions module would then be something like `[[take(patient:apple1), eat(patient:apple1)]]`. In this situation the action module instructs the theorem prover to create a copy of the current world A-Box and then the actions are performed in sequence on this copy. Namely, if the first action succeeds, then its effects are incorporated into the copy and the preconditions of the next action in the sequence are evaluated with respect to this updated A-Box. If one of the actions in the sequence fails, then the A-Box copy is discarded and an error message is reported to the player. Otherwise, if all the actions in the sequence succeed then the original world A-Box is replaced by the updated copy.

If the input sentence was ambiguous, the actions module tries to decide which of the readings the user intended by trying each action sequence in parallel. In order to do this, the action module instructs the theorem prover to create a copy of the current world A-Box for each reading. Then the actions in each reading are performed in sequence on its own copy. If only one possible sequence succeeds, the action module assumes this is the command the player had in mind, and commits to the end result of this sequence (i.e. the original world A-Box is replaced by the copy updated with the effects of the successful sequence). Otherwise, if more that one sequence is possible, it reports that the input sentence could not be disambiguated; if none is successful, an error message is printed.

The Content Determination module is the first module of the generation phase of the game, whose task is to produce texts in reaction to the user input to let her know what the game world currently looks like and how it was affected by the actions that were executed. The input for this component is the instantiated user knowledge slots of the actions the game just performed.

Content determination is the task to decide what to say to the player. In many cases, it is enough to pass to the next module the facts that the action *adds* to the player beliefs when it is executed; assuming that the player can infer the information that is *deleted*. In our ongoing example, this part would contain the following list:

```
[has-location(apple1 myself)]
```

However, there are actions for which some further work has to be done. In particular, the user knowledge slots of some actions contain literals of the form `describe(X)`. Such literals are interpreted as an instruction to this module to generate a detailed description of the individual `X`. This kind of instructions are needed to handle actions like *look* which have no effects on the world and only modify the player's knowledge. Another example are actions like move, which put the player into a new room and beside updating the world state should produce a description of the new location.

For example, if the player says "Look at the green apple", the instantiated knowledge slot of this action would be as follows:

```
add:  [describe(apple1)]
delete:  nil
```

Given such input, the content determination module replaces `describe(apple1)` by a list of properties that `apple1` has. In order to do this it queries RACER to return all most specific concepts that `apple1` belongs to in world A-Box, as well as some role assertions in which `apple1` participates that are meaningful to the player. It will then group this information into different sentences to be verbalised, one for each concept and role. The result for the input "Look at the green apple" would be:

```
[content(goal: l1
         sem: [apple(apple1) green(apple1)])
 content(goal: l2
         sem: [has-detail(apple1 worm1)])]
```

The Reference Generation module generates natural-language NPs that refer to individuals like `apple1` in a way that is meaningful to the player.

To refer to an object that the player already has encountered, a definite description is constructed that, given the player beliefs, uniquely identifies this object. The properties of the target referent are looked in some predefined order (e.g., first its type, then its colour, its location, parts it may have, and so on). A property is added to the description if at least one other object (a *distractor*) is excluded from it because it doesn't share this property. This is done until the description uniquely identifies the target referent. Once more, RACER queries on the player A-Box are used to compute the properties of the target referent and the distracting instances, and to check whether a given property is of a certain kind (e.g., colour).

Following the cycle of our original example, "Take the green apple", the content that needs to be verbalised for it is:

```
content(goal: l1
        sem: [apple(apple1) def(apple1) green(apple1)
              has-location(apple1 myself)])
```

The reference generation task is simpler for objects which are new to the player (newness can be determined by querying whether the individual is mentioned in the player A-Box). In this case, an indefinite NP containing the type and (if it has one) colour of the object is generated. RACER retrieval functionality is used to extract this information from the world A-Box.

In our example "Look at the green apple", if the player knows that there are two apples and that the second apple is red but she doesn't know about the worm, the second sentence to verbalise would be enriched as follows:

```
content(goal: l2
        sem: [has-detail(apple1 worm1) worm(worm1) apple(apple1)
              undef(worm1) def(apple1) green(apple1)])
```

The message now contains the information that an indefinite reference to `worm1` should be built, referring to it as "a worm". `apple1` should be referred to by the definite description "the green apple". The colour was added to distinguish it from the other apple which is red.

The Realisation module is responsible of casting into a text all the information previously gathered. This is done sentence by sentence, using a standard surface realisation algorithm for tree-adjoining grammars (TAG) (see [33]). This module starts by selecting trees that can be used to exactly cover the list of facts constructed in the previous modules. It then uses the TAG operations of substitution and adjunction to create a grammatically correct sentence expressing the desired semantic content.

The message output by FrOz for the input "Take the green apple" which closes the cycle is:

```
You have the green apple
```

While the message for the command "Look at the green apple" would be:

```
The apple is green
The green apple has a worm
```

verbalising in this way the inputs received from the previous module.

2.2.3 FrOz Scenarios

FrOz can be instantiated with different *scenarios*. The functionality offered by the different modules is shared by all scenarios, while each scenario has its own elements where it codifies its specific characteristics. The elements that constitute an scenario are:

- *The scenario T-Box:* Each scenario can define its own concepts and roles in a hierarchy. Such T-Box is static for a given scenario.
- *The world A-Box:* The world A-Box in a scenario contains a definition of the state of the world when the game begins. The world A-Box changes during the game according to the effects of the actions performed by the player.
- *The player A-Box:* The player A-Box in a scenario is empty when the game starts. It is modified by the effects on the user beliefs each time the player performas an action.
- *The action database:* Each scenario can define different actions that can be performed over the world.
- *The Lexicons:* Each scenario defines two lexicons. One for parsing and another for generation. This is actually not convinient because the information is duplicated. The corresponding modules should be modified in order to use the same lexicon.

FrOz current distribution[3] defines two scenarios:

The FairyTaleCastle: This scenario defines 67 concepts in its T-Box and 26 objects in the world A-Box. Moreover it has 28 actions in the action database, 110 lexical entries for parsing and 138 for generation.

The SpaceStation: This scenario defines 51 concepts in its T-Box and 17 objects in the world A-Box. Moreover it has 25 actions in the action database, 101 lexical entries for parsing and 115 for generation.

[3] `http://www.coli.uni-saarland.de/~koller/projects/froz`

Chapter 3

The Role of Planning in AI

The Artificial Intelligence area called Planning investigates control algorithms that synthesise a sequence of actions which achieve some predefined goal. Although Planning is far from being a new topic in Artificial Intelligence, recent developments have revolutionised the field.

In this chapter we begin by presenting a general introduction to planning in Section 3.1. This introduction focuses in modern planning techniques that were developed during the last decades. The following Section describes the Planning Domain Definition Language (PDDL) the standard planning language used for the International Planning Competition since 1998. A subset of PDDL is the language accepted by Blackbox, the planner involved in this thesis implementation.

3.1 Artificial Intelligence Planning

A typical specification of a *planning task* defines three main elements which are usually represented in some formal language:

1) A description of the world *initial state*.
2) A description of the intended world state, the *goal*.
3) A description of the *domain* (the actions allowed over the world).

These elements are usually organised in two parts: the *planning problem* that includes the initial state and the goal, and the *planning domain*.

The output of a *planner* is a sequence of actions such that, when executed in order over the world initial state, achieves the goal. This formulation of a planning task is highly abstract. In fact, it specifies a kind of planning task specifications parametrised by the language used to represent the world state, the goal and the

actions. For example, propositional logic can be used to describe the actions effects but in this way the specification of actions that have universally quantified effects will be complex. Or we can use first order predicate calculus to describe the effects of such actions but this assumes that all the actions have deterministic effects [62]. This is the reason why there exists a wide range of planning specification languages that differ in their expressive power. The decision of which language to use will depend on the problem that we need to model (and also, on the performance requirements of our application).

The first implemented planning system was called STRIPS, designed in the early 70s to control a mobile robot [24]. More than the original planner, the representation language used by STRIPS, and popularly known as the STRIPS language, is still a standard in the area. The STRIPS language describes the initial world state in a planning problem specification as a complete set of ground literals. The goal is defined with a propositional conjunction and all the world states that satisfy the goal are considered equally good. The domain specification (the formal definition of the available actions) completes the planning task specification. Each action is described as a conjunction of preconditions and a conjunction of effects that defines a transition function between world states. An action can be executed in every world state w that satisfies the formula represented by the conjunction of the preconditions. The result of executing an action over a world state w is obtained by taking the description of the world state w, adding at once all the action effects and eliminating contradictory literals when they appear.

Several works were born from this traditional planning representation. In [40], Lifschitz formally analyses the STRIPS language. In [16] a simple algorithm for plan search for STRIPS style specifications is proved to be complete for PSPACE. Fikes and Nilsson [25] discuss the historical background for the STRIPS project and relate it with more modern works in the area.

Most of the planning algorithms proposed in the 70s were incomplete: sometimes they failed to find a solution even if one existed. This behaviour is evidenced in cases such as the one proposed by the "Sussman Anomaly" [56], a simple example in the block world that cannot be solved by the STRIPS algorithm. Systems such as WARPLAN [58] solved this problem using techniques known as "interleaved planning".

The first generic planners appeared only in the late 80s. An example of this planner generation is TWEAK [17] developed by Chapman, who also proved that some planning tasks are undecidable while others are in NP.

3.1.1 The Current State of the Art

Recent developments have revolutionised AI planning algorithms. In particular, the two approaches that have received more attention are introduced in the rest

of this section. Both approaches have much in common and had an impact in constraint satisfaction and search technologies. The current performance of the planners developed following these approaches is outstanding; they can quickly solve problems that are many times more difficult than those used for benchmarking only a few years ago.

Graph-based Planner: Graphplan [11] is a simple, elegant algorithm that yields an extremely speedy planner. Graphplan alternates between two phases: graph expansion and solution extraction. The graph expansion phase extends a planning graph forward in "time" until it has achieved a necessary (but insufficient) condition for plan existence. The solution extraction phase then performs a backward-chaining search on the graph, looking for a plan that solves the problem; if no solution is found, the cycle repeats by further expanding the planning graph.

The original (somewhat dated) C implementation of Graphplan is available at `www.cs.cmu.edu/afs/cs.cmu.edu/user/avrim/www/graphplan`; while IPP [38] is a highly optimised C implementation of Graphplan extended to handle expressive actions (e.g., universal quantification and conditional effects) available at `www.informatik.uni-freiburg.de/~koehler`.

Sat-based Planners: Despite the early formulation of planning as theorem proving [27], most researchers have long assumed that special-purpose planning algorithms are necessary for practical performance. However, recent improvements in the performance of propositional satisfiability methods suggest that compilation to SAT might yield excellent STRIPS-style planners [35].

Figure 3.1 shows the architecture of a typical SAT-based planner. The *compiler* takes a planning task as input, guesses a plan length, and generates a propositional logic formula, which if satisfied, implies the existence of a solution plan; a *symbol table* records the correspondence between propositional variables and the planning instance. The *simplifier* uses fast (linear time) techniques to shrink the propositional formula. The *solver* uses systematic or stochastic methods to find a satisfying assignment which the *decoder* translates (using the symbol table) into a solution plan. If the solver finds that the formula is unsatisfiable, then the compiler generates a new encoding reflecting a longer plan length.

An available implementation of this approach is the Medic planner [21], a flexible testbed, implemented in Lisp, allowing direct comparison of over a dozen different SAT encodings. See `ftp://ftp.cs.washington.edu/pub/ai/medic.tar.gz`.

Hybrid systems: Blackbox [36], one of the best performing planners nowadays, uses a hybrid approach that combines the previous two techniques. GraphPlan bears an important similarity to SatPlan: both systems work in two phases, first

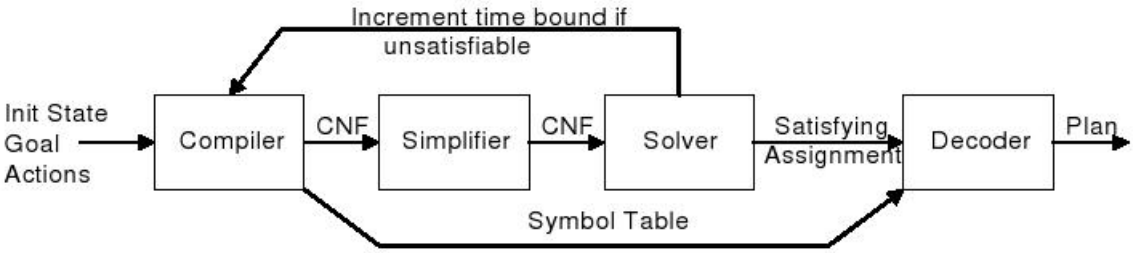

Figure 3.1: Architecture of a typical SAT-based planning system.

creating a propositional structure (in GraphPlan, a plan graph, in SatPlan, a propositional formula) and then searching that structure. The propositional structure corresponds to a fixed plan length, and the search reveals whether a plan of that length exists. In [36] it is shown that the plan graph has a direct translation into propositional logic, and that the form of the resulting formula is very close to the original conventions for SatPlan.

Blackbox was designed as a system that combines the best features of GraphPlan (the use of a better algorithm for *instantiating* the propositional structure) and SatPlan (the use of more powerful *search* algorithms). Blackbox follows the standard SatPlan architecture we introduced above but, as it integrates GraphPlan ideas, it works in three phases:

1. A planning task, specified in PDDL with basic STRIPS-style actions and typing, is converted to a plan graph

2. The plan graph is converted into a propositional formula

3. The formula is solved by any variety of fast SAT engines allowing the use of the engine that is best suited for a particular type of problem.

The performance of Blackbox is impressive. As an example, it is mentioned in [36] that it requires only six minutes to find a 105-action logistics plan in a framework with 10^{16} possible states.

The implementations mentioned so far and, in general, most implementations available today, accept planning tasks specified in the Planning Domain Definition Language (PDDL), the standard planning language used for the International Planning Competition. We will describe this language in the next section.

3.2 Planning Domain Definition Language (PDDL)

PDDL is intended to express the "physics" of a domain, that is, what predicates there are, what actions are possible, and what the effects of actions are. Most planners require in addition some kind of "advice" or control information in order to guide the search they perform. However, PDDL provide no advice notation and its authors explicitly encourage planner developers to extend its basic notation in the way required by the particular planner implementation.

Even without these extensions, few planners will handle the entire PDDL. Hence, this language is factored into subsets of features, called *requirements*. Every domain defined using PDDL should specify which requirements it assumes. Any planner then can easily determine if it will be able to handle certain domain specification. The syntactic requirements supported by PDDL that are relevant for this thesis are: basic STRIPS-style; typing; conditional effects; universal and existential quantification; and domain axioms over stratified theories.

3.2.1 PDDL with Basic STRIPS-Style and Typing

We start our discussion by restricting our attention to the simple PDDL subset that handles STRIPS planning tasks in a deterministic, fully-specified world. In other words, both the preconditions and effects of actions are conjunctions of literals. After covering the basics, we describe the PDDL subsets with added features which offer a more expressive action language.

In what follows, we will describe PDDL syntax while introducing its semantics intuitively through a worked out example. For a complete definition of its semantics see [42]. The planning problem example we will use in this section is based on the FairyTaleCastle scenario provided with FrOz. In the initial state of this example, a door is locked and its key is on a table, and the goal is to arrive to a state where a given door is unlocked.

Before describing how PDDL specifies the planning domain and the planning problem, we will describe the type system supported by PDDL. Types should be defined as part of the planning domain. Typing is important when defining planning tasks because it reduces the search space for those planners that ground the specification before starting the search.

0) Types: PDDL allows for the definition of types that can then be used to type both objects in the planning problem and the arguments of predicates and actions.

In our example we can declare the types:

```
(:types
   key takeable lockable table
   player room container door top)
```

where `top` is a type that is applied to all the individuals in the planning problem in order to symplify the typing of paramenters.

In general, the type definition in PDDL is specified as:

(`:types <type-list>*`)

where `<type-list>` is a list of identifiers (type names).

The predicates involved in the planning task can be declared and the type of their parameters specified as shown in the following example:

```
(:predicates
   (haslocation x - top y - container)
   (locked x - lockable)
   (fitsin x - key y - lockable))
```

The predicate `(haslocation x y)` holds when `x` is located in the container `y`, `(locked x)` holds when the lockable object `x` is locked, and `(fitsin x y)` holds in the world when the key `x` opens the lockable object `y`.

In general, predicate definitions in PDDL are specified as:

(`:predicates {<ident> {<variable> - <type>}*}*`)

where `<ident>` is the predicate name followed by a list of elements formed by a variable name `<variable>` and its type `<type>`.

1) Initial State: Remember that the initial state is a description of the world state when the plan begins. It includes the objects available in the planning problem as well as an specification of which predicates are applicable to them, when the plan begins. Since types are simply a specific kind of predicate which have the particularity that cannot be affected by actions effects, object types are also specified here.

Returning to our example, the object and initial state definitions will look as follows:

```
(:objects
   empfang - (either room container top)
   myself - (either player container top)
   door1 - (either door lockable top)
```

```
    key1 - (either key takeable top)
    table1 - (either table container top))

  (:init
    (haslocation myself empfang)
    (haslocation key1 table1)
    (haslocation door1 empfang)
    (haslocation table1 empfang)
    (locked door1)
    (fitsin key1 door1))
```

In general, the object definition in PDDL is specified as:

(:objects {<ident> - (either {<type>}*)}*)

where <ident> is an identifier for the name of an object followed by a list of types to which it belongs. In this definition either is the PDDL reserved word which indicates that some object belongs to more than one type. And, in general the initial state definition in PDDL is specified as:

(:initial {(<ident>{<object>}*)}*)

where <ident> is the name of some predicate and <object> is the name of some of the defined objects.

Most planners that accept PDDL planning specifications assume that the initial state is closed (they implement a Closed World Assumption), i.e., any fact about the initial state not explicitly indicated is assumed to be false.

2) Goal: The goal is a description of the intended world state but unlike the initial state it is usually not a complete description of the world. Any state that satisfies the literals included in the goal is acceptable as a goal state.

Remember that in our example the goal is a world in which the door is unlocked. This can be expressed in PDDL in the following way:

```
  (:goal
     (no-locked door))
```

In general the goal definition in PDDL is specified as:

(:goal {(<ident>{<object>}*)}*)

where <ident> is the name of some predicate and <object> is the name of some of the defined objects.

3) Domain: The domain includes a crucial element in planning: the *actions*. Actions are schemes whose parameters are instantiated with the objects defined for the planning problem. Each action specifies three elements:

- Action name and parameter list.
- Preconditions: a conjunction of literals that state which are the conditions that must be satisfied by the world in order to be able to execute the action.
- Effects: a conjunction of literals that describe how the world changes when the action is executed.

Two sample actions in PDDL are:

```
(:action take
   :parameters (?x - takeable ?y - container)
   :precondition
     (not(haslocation ?x myself))
     (haslocation ?x ?y)
   :effect
     (not(haslocation ?x ?y))
     (haslocation ?x myself))

(:action unlock
   :parameters (?x - lockable ?y - key)
   :precondition
     (locked ?x)
     (haslocation ?y myself)
     (fitsin ?y ?x)
   :effect
     (not(locked ?x)))
```

The first action allows the object `myself` to take an object that is takeable (`?x - takeable`), which `myself` is not holding already (`not(haslocation ?x myself)`) and that is located in some container `?y` (`(haslocation ?x ?y)`). The action has two effects: the object is relocated to `myself` (`(haslocation ?x myself)`) and hence, it no longer located where it used to be `(not(haslocation ?x ?y))`. The second action can be interpreted similarly.

3.2.2 Handling Expressive PDDL

Until now, our discussion has been restricted to the problem of planning with the STRIPS-based representation in which actions are limited to quantifier-free,

conjunctive preconditions and effects. Since this representation is severely limited, this section discusses extensions to more expressive representations aimed at complex, real-world domains.

Conditional Effects. Conditional effects are used to describe actions whose effects are context-dependent. The basic idea is simple: allow a special *when* clause in the syntax of action effects. *when* takes two arguments, an antecedent and a consequent; execution of the action will have the consequent effect just in the case that the antecedent is true immediately before execution (i.e., much like the action precondition determines if execution itself is legal). Note also that, like an action precondition, the antecedent part refers to the world before the action is executed while the consequent refers to the world after execution. It can be assumed that the consequent is a conjunction of positive or negative literals. Conditional effects are useful when combined with quantification as we will see in the example below.

Universal and Existential Quantification. PDDL allows action schemata with universal and existential quantification. In action effects, only universal quantification is allowed, but goals, preconditions, and conditional effect antecedents may have interleaved universal and existential quantifiers. Quantified formulae are compiled into the corresponding Herbrand base, universal quantification is codified using conjunction while existential is codified using disjunction. Existential quantification is forbidden in action effects because they are equivalent to disjunctive effects and imply non-determinism and hence require reasoning about uncertainty.

As shown in the following example, we can use a universally quantified conditional effects and existentially quantified preconditions to rewrite the action take introduced in Section 3.2.1 and avoid the use of a second parameter.

```
(:action take
   :parameters (?x - takeable)
   :precondition
     (not(haslocation ?x myself))
     (exists(?y - container)(haslocation ?x ?y))
   :effect
     (forall(?y - container)(when(haslocation ?x ?y)
                                 (not(haslocation ?x ?y))))
     (haslocation ?x myself))
```

Domain axioms. Axioms are logical formulae that assert relationships among propositions that hold within a situation (as opposed to action definitions, which define relationships across successive situations).

Formally, the syntax for axioms is the following:

```
(:axiom
   :vars ({<variable> - <type>}*)
   :context <assertion>*
   :implies <assertion>*)
```

where the `:vars` field behaves like a universal quantifier. All the variables that occur in the axiom must be declared here. See Section 4.3.3 for a complete example.

Action definitions are not allowed to have effects that modify predicates which occur in the `:implies` field of an axiom. The intention is that action definitions mention "primitive" predicates (like `haslocation`), and that all changes in truth value of "derived" predicates (like `accessible`) occur through axioms. Most planners do not verify this restriction syntactically but they do not take responsibility for the outcome due to the complex interactions among actions and axioms.

However, without axioms, the action definitions will have to describe changes in all predicates that might be affected by an action, which leads to a complex "domain engineering" problem.

Even though expressive PDDL can handle more complex domains, few planners will handle the entire PDDL (as we mentioned already in the beginning of this section). The reason for this is that, as expected, the more expressive the language accepted by the planner the higher the computational complexity of the problem that involves finding a plan. Hence, expressive PDDL will not be desirable for all applications; a balance of expressivity and complexity should be found.

Chapter 4

Dynamic Planning in a Dialogue System

In this chapter we explain how all the pieces we have analysed so far fit together. We will show how RACER and Blackbox can collaborate to add planning capabilities to FrOz.

We will start this chapter "talking" with FrOz in order to see clearly why it needs to do planning. In Section 4.2 we will introduce FrOz new version, *FrOz Advanced (FrOzA)*, and describe its architecture which includes the new *Dynamic Planning* module. The rest of the chapter will describe in detail the design of this module. Finally, we will close this chapter showing an interaction with FrOzA and discuss briefly the improvements.

4.1 The Problem we solve

When playing with FrOz, a situation such as the following is commonplace:

```
(1)  FrOz:   You are in a brown couch. There is a brown frog,
             a green apple and a green frog in the couch.
             The couch is in a drawing-room.
             There is a brown table in the drawing-room.
             The drawing-room has a exit leading south.
(2)  Player: look at the apple
(3)  FrOz:   The apple is green. The apple has a worm.
(4)  Player: eat it
(5)  FrOz:   You can't do this! You do not have the apple.
(6)  Player: take it
(7)  FrOz:   You have the apple.
(8)  Player: eat it
```

```
(9)  FrOz:   You can't do this! The apple has the worm.
(10) Player: take the worm
(11) FrOz:   You have the worm.
(12) Player: eat it
(13) FrOz:   You can't do this! You can not eat the worm.
(14) Player: eat the apple
(15) FrOz:   You eat the apple.
```

Almost from the beginning of this sample dialogue with FrOz the player is trying to perform a single action, namely "eat an apple". However it takes her not one but six turns in the game to succeed! Her command in (4) is rejected because she is not actually holding the apple (`(inventory-object X)` is a precondition of the action `eat(X)`). She has to first input explicitly the command "take it" in (6). But this is not enough, when she tries again in (8) FrOz fails a second time because there is a worm in the apple (`not((part-of X Y) and (disgusting Y))` is a precondition of `eat(X)`, and worms are clearly disgusting) although the player already knows about the worm and does not seem to care. She is forced to indicate explicitly that she wants to take the worm off the apple (in command (10)) in order to be able to eat "it". And it is this last "it" which puts her in trouble in line (12) (her confusion is understandable given all the previous tedious interaction!). She fixes the misunderstanding in command (14) and finally succeeds.

As we can see from the example, the player's experience can be easily hampered because the game is not able to fill the gaps in the input received, and free the player from the nuisance of specifying simple, extra actions necessary in order to meet the preconditions. A more natural interaction would be achieved if the game made an effort to overcome presuppositions that the player may make. In our setup, the presupposition made by the player is to think that all obvious preconditions of a given action hold in the current state of the world.

4.1.1 Why Planning? Why Dynamic?

We would like FrOz to avoid tedious interaction steps with the player in order to make the game experience more natural. To this aim, the game has to be able to compute the sequence of actions that should be executed in order to get from the world state where some action fails to the state in which this action can be executed (i.e., the state where all the action preconditions hold); and to execute these actions automatically, even if the player has failed to specify them in detail. The task of determining the proper sequence of actions is clearly a planning task that can be defined as follows.

Let A be an action defined in the scenario S with the set of preconditions P, and let W be a game world state. Suppose that the execution of A fails over W

because there exists some precondition in P that does not hold in W, then the planning task would be specified as follows:

1) *The initial state:* A description of the world state W (the world state where action A fails).

2) *The goal:* A description of a world W' such that all preconditions in P hold in W' (the world state where action A can be executed).

3) *The domain:* A description of the actions in the game scenario S.

Since the planning task we want to tackle can be formally and completely specified, we do not need to implement a specialised planning algorithm (as is the case for most planning approaches in dialogue systems that we discussed in Chapter 1). Instead, we can profit from an off-the-shelf planner to enhance our dialogue system capabilities. In what follows, we will discuss how a general purpose planner can be used to help FrOz.

Now, let us discuss why we call *dynamic* the planning required in this setup. In FrOz, the game world state changes while the player explores its environment and such changes are registered in the game knowledge bases. It turns out, then, that the kind of planning needed is embedded in a complex framework because the initial state and the goal need to be recalculated *dynamically* (and in a generic way) during game execution each time an action fails. This involves interfacing with the inference system RACER, which is used in FrOz to model the changing knowledge of the player.

We have argued that we can provide an off-the-shelf planner with all what it needs to accomplish its task but, is there any off-the-shelf planner that can output the plans we need within our time constraints (real time)? We will approach this question in the following section.

4.1.2 Characteristics of a Suitable Planner

Given our setup, the characteristics of a suitable planner are:

- **Short turnaround time:** Fast responses are critical for a natural interaction with the player. This is due to the fact that planning problems are defined dynamically during game execution and solutions have to be found quickly in order to allow for smooth and natural dialogues during the game.

- **Optimal plans:** We need the planner to be able to find plans that are minimal in the number of actions, i.e., plans that do not contain unnecessary actions. Optimal plans are crucial because the game should not force the player to perform actions not intended by her.

- **Expressive Specification Language:** Since we would like the planner to be able to mimic game execution it is desirable that it accepts a planning language with an expressive power equivalent to that offered by the formalism used to codify the game knowledge bases (the $\mathcal{ALCIF}$ Description Logics).

The first two requirements are mandatory. We cannot accept plans that are not optimal because they will make the game perform arbitrary, unnecessary actions on its own, a clear no go. On the other hand, FrOz is an interactive application so we cannot afford waiting for the planner more that a couple of seconds. As we have already argued in Chapter 3 there are nowadays off-the-shelf planners that excels in the computation of short, optimal plans in complex domains. Blackbox is one of them and in particular, it is an extension of SatPlan, the winner of most International Planning Competitions including last one (IPC 2006[1]). Blackbox is then an excellent candidate and it is the planer of our choice. However, we should notice that, even though we will be usually referring to Blackbox capabilities, what we will describe is a general architecture where the particular planner can be easily changed (most planners available today have a standard input and output).

The last desirable planner characteristic cannot be addressed so easily, we will discuss it in detail in Section 4.3.

4.2 Dynamic Planning: System Architecture

The *Dynamic Planning* module will be in charge of providing dynamic planning capabilities to FrOzA. We will describe now how this module can be integrated into FrOz architecture (described in Chapter 2).

In FrOzA, when an action specified by the player fails because some of its preconditions do not hold in the game world, the actions module will invoke the services provided by the Dynamic Planning module, as depicted in Figure 4.1. The Dynamic Planning module will perform two main tasks:

1. **Codify** the planning problem (initial state and goal) dynamically. And then invoke the planner with the planning problem and the scenario planning domain.

2. **De-codify** the plan found by the planner and reinsert it in the game cycle.

The detailed design of this two tasks will be addressed in Section 4.4. The planning domain specification is part of the scenario definition but can be generated automatically from the scenario action database and T-Box, once and for all, for each game scenario. The details for its generation will be presented in Section 4.3.

[1] http://zeus.ing.unibs.it/ipc-5

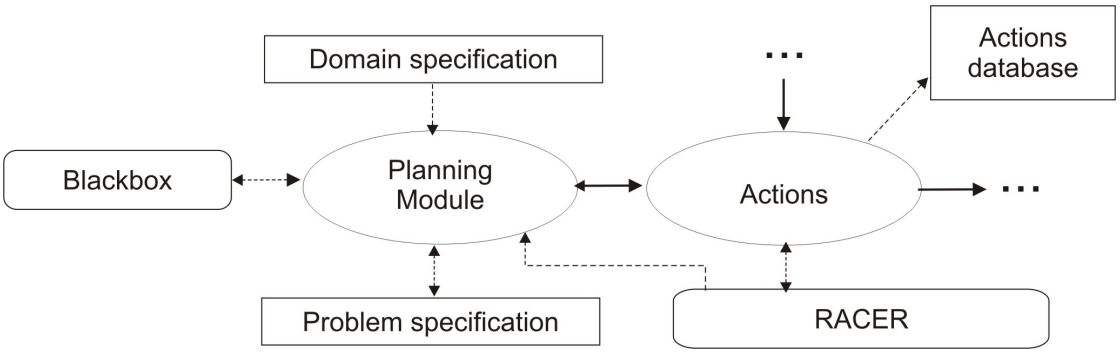

Figure 4.1: FrOzA architecture

4.3 Generation of the Planning Domain

The domain specification involves a number of important design decisions and, hence, it is difficult to generate. However, as it is independent of the state of the game at a particular moment, it can be generated offline, once and for all for each game scenario. And, more importantly, a generic translation algorithm can be defined such that it accepts arbitrary scenarios. The information required is obtained mainly from the scenario actions database but we also need to query RACER about definitions in the scenario T-Box.

4.3.1 Typing

In this section we will first analyse the pros and cons of defining types when generating planning domains. We then describe how types and predicates can be obtained from FrOz scenarios in order to include them in the generated domain.

Typed vs. Untyped Definitions: As we described in Chapter 3, PDDL offers the possibility to define types that can then be applied to predicates parameters and objects. It is usually advised to specify types as constrained as possible, because an adequate type system might result in far less actions instantiated during the search of a plan for a given goal. Moreover, types are also useful at the conceptual level, as they make the planning domain easier to understand and simpler to debug.

Of course, automatically generating proper types makes the construction of the planning domain more complex. We need to interact with the game knowledge bases in order to obtain the necessary information. Despite this, we will specify types whenever possible for the sake of efficiency during game execution.

Defining Types and Predicates: We will include in the domain all the predicates, i.e., concepts and roles, in the T-Box. In order to obtain them, we issue the queries `(concept-descendants *top*)` and `(all-roles)` to RACER. Afterwards we classify the predicates in:

- Static predicates (or types): Concepts that are *static*, i.e., they are not modified by any action effect in the action database. The domain specification will list them in the `:types` PDDL environment.

- Dynamic predicates: All other concepts (which are not *static*). The specification indicates the predicate name and its typed parameter. The parameter type corresponds to the direct subsumer of this concept in the T-Box.

- Binary predicates: Roles. They have two parameters corresponding to the domain and range individuals. Each parameter is typed with the concept that RACER informs to be the most specific one that subsumes the concepts related by the role as exemplified in Example 6 below.

According to PDDL, both dynamic and binary predicates are specified inside the `:predicates` environment.

Example 6. The role `haslocation` specified in the FairyTaleCastle relates individuals with their positions. In PDDL this role is defined as a binary predicate `(haslocation x y)`. In order to obtain the type of the parameters we issue the following queries to RACER:

- The type of `x` is the result of the query `(concept-parents(some haslocation top))`.

- The type of `y` is the result of the query `(concept-parents(some (inv haslocation) top))`.

The obtained definition for the predicate `haslocation` is `(haslocation x - top y - genericcontainer)`.

4.3.2 The Actions

The generation of suitable PDDL actions specifications is the most complex task in order to specify the domain. To address it we will first provide the main intuitions through an example and then formalise them. Consider the action `take` we discussed in Section 2.2.2.

take(patient:X)	
preconditions	instance(X takeable), instance(X accessible), not(instance(X inventory-object))
effects	*add:* instance(X inventory-object) *delete:* related(X individual-filler(X haslocation) haslocation)
player beliefs	*add:* instance(X inventory-object) *delete:* related(X individual-filler(X haslocation) haslocation)

Figure 4.2: Action entry `take` in the action database

This action would be encoded in PDDL as follows:

```
(:action take
    :parameters (?x - takeable ?y - top)
    :precondition
        (accessible ?x)
        (not(inventoryobject ?x))
        (haslocation ?x ?y)
    :effect
        (inventoryobject ?x)
        (not(haslocation ?x ?y)))
```

The first obvious difference between the two representations is that action parameters in FrOz are not typed, while Blackbox allows typing. We can type the parameters as follows. We say that a parameter `?x` belongs to the type `t` if `t` is a static concept and `instance(X t)` is a precondition for the action. In this way we avoid the instantiation of the action `take` with individuals that do not satisfy this precondition (moreover, the precondition is no longer necessary and we can eliminate it from the specification of the action). In cases when no such precondition exists then `?x` belongs to the type `top`.

The second evident discrepancy in the proposed translation is that the action has now two parameters instead of only one. The second parameter (`?y`) represents

the individual that the RACER expression `individual-filler(X haslocation)` resolves to. But, how can we be sure that Blackbox will instantiate this parameter with the appropriate individual? For this example, we just need to tell the planner that `?x` has to be related with `?y` by the functional role `haslocation` adding the precondition `(haslocation ?x ?y)` to the action. Having explained this, the required encoding can be directly obtained from the representation of the action `take` in FrOz's actions database.

The embedding of RACER expressions in action specifications is the most complex issue we have to deal with. In order to translate such actions into STRIPS-style PDDL actions, ad hoc solutions need to be found. Through the rest of this section our approach will be first to describe the general structure of an action database entry. And then we will present a formal translation into PDDL STRIPS-style (the fragment of PDDL accepted by Blackbox), providing ad hoc solutions whenever no general one is possible. Finally, we will discuss general solutions for this problems using Expressive PDDL.

FrOz Action Database

Each entry in this database corresponds to an action the player can execute in the game scenario, and specifies the action's preconditions and effects. Recall (from Chapter 2) that the effects are divided into the ones that modify the world A-Box (*effects*) and those that modify the player A-Box (*player beliefs*). The FairyTaleCastle scenario, has 26 entries in its action database. Each entry has the following structure:

`<action-name> [({<sem-role><param>}+)]`	
preconditions	`<assertion>*`
effects	add: `<assertion>*` delete: `<assertion>*`
player beliefs	add: `<assertion>*` delete: `<assertion>*`

Where:

- `<action-name>` is an identifier for the entry, i.e., the action name.
- `<param>` specify the parameter name and `<sem-role>` its semantic role (patient, target or instrument).
- `<assertion>` is an expression of one of the following forms:
  - `instance(<indiv> <concept>)` that when included in:

    - The preconditions: Holds if `<indiv>` belongs to `<concept>` in the world A-Box.
    - The added (deleted) effects: Asserts (retracts) in the world A-Box that an `<indiv>` is an instance of `<concept>`.
- `related(<indiv> <indiv> <role>)` that when included in:
    - The preconditions: Holds if `<indiv>` is related to `<indiv>` by `<role>` in the world A-Box.
    - The added (deleted) effects: Asserts (retracts) in the world A-Box that `<indiv>` is related to `<indiv>` by `<role>`.
- `not(<assertion>)` can only appear in the preconditions. It holds in the case that `<assertion>` does not hold in the world A-Box.
- `describe(<individual>|<concept>)` can only appear as an added player belief. It indicates the Content Determination module to add information about one or more individuals to the player A-Box.

The assertions in the player beliefs are interpreted in an analogous way to those in the effects but over the player A-Box instead of the world A-Box.

`<indiv>` is either a parameter name `<param>`, an individual name (that belongs to the corresponding A-Box), or a RACER expression that returns an individual name. A `<concept>` is the name of some primitive concept (that belongs to the game T-Box) if the `<assertion>` is in the effects. Otherwise, if the `<assertion>` is in the preconditions then `<concept>` can be a primitive concept, a derived concept or an arbitrary complex RACER concept expression. And a `<role>` is either the name of some role (that belongs to the game T-Box) or a RACER role expression.

A RACER expression is embedded in the action specification when the action cannot be specified completely in advance as it depends on the current state of the game. This is the case of most interesting actions such as the `take` (see Figure 4.2). This entry includes the RACER expression `individual-filler(X has-location)` that should be resolved to an individual in the world.

The database also contains actions that are useful in the context of the game but that do not change the state of the world (like `look` and `inventory`), they only update the player beliefs (this kind of action was mentioned in Section 2.2.2, during the description of the Content Determination Module). Because these actions do not change the game world they are never required in optimal plans and we do not need to codify them.

General schema for the translation

In the planning domain, the effects of the actions will correspond to the effects over the world and not to the effects over the player beliefs. This is due to the fact

that the player effects can include information that updates her beliefs and we do not want the planner to discover new information for the player. Then, during the translation we will work over the following reduced schema:

<action-name> [(<parameter>+)]	
preconditions	<assertion>*
effects	add: <assertion>*
	delete: <assertion>*

This general schema can be roughly translated into the following PDDL schema:

```
(:action <action-name>
    [:parameters ({?<param> - <type>}+)]
    :precondition (<assertionpddl>)*
    :effect
       (<assertionpddl>)*          /* For each added assertion */
       (not(<assertionpddl>))*)    /* For each deleted assertion */
```

Where:

- <param> is a parameter name and <type> is a static concept such that instance(<param> <type>) is a precondition of the action. If no such precondition is defined in the action entry then <type> is top. If a type exists, then it is unique because the scenario taxonomy is properly defined.

- <assertionpddl> is specified according to the corresponding <assertion> in the entry specification. Those <assertion> that involve a static concept are ignored as they have already been considered when defining the parameter types. If <assertion> is an expression of the following form:

  - instance(<indiv> <concept>), then it is translated into the PDDL formula (<conceptpddl> <indivpddl>).
  - related(<indiv> <indiv> <role>), then it is translated into the PDDL formula (<rolepddl> <indivpddl> <indivpddl>).
  - not(<assertion>), then it is recursively translated into the PDDL formula (not(<assertionpddl>)).

  Where <indivpddl> is:

  - If <indiv> is a parameter name or an individual name in the A-Box then <indivpddl> is just <indiv>.

– If `<indiv>` is a RACER expression that returns and individual in the A-Box, namely `(individual-filler x <role>)`, then the translation will depend on the PDDL subset we are using.

Where `<conceptpddl>` is:

– If `<concept>` is the name of a primitive concept in the T-Box then `<conceptpddl>` is just `<concept>`.

– If `<concept>` is a derived concept or an arbitrary RACER expression then the translation will depend on the PDDL subset we are using.

Where `<rolepddl>` is:

– If `<role>` is a primitive role name in the T-Box then `<rolepddl>` is just `<role>`.

– If `<role>` is a RACER expression of the form `(inv <role>)` then the assertion `related(x1 x2 (inv <role>))` is translated into the PDDL assertion `(<role> x2 x1)`.

– If `<role>` is a derived role, i.e. the inverse of a primitive role `r` then the assertion `related(x1 x2 (<role>))` is translated into the PDDL assertion `(r x2 x1)`.

Now we will explain how to codify RACER expressions embedded in actions in STRIPS-style PDDL and show why the expressivity of this language cannot capture completely the planning task. In Section 4.3.3 we will discuss a complete codification using the extended PDDL subset.

Codifying FrOz Actions in STRIPS-style PDDL

In the previous schema there are two issues that were left unspecified, namely how to codify in PDDL a RACER expression `E` that either represents a generic individual or a derived concept.

Representing dynamic individuals: Our approach is to add to the action a new parameter `?y` that represents the individual. Also, we replace each occurrence of `E` in the action definition by `?y`. Moreover, we add a precondition such that the action will only be executed if the parameter is instantiated with the individual `E` would resolve to. If `E` is of the form:

- `(individual-filler x <role>)` we add the precondition `<role> ?x ?y)`.
- `(concept-instances <concept>)` we add the precondition `<concept> ?y)`.

Even though, in general, RACER expressions that include this command can return more than one individual, they return only one in the framework of the game.

Represent dynamic concepts: `E` can be the name of a derived concept defined in the T-Box or can be an arbitrarily complex concept expression (these two are equivalent, as we can transform one case into the other). General RACER concept expressions cannot be codified in PDDL without quantification, hence our approach is to replace them by derived concepts added to the T-Box. This approach is robust because we can deal with arbitrary T-Boxes, however it is not complete as we can see in the following example.

Suppose that action A has the effect a, action B has the precondition b and effect c, and the action we want to take has the precondition c; and further that b is different from a but can be proved from a using a T-Box axiom. In this case, the plan $[A, B]$ should be found but the present system would however not be able to come up with this plan because it would not recognise that A achieves B's preconditions.

We work around this problem performing a syntactic analysis of the T-Box definitions and adding to the actions effects also those predicates that are affected indirectly. This approach needs further analysis but it will not solve the problem completely as it cannot handle recursive T-Box definitions.

In order to codify recursive definitions such as `accessible` in the FairyTale-Castle scenario we add two extra actions to the planning domain. One of this actions asserts the predicate defined and the other one retracts it. For the case of `accessible` the actions are defined as follows:

```
(:action assert-accessible
   :parameters (?x - container ?z - top)
   :precondition
     (accessible ?x)
     (open ?x)
     (has-location ?z ?x)
   :effect (accessible ?z))
(:action retract-accessible
   :parameters (?x - container ?z - top)
   :precondition
     (not(accessible ?x))
     (has-location ?z ?x)
     (accessible ?z)
     (not(here ?z))
   :effect (not(accessible ?z)))
```

This translation is ad-hoc and hand tailored for the FairyTaleCastle scenario. However, we believe that the idea is general enough to be adapted to other scenarios that define recursive predicates.

The actual encoding needed for Blackbox is more complicated because Blackbox current version does not handle the Closed World Assumption. Hence, we have to deal with it ourselves explicitly adding a predicate `no-predicate` for each `predicate` in the planning domain, as well as the pertinent effects to each action.

4.3.3 Using Expressive PDDL

Using the additional requirements that expressive PDDL has to offer we can completely specify the planning task we are dealing with. In particular we can add domain axioms to the domain definition to capture T-Box axioms.

Let us exemplify this approach using the concept `accessible` from the FairyTaleCastle scenario. `accessible` was introduced in Chapter 2 and we repeated its definition here.

$$\begin{aligned}\texttt{accessible} \equiv\ & \texttt{here} \sqcup \\ & \exists\texttt{has-location.here} \sqcup \\ & \exists\texttt{has-location.(accessible} \sqcap \texttt{open)} \sqcup \\ & \exists\texttt{part-of.accessible}\end{aligned}$$

Remember that this axiom means that the location where the player is currently standing is accessible to him, as well as the individuals that are in the same location. If such individual is some kind of container and it is open then its contents are also accessible; and if it has parts, its parts are accessible as well.

We can translate this description logic definition into two PDDL axioms, the following one and a second to represent the other direction of the implication.

```
(:axiom
   :vars (?x - top)
   :context (or (here ?x)
                (exists(?y - container)(and(haslocation ?x ?y)
                                           (here ?y)))
                (exists(?y - container)(and(haslocation ?x ?y)
                                           (and(accessible ?y)
                                               (open ?y))))
                (exists(?y - top)(and(partof ?x ?y)
                                     (accessible ?y))))
   :implies (accessible ?y))
```

From this example a general translation transforming DL definitions into PDDL axioms can be formalised. However, the semantics of PDDL axioms is not fully specified and it may vary from planner to planner. This approach requires further analysis, in particular, possible interactions among axioms should be tested.

Such a codification can be handled by general purpose planners available today, such as PaDoK. This planner accepts a very expressive PDDL language including quantification, conditional effects and domain axioms. In [8] we analyse its performance when planning over FrOz game setup and we conclude that PaDoK current performance is far below our expectations. It cannot find optimal plans and its average turnaround time is not good enough for our setup. There is today no planner that can deal with this expressive language and, at the same time, is able to find optimal plans, and has appropriate performance for our framework. Hence, the compromise solution found for this thesis is to use Blackbox, a planner that accepts a less expressive language but it provide the preformance requirements for our task

4.4 Dynamic Planning: Detailed Design

In this section we analyse in detail the tasks of the Dynamic Planning module. We will begin by describing how this module generates the planning problem that, together with the planning domain, conform Blackbox input. In Section 4.4.2 we analyse how FrOzA will use this plans.

4.4.1 Codifying the Problem

The problem specification clearly depends on the state of the game and on the input of the player at a particular moment. Hence, this specification should be automatically generated on-line during the execution of the game each time FrOz fails because the preconditions of some action entered by the player do not hold in the game world. The necessary information will be obtained from the player input as well as from the state of the game knowledge bases.

Remember that a problem specification consists of two parts. The first one, is the definition of the *initial state* including the objects involved in the problem. The other part is the *goal* of the planning problem.

In order to define the objects, types and initial state of a planning problem we need to query RACER about the objects and their properties in the knowledge base. However, FrOz has two A-Boxes (the world's and the player's) and we need to decide which one can give us the information we need. It seems natural to choose the player's because we do not want Blackbox to return plans including actions

that the player is not aware she can perform. However, we should remember that this knowledge base may contain information that is inconsistent with respect to the current state of the game. So it is possible that Blackbox actually returns a plan that cannot be executed over the game world. To clarify this point, let us return to the example where the player tries to unlock a door with a key. Suppose that the key was originally lying on the table, but without the player knowing, it is now in possession of a thief. As a consequence, the key is on the table in the player's knowledge base, but in the game world the thief has it. With the added planning capabilities, FrOzA would decide to take the key from the table and unlock the door with it. But this sequence of actions fails because the key is no longer accessible. Hence, after finding a plan it is still necessary to check whether it can actually be executed over the game world before applying its effects.

On the contrary, if instead of finding a plan according to the player's beliefs we would have planned using the world knowledge, FrOzA would have automatically taken the key from the thief (for example, by using the steal action) and opened the door for the player, while the player is not even aware where the key actually was. This is clearly inappropriate because we only want FrOzA to take actions for the player if we can be sure that these actions agree with the player intentions.

We are ready now to provide the details of the generation of a planning problem for a given state of the game.

1) Defining the initial state:

As we described in Chapter 3, in PDDL the objects with their corresponding types are specified in the following format:

```
(:objects
   {<object> - (either <concept>+)}*)
```

In order to find out which are the objects in the player knowledge base we send to RACER the query `all-individuals(PABox)` where `PABox` is the name of the player ABox. This query will return all the individuals that are already known by the player (for example `[myself, apple1, couch1]`). After obtaining the objects in this way we need to obtain their types. So we query RACER with `individual-types(<object> PABox)` for each `<object>` obtained in the previous query. RACER will return all the concepts each `<object>` belongs to. During the planning domain generation we classified concepts in *static* and *dynamic* and we will use this information now to record as `<concept>` all the static concepts each `<object>` belongs to. Obtaining, for example, the following:

```
(:objects
  apple1 - (either apple takeable object top edible)
```

```
  couch1 - (either couch seating top container)
  myself - (either player container top)
  worm1 - (either worm disgusting top))
```

As described in Chapter 3, the initial state is specified in the following format:

```
(:init
   {(<concept> <object>)|(<role> <object1> <object2>)}*)
```

Here we include literals that correspond to the player ABox assertions of the dynamic concepts and roles at the moment of the player input. I.e., we will record here concepts that are not asserted as types because they are dynamic. We also query RACER to obtain the role assertions issuing the query `all-role-assertions(PABox)`. Moreover, we need to *close* the initial state definition under negation in case the planning domain includes actions with negated preconditions. I.e., we need to explicitly include literals for the negated predicates such as `no-haslocation` in order to specify all the objects that are not related by the predicate `haslocation`. This is due to the fact that Blackbox current version does not implement a Closed World Assumption with respect to the initial state. A fragment of the initial state of the game interaction in Section 4.1 would be:

```
(:init
    (haslocation myself couch1)
    (haslocation apple1 couch1)
    (partof worm1 apple1)
    (no-haslocation apple1 myself)
    (no-haslocation myself myself)
    (no-haslocation worm1 myself)
    (no-partof worm1 myself)
    (no-partof apple1 myself)
    ...)
```

2) Defining the goal:

If we want the player to be able to execute an action she was not able to execute before, we need the preconditions of such an action to hold. Then, the goals of our planning problem will be the preconditions of the action that the player wants to execute. This information can be obtained from the actions module output. The action module output is a list of readings corresponding to the different interpretations of the received command (see Chapter 2 for further details). Each interpretation is a sequence of actions. The output of the Action module shows

which of these actions can be executed, which of these actions fail (because its preconditions are not satisfied), and which of these actions were not tested.

The Dynamic Planning module is invoked with this list of readings. If the list contains more than one reading (notice that all of them should have failed otherwise the dynamic planning module would not have been invoked) then FrOzA will pass the control to the Content Determination module who will inform the player about the ambiguity. Otherwise, the input received by the Dynamic Planning module will be only one reading with exactly one failed action, but may contain arbitrary successful and untested actions (as soon as the first failing action is found the rest of the sequence of actions is not evaluated any more). The general format of the input is as follows:

```
[reading(executable: <executable-actions>
         failed:[conjunct(action:<action-name>
                          failPre:<failed-precond>
                          succPre:<successful-precond>
                          untestedPre:<untested-precond>
                          uk:<updates-on-players-beliefs>)]
         untested: <untested-actions>)]
```

In FrOz this structure is passed on to the Content Determination module and an error is reported informing the precondition that failed. In FrOzA, however, Blackbox will be invoked with a plan goal containing the union of the lists of literals `<failed-precond>`, `<successful-precond>` and `<untested-precond>`. We need to include in the goal all the preconditions and not only those that failed because otherwise it could be the case that the actions that achieve the failed preconditions retract the successful or untested ones. From this set we eliminate those literals that refer to static concepts because no action will modify them.

The format of these preconditions is the one described in Section 4.3.2. Hence, also here we face the problem of embedded RACER expressions. Expressions that represents an individual can be resolved by asking them to RACER. Expressions that represents a concept or role can be replaced with a derived concept or role whose definition is added to the planning domain. However, this has an impact on the generation component. The lexicon and grammar need to be extended in order to consider the added concepts and roles.

In order to finish the definition of the planning problem in our running example we specify the goal according to the previous procedure, and obtain:

```
(:goal
     (inventory-object apple1)
     (no-partof worm1 apple1))
```

We are now ready to invoke Blackbox using the problem and the domain definition generated. In the next section we will discuss how to use the plans found by Blackbox.

4.4.2 De-codifying and Using Plans

Given the domain and problem specifications, Blackbox is able to find the sequence of actions required in order to get from the state where the action fails to the state in which this action can be executed according to the player's beliefs, in case such a sequence exists. Also, Blackbox will be instructed to find plans with a maximum length of 2 or 3 actions because we do not want the planner to solve the game for the player, even if the player already has all the information to solve it. The appropriate length of such plans will be analysed in Chapter 5.

If Blackbox does not find a plan, FrOzA will silently pass the control to the Content Determination module as FrOz would do. Otherwise, if a plan is found, FrOzA will try to execute it and then to execute the action that originally failed, and also the successful and untested actions included in the reading. Let us explain how FrOzA manage to do this using an example.

Given the goal in the example of Section 4.4.1, the plan returned by Blackbox would include two actions:

```
-------------------------------------------------
Begin plan
1 (take apple1 couch1)
2 (takeoff worm1 apple1)
End plan
-------------------------------------------------
```

The actions 'take the apple' and 'take the worm' can be performed, in that order, from the current state in the game.

Now, let us discuss how FrOzA is able to execute these actions on its own. The fundamental idea is to take advantage of FrOz ability to handle conjunctions of actions (see Section 2 for details). We will reinsert the plan as if the input received by the player had specified in detail all the actions included in the plan, plus the actions she actually provided. Then, the input reinserted in the actions module for our example will be:

```
[[take(patient:apple1),
  takeoff(patient:worm1),
  eat(patient:apple1)]]
```

Notice that we need to determine the semantic role of the actions parameters and discard those parameters that were introduced for the purpose of the translation to PDDL. This is done encoding semantic roles information in the planning domain when it is generated according to the name and number of parameters each action has.

From here on, the processing cycle in FrOzA continue as in FrOz. Inserting this sequence of actions in the normal cycle allows for the verification of the preconditions over the game world. If the plan is successful, the changes on the world are informed to the player. As we mentioned before, it is possible that the plan fails because the game knowledge base can be inconsistent with respect to the players knowledge base. Then it is possible that the plan found using the player beliefs is not executable in the world. Also in this case the normal output of FrOz is appropriate because it will inform about the preconditions that cause the failure and the player will found out that her beliefs were mistaken and will update them.

Let us return to our simple example. With the plan found by Blackbox FrOzA is able to execute the sequence of actions on its own and the 15-turns interaction analysed in the beginning of this chapter takes only 5 turns in FrOzA:

```
(1)  FrOzA:  You are in a brown couch. There is a brown frog,
             a green apple and a green frog in the couch.
             The couch is in a drawing-room.
             There is a brown table in the drawing-room.
             The drawing-room has a exit leading south.
(2)  Player: look at the apple
(3)  FrOzA:  The apple is green. The apple has a worm.
(4)  Player: eat it
(5)  FrOzA:  You have the apple.
             You have the worm.
             You eat the apple.
```

Even if the output does not sound completely natural (no changes have been made to the generation module), it is correct and much more effective than the original interaction with FrOz. The output can be improved by making the Generation component aware that certain of the actions just executed involved planning. A more suitable answer would be:

```
FrOzA: You eat the apple
       [taking it and taking the worm from it first].
```

Chapter 5

Evaluation: Cases of Study

5.1 Introduction

System evaluation today is much of an art and craft as it is a science with stablished standards and procedures of a good engineering practice. In particular, little is still known about evaluation of dialogue components and integrated dialogue systems. In [10], Bernsen introduce the following classification of types of evaluation:

Performance evaluation: Measurement of the performance of the system and its components in terms of a set of quantitative paramenters.

Diagnostics evaluation: Detection and diagnosis of design and implementation errors.

Adequacy evaluation: How well the system and its components fit the purposes and meet actual user needs and expectatives.

The first two kinds of evaluations are to be performed by the system expert with predefined test-cases. While the third one need to involve target users.

This chapter presents an evaluation of FrOzA prototype. Given that we have not developed FrOzA from scrath but from the working prototype FrOz, we will focus this evaluation in testing the added dynamic planning capabilities. Due to time restrictions, the evaluation of the system will be necessarily limited. In particular, given the current development state of FrOzA, adequacy testing for users is not yet practicable. Much more complex scenarios (and crucially, making parsing more robust) should be developed in order to be able to obtain representative results.

For our purposes then, we will organize our evaluation in two stages:

- First of all, the planner will be tested separately in order to evaluate its performance and its suitability for the task.

- Second, FrOzA will be tested on different specific, particularly interesting, test-cases that pose challenges to the Dynamic Planning Module. Such test-cases show FrOzA's added capabilities in comparison to FrOz. This stage includes diagnosis evaluation for the integrated system and performance of the enhanced system compared with the previous one.

5.2 Component Evaluation

5.2.1 Blackbox Performance

Blackbox is a state-of-the-art planner that excells in the computation of short, optimal plans in complex domains. It is an extension of SatPlan, the winner of most International Planning Competitions including last one (IPC 2006[1]). As a representative example of Blackbox performance, Kautz describes in [36] describes a planning problem where Blackbox requires only six minutes to find a 105-action logistics plan in a world with 10^{16} possible states.

This is fairly impressive, but we should verify that the performance of Blackbox is similar in the specific kind of planning domains we are interested within the framework of FrOzA.

Having said this there is not much left to say about Blackbox performance. Now, let us see if its perfomance is also suitable for our setup. Blackbox developers point out that Blackbox's approach is not suitable if the folllowing characteristics are present in the problem:

- The domain is too large for propositional planning approaches (where 'too large' means thousands or ten of thousands actions and objects);
- Long sequential plans are needed;
- Optimal planning is not necessary.

Our task is exacly the complement of these three characteristics. To begin with, the domains FrOz has to deal with are far smaller than the benchmarks Blackbox has been tested with. For the world domains provided with FrOz (around 20 actions schemes and 30 individuals that instantiate around 100 actions), it only takes the planner a couple of milliseconds to find a suitable plan or to answer that there is none.

In order to check the scalability of our approach, we tested Blackbox with up to 6000 instantiated actions, but even for this huge scenario (in comparison with the usual scenarios provided with FrOz), Blackbox still takes less than a second to

[1] http://zeus.ing.unibs.it/ipc-5

return a plan or to say that there is none. It is clear that the size of our problems domains is not an issue for Blackbox performance.

Moreover, we need short plans because we do not want the planner to solve the game for the player. In this respect, we can easily instruct Blackbox to look only for plans with up to say two actions. Finally, in our setup optimal plans (i.e., plans containing no superfluous actions) are mandatory because otherwise we risk executing actions the player would not have thought of performing.

Given these facts it is clear that our choice of Blackbox as the planner for the task is well motivated.

5.3 Integrated Evaluation

This section present three test-cases where FrOzA is tested on different specific, particularly interesting, situations that pose challenges to the Dynamic Planning Module. Such test-cases show FrOzA's added capabilities in comparison to FrOz.

5.3.1 Test-case 1

The main goal of this test case is to compare FrOz and FrOzA performance over the same interaction. This test case was developed over the FairyTaleScenario for FrOz. During this simple sample session our aim in the game is to kill the green frog and to eat the red apple.

The interaction with FrOz evolves as follows:

```
(1)    FrOz: You are in a brown couch.
             There is a brown frog, a green apple
             and a green frog in the couch.
             The couch is in a drawing-room.
             There is a brown table in the drawing-room.
             The drawing-room has a exit leading south.
(2)  Player: look at the green frog
(3)    FrOz: The green ugly frog is alive.
             The green frog has a silver crown and a sword.
(4)  Player: kill it with the sword
(5)    FrOz: You can't do this!
             You do not have the sword.
(6)  Player: take the sword
(7)    FrOz: You have the sword.
(8)  Player: kill the frog with the sword
(9)    FrOz: There is more than one frog.
```

```
(10) Player: kill the green frog with the sword
(11)   FrOz: The green frog is dead.
(12) Player: stand up
(13)   FrOz: You are in the drawing-room.
(14) Player: look
(15)   FrOz: There is the couch and the table in the drawing-room.
             The drawing-room has a exit leading south.
(16) Player: look at the table
(17)   FrOz: The table is brown.
             The table contains a red apple.
(18) Player: look at the apple on the table
(19)   FrOz: The apple is red.
             The red apple has a worm.
(20) Player: throw the worm to the couch
(21)   FrOz: You can't do this!
             You do not have the worm.
(22) Player: take the worm
(23)   FrOz: You have the worm.
(24) Player: throw it to the couch
             The worm is in the couch.
             The worm is dead.
(25) Player: eat the red apple
(26)   FrOz: You can't do this!
             You do not have the red apple.
(27) Player: take the apple
(28)   FrOz: You have the red apple.
(29) Player: eat it
             You eat the red apple.
```

We need 29 interactions with FrOz in order to achieve our aims. In the turns (5), (21) and (26) the game fails to perform the action asked by the user because some preconditions for the action do not hold in the current state of the world.

If we play FrOzA following the same inputs as in the previous example, the interaction is identical up to the 4th turn, then it evolves as follows:

```
(4)  Player: kill it with the sword
(5)   FrOzA: You have the sword.
             The green frog is dead.
(6)  Player: stand up
(7)   FrOzA: You are in the drawing-room.
(8)  Player: look
```

```
(9)   FrOzA: There is the couch and the table in the drawing-room.
             The drawing-room has a exit leading south.
(10) Player: look at the table
(11)  FrOzA: The table is brown.
             The table contains a red apple.
(12) Player: look at the apple on the table
(13)  FrOzA: The apple is red.
             The red apple has a worm.
(14) Player: throw the worm to the couch
(15)  FrOzA: You have the worm.
             The worm is in the couch.
             The worm is dead.
(16) Player: eat the red apple
(17)  FrOzA: You have the red apple.
             You eat the red apple.
```

With FrOzA only 17 turns are necessary in order to achieve the same aims. In this example, FrOzA is 40% more effective than FrOz, 40% less interactions are needed in order to reach the same world state. Moreover, and ever more important, the interaction with FrOzA is much more natural, with the system showing a clear collaborative behavior. This is because FrOzA is using its dynamic planning capabilities to fill in the gaps in the user input. Concretely, FrOzA uses planning for the interactions (5), (15) and (17). Let us analyse each of them in detail, to understand exactly what was the job of the Dynamic Planning Module.

Turn 5

During this turn, the action `kill(patient:priesemut instr:sword2)` fails because its precondition `(inventory-object sword2)` does not hold in the current game state. The dyanmic module of FrOzA will generate the following planning problem as explained in Chapter 4:

```
(define (problem froza)
(:domain fairytalecastle)
(:objects
  empfang - (either drawing-room room generic-container top)
  sword2 - (either sword takeable object top weapon)
  apple1 - (either apple takeable object top edible ...)
  crown2 - (either crown takeable object top silver colour ...)
  priesemut2 - (either frog takeable top easy-to-kill brown ...)
  drawing2treasure-exit - (either south-exit exit open-closed ...)
```

```
  couch1 - (either couch seating open-closed generic-container ...)
  table1 - (either table open-container  generic-container ...)
  myself - (either player generic-container top open-closed)
  priesemut - (either frog takeable easy-to-kill green ...)
)
(:init
  (accessible sword2)
  (accessible apple1)
  (accessible crown2)
  (accessible priesemut2)
  (open couch1)
  (here couch1)
  (accessible couch1)
  (open table1)
  (open myself)
  (accessible myself)
  (accessible priesemut)
  (alive priesemut)
  (has-exit empfang drawing2treasure-exit)
  (has-location apple1 couch1)
  (has-location priesemut2 couch1)
  (has-location couch1 empfang)
  (has-location table1 empfang)
  (has-location myself couch1)
  (has-detail priesemut sword2)
  (has-detail priesemut crown2)
  (has-location priesemut couch1)
  (no-haslocation apple1 table1)
  ...  [here goes the complete list of negated atoms
        required by the Close World Assumption]
)
(:goal
  (inventory-object sword2))
)
```

With this planning problem and the domain specification of the FairyTaleCastle (which was created as explained in Chapter 4) Blackbox output is the following:

```
blackbox version 42
command line:  blackbox
               -o FairyTaleCastle.PDDL
```

```
               -f problem1152384653.PDDL
               -g plan1152384653

Problem name: froza
Facts loaded.
time: 1, 591 facts and 26 exclusive pairs.
Goals first reachable in 1 steps.
1762 nodes created.

####################################################
goals at time 2:
  inventory-object_sword2

----------------------------------------------------
Begin plan
1 (take=patient sword2 priesemut)
End plan
----------------------------------------------------

1 total actions in plan

####################################################
Total elapsed time:   0.04 seconds
Time in milliseconds: 36

####################################################
```

This plan is parsed by the dynamic planning module and the conjunction of actions `[[take(patient:sword2) kill(patient:priesemut instr:sword2)]]` is reinserted in the game cycle. Obtaining in this way the output shown in turn (5): `You have the sword. The green frog is dead.`

Turn 15

During this turn, the action `throw(patient:worm1 target:couch1)` fails because its precondition `(inventory-object worm1)` does not hold in the current game state. Then the dynamic module of FrOzA will generate the corresponding planning problem in a way analogous to the previous one and the plan found by Blackbox will be:

```
----------------------------------------------------
```

```
Begin plan
1 (take=patient worm1 apple2)
End plan
--------------------------------------------------
Total elapsed time in milliseconds: 69
```

Then the conjunction of actions `[[take(patient:worm1) throw(patient:worm1 target: couch1)]]` is output by the dyanmic planning module and reinserted in the game cycle.

Turn 17

During this turn, the action `eat(patient:apple2)` fails because its precondition `(inventory- object apple2)` does not hold in the current game state. Then the dynamic module of FrOzA will generate the corresponding planning problem in a way analogous to the previous one, and the plan found by Blackbox will be:

```
--------------------------------------------------
Begin plan
1 (take=patient apple2 table1)
End plan
--------------------------------------------------
Total elapsed time in milliseconds: 39
```

Then the conjunction of actions `[[take(patient:apple2) eat(patient:apple2)]]` is output by the dyanmic planning module and reinserted in the game cycle.

5.3.2 Test-case 2

The main goal of this test case is to show that FrOzA's planning capabilities are dynamic. As we already mentioned, we say that planning is 'dynamic' in this setup because the planning problem chages according to the player beliefs. The more the player knows about the game world, the more the planning capabilities of FrOzA are used. This is illustrated in the following interaction.

```
(1)   FrOzA: You are in a brown couch.
             There is a brown frog, a green apple
             and a green frog in the couch.
             The couch is in a drawing-room.
             There is a brown table in the drawing-room.
             The drawing-room has a exit leading south.
(2)  Player: inventory
```

```
(3)   FrOzA: You have a silver key.
(4)  Player: stand up
(5)   FrOzA: You are in the drawing-room.
(6)  Player: go south
(7)   FrOzA: You are in a treasury.
             There is a chest and a dragon in the treasury.
             The treasury has a exit leading north.
(8)  Player: look at the chest
(9)   FrOzA: The chest (which is locked) is wooden.
(10) Player: open the chest
(11)  FrOzA: You can't do this!
             The chest is not unlocked.
(12) Player: unlock the chest with the key and open it
(13)  FrOzA: The chest is unlocked.
             The silver key fits into the chest.
             The chest is open.
             The chest contains a golden crown and a silver sword.
(14) Player: lock it with the key
(15)  FrOzA: The chest is closed.
             The chest is locked.
(16) Player: look at the chest
(17)  FrOzA: The chest (which is locked) is wooden.
(18) Player: open the chest
(19)  FrOzA: The chest is unlocked.
             The key fits into the chest.
             The chest is open.
             The chest contains the crown and the sword.
```

Notice that the player input "open the chest" in (10) fails but succeeds in (18), even if the chest state reported by FrOzA is exactly the same: "The chest (which is locked) is wooden.". This change in FrOzA behaviour is due to the fact that the player knowledge in (18) is different than her knowledge in (10). The player learns in (13) that "The silver key fits into the chest.", hence in (19) FrOzA is able to infer that, when the player says "open the chest", she is willing to unlock the chest with the key first.

Now that we have an intuitive idea of what is going on let us analyze FrOzA's behavior in detail for this test-case.

Turn 11

In this turn FrOzA tries to find a plan in order to be able to open the chest. It then constructs the following planning problem:

```
(define (problem froza)
(:domain fairytalecastle)
(:objects
  empfang - (either drawing-room room generic-container top)
  schatzkammer - (either treasury room generic-container top)
  apple1 - (either apple takeable object top edible green ...)
  key1 - (either key takeable object top silver ...)
  priesemut - (either frog takeable object top easy-to-kill green ...)
  priesemut2 - (either frog takeable object top easy-to-kill brown ...)
  grisu - (either dragon object top not-so-easy-to-kill ...)
  chest1 - (either chest open-closed locked-unlocked generic-container ...)
  treasure2drawing-exit - (either north-exit exit ...)
  drawing2treasure-exit - (either south-exit exit ...)
  couch1 - (either couch seating open-closed generic-container ...)
  table1 - (either table open-container generic-container ...)
  myself - (either player generic-container ...)
)
(:init
  (here schatzkammer)
  (accessible schatzkammer)
  (inventory-object key1)
  (accessible key1)
  (accessible grisu)
  (accessible chest1)
  (open couch1)
  (open table1)
  (open myself)
  (accessible myself)
  (has-exit empfang drawing2treasure-exit)
  (has-exit schatzkammer treasure2drawing-exit)
  (has-location apple1 couch1)
  (has-location key1 myself)
  (has-location priesemut couch1)
  (has-location priesemut2 couch1)
  (has-location grisu schatzkammer)
  (has-location chest1 schatzkammer)
  (has-location couch1 empfang)
  (has-location table1 empfang)
  (has-location myself schatzkammer)
)
```

```
(:goal
  (unlocked chest1))
)
```

With this planning problem and the domain specification of the FairyTaleCastle Blackbox output is the following:

```
-----------------------------------------------------
NO SOLUTION
Problem not solvable: can't even reach non-mutex goals
-----------------------------------------------------
Total elapsed time in milliseconds: 297
```

Turn 19

In this turn the assertion `(fits-in key1 chest1)` is included in the planning problem initial state and then the planner is able to find the following plan:

```
-----------------------------------------------------
Begin plan
1 (unlock=patient=instr chest1 key1)
End plan
-----------------------------------------------------
Total elapsed time in milliseconds: 94
```

5.3.3 Test-case 3

The main goal of this test case is to show FrOzA behaviour when the player knowledge is inconsistent with the game knowledge. Moreover, in this test-case we show that FrOzA added planning capabilities can be used over a different game scenario. For this example we play on an extension of the game scenario called Space Station provided with FrOz current distribution. Let us analyse the following game fragment:

```
(1)  FrOzA: You are in a cargo-room.
            There is a silver toolbox and a brown crate in the cargo-room.
            The cargo-room has a exit leading north.
(2) Player: open the toolbox
(3)  FrOzA: The toolbox is open.
            The toolbox contains a silver knife, a white tube of foodpaste,
            a red hacksaw and a white id-card.
(4) Player: eat the foodpaste
(5)  FrOzA: ...
```

What do you think that FrOzA will answer? Let us see, when FrOzA tries to execute the action eat over the world it fails because some of its preconditions does not hold. Then, as usual, FrOzA invoques the planning module which generates the following planning problem:

```
(define (problem froza)
(:domain spacestation)
(:objects
  scooter-cargo - (either cargo-room room generic-container top)
  tube1 - (either foodpaste edible object top takeable tube ...)
  saw1 - (either hacksaw saw takeable object top red colour ...)
  id1 - (either id-card takeable object top white colour ...)
  knife1 - (either knife weapon top takeable object silver ...)
  cargo2bridge-exit - (either fore-exit exit open-closed top ...)
  crate1 - (either crate open-container generic-container brown ...)
  myself - (either player generic-container top open-closed)
  toolbox1 - (either toolbox open-closed generic-container ...)
)
(:init
  (here scooter-cargo)
  (accessible scooter-cargo)
  (accessible tube1)
  (accessible saw1)
  (accessible id1)
  (accessible knife1)
  (open crate1)
  (open myself)
  (accessible myself)
  (accessible toolbox1)
  (open toolbox1)
  (has-exit scooter-cargo cargo2bridge-exit)
  (has-location tube1 toolbox1)
  (has-location saw1 toolbox1)
  (has-location id1 toolbox1)
  (has-location knife1 toolbox1)
  (has-location crate1 scooter-cargo)
  (has-location myself scooter-cargo)
  (has-location toolbox1 scooter-cargo)
  (no-haslocation tube1 crate1)
  ...
  (no-empty tube1)
```

```
  ...
)
(:goal
  (inventory-object tube1)
  (no-empty tube1)
)
)
```

Notice that the preconditions for the eat action in the Space Station scenario include the literals `(inventory-object tube1)` and `(not(empty tube1))`.

In the initial state we can observe that the literal `(no-empty tube1)` is included because of the CWA and given that the assertion `(empty tube1)` is not in the player ABox in the current state. The the following plan is found:

```
--------------------------------------------------
Begin plan
1 (take=patient tube1 toolbox1)
End plan
--------------------------------------------------
Total elapsed time in milliseconds: 39
```

Hence, we expect FrOzA's answer to be:

```
...
(4) Player: eat the foodpaste
(5)  FrOzA: You have the tube of foodpaste.
            You eat the tube of foodpaste.
```

However, when FrOzA tries to execute the sequence of actions `[[take(patient:tube1) eat(patient:tube1)]]` it fails because in the world it is the case that `(empty tube1)` and the answer is:

```
...
(4) Player: eat the foodpaste
(5)  FrOzA: You can't do this!
            The tube of foodpaste is empty.
```

This is an example of why it is necessary to verify the actions preconditions on the world also when plans are found by the planner as we discussed in Chapter 4.

Chapter 6

Concluding remarks

One of the aims in the original development of FrOz was to integrate RACER, a state-of-the-art reasoner to the architecture, in order to provide a dialogue system with necessary inference capabilities. This thesis is another step in the same direction, as we show how to integrate planning, a different kind of inference, into FrOz.

The achievement of this goal involved several tasks. To begin with, we identified the essential planner characteristics, we chose a suitable state-of-the-art planner and we evaluated its performance. Moreover, we designed the planner integration into FrOz architecture. Furthermore, we implemented the integration into FrOz code, and we evaluated the new capabilities in the dialogue system.

The addition of planning capabilities enhanced the dialogue system in several ways. To begin with, the interaction is more effective (less turns are needed to reach the same goal); and more robust as, frequently, FrOzA is able to find a suitable sequence of actions in cases where FrOz would break down. In addition, the use of FrOzA planning capabilities increases as the player knows more about the game world, so FrOzA behaviour adapts to the player experience. What is more, this work did not define an ad-hoc procedure to handle the task at hand. Instead, a general procedure was defined which specifies how to generate a PDDL planning specification from a formal representation of the scenarios of the dialogue system FrOz. These scenarios are generic, they are specified through description logic knowledge bases and a STRIPS-based representation of actions, hence the translation presented is generic enough to be applicable in other dialogue systems that use similar encoding formalisms. To end with, although there is no planner today able to handle the complete planning task specification at hand, we showed that the game performance can benefit just from an incomplete codification.

In this work we also profited from a state-of-the-art system, following the original FrOz development premise. We believe that finding the way to take advantage of current technologies is an important issue if our goal is to design a generic

dialogue system.

This thesis only explores some of the possibilities of the framework, and there are many more interesting research directions to investigate.

We have already discused some ways in which Blackbox behaviour can be tailored keeping in mind that FrOz is a computer game. However, there is room for improvements. As we have already noted in Chapter 4, in some cases, our new version of FrOz will not be able to come up with a plan, even if one exists, because the encoding described is incomplete. This is due to the fact that Blackbox input language is less expressive than the language supported by RACER. As arbitrary queries to RACER can be embedded in FrOz actions databases, we cannot expect to be able to cover them in full generality. Nonetheless, the proposed encoding does improve FrOz behaviour in many common cases. Maybe in the future, planners will be able to handle more expressive languages within the time restrictions imposed by an interactive system such as FrOzA.

Once the planner outputs a plan, there is one additional issue that should be taken into account. We want the game to perform actions for the player only if the game can assume that the plan is sufficiently 'simple'. We approached this fact planning on the player beliefs instead of the game knowledge so the planner can only came up with plans when the player already knows how to perform an action. However, there are actions (such as killing the dragon in the FairyTale-Castle scenario) that we never want FrOzA to execute on its own, because that would be given away the game. How can we guarantee this? One possibility is the following, given a game scenario some actions can be considered *minor* or trivial, while others cannot. This classification of actions into essential (which can only be performed explicitly by the player) and minor (which can be executed via planning) should be specified by the scenario designer; and the planning domain would include only those actions that are specified as minor.

Although in this work we only modified the way in which the game handles the execution of actions, this modification has interesting consequences in other natural language processing modules, such as generation. The generation component could be enhanced so that it will render differently the case where the changes it is reporting over the world were explicitly indicated by the player, and the case where they were caused by actions automatically performed by the game (as discussed in Chapter 4).

Moreover, the integration of a generic planner we have described can be useful for other interesting extensions of the game. For example, offering 'hints' to the player during the game execution, or answering player's questions. Such extensions will be the topic of our future research. But the next issue in our research agenda is adequacy evaluation. We should empirically test whether the plans we obtain from Blackbox in our setup are indeed useful in real game situations. This can

only be verified by compiling and analysing a corpus of actual interaction with FrOzA in more real and complex scenarios.

Bibliography

[1] S. Ali, editor. *Knowledge Representation for Natural Language Processing in Implemented Systems. Papers from the AAAI Fall Symposium.* The AAAI Press, 1996. Technical Report FS-94-04.

[2] J. Allen. *Natural language understanding (2nd ed.).* Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1995.

[3] J. Allen, D. Byron, M. Dzikovska, G. Ferguson, L. Galescu, and A. Stent. Towards conversational human-computer interaction. *AI Magazine*, 2001.

[4] James Allen. *Natural language understanding.* Addison Wesley, 2nd edition, 1994.

[5] James F. Allen. The TRAINS project: A case study in building a conversational planning agent. *Journal of Experimental and Theoretical AI (JETAI)*, 7:7–48, 1995.

[6] J. Austin. *How to do Things with Words.* Oxford University Press, New York, 1962.

[7] F. Baader, B. Hollunder, B. Nebel, H. Profitlich, and E. Franconi. An empirical analysis of optimization techniques for terminological representation systems or: Making KRIS get a move on. *Journal of Applied Intelligence*, 4:109–132, 1994.

[8] Luciana Benotti. Codifying actions for a text adventure. Technical report, Free University of Bolzano, 2005.

[9] Luciana Benotti. "DRINK ME": Handling actions through planning in a text game adventure. In Janneke Huitink and Sophia Katrenko, editors, *XI ESSLLI Student Session*, pages 160–172, 2006.

[10] N. Bernsen, L. Dybkjaer, and L. Dybkjaer. *Designing Interactive Speech Systems: From First Ideas to User Testing.* Springer-Verlag New York, Inc., 1997.

[11] A. Blum and M. Furst. Fast planning through planning graph analysis. In *Proc. of the 14th International Joint Conference on Artificial Intelligence (IJCAI 95)*, pages 1636–1642, 1995.

[12] D. Bobrow, R. Kaplan, M. Kay, D. Norman, H. Thompson, and T. Winograd. GUS, a frame-driven dialog system. *Artificial Intelligence*, 8:155–173, 1977.

[13] A. Borgida, R. Brachman, D. McGuinness, and L. Resnick. CLASSIC: A structural data model for objects. In *SIGMOD '89: Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, pages 58–67, New York, NY, USA, 1989. ACM Press.

[14] R. Brachman and H. Levesque. The tractability of subsumption in frame-based description languages. In *AAAI*, pages 34–37, 1984.

[15] R. Brachman and J. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9(2):171–216, 1985.

[16] T. Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1-2):165–204, 1994.

[17] David Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32(3):333–377, 1987.

[18] P. Cohen. *On knowing what to say: planning speech acts.* PhD thesis, 1978.

[19] K. Colby, S. Weber, and F. Hilf. Artificial paranoia. *Artificial Intelligence*, 2(1):1–25, 1971.

[20] D. Duchier and R. Debusmann. Topological dependency trees: A constraint-based account of linear precedence. In *Proceedings of the 39th ACL*, 2001.

[21] M. Ernst, T. Millstein, and D. Weld. Automatic SAT-compilation of planning problems. In *Proc. of the 14th International Joint Conference on Artificial Intelligence (IJCAI 97)*, pages 1169–1177, 1997.

[22] G. Ferguson and J. Allen. TRIPS: an integrated intelligent problem-solving assistant. In *Proceedings of the 10th conference on innovative applications of artificial intelligence*, pages 567–572, Menlo Park, CA, USA, 1998. American Association for Artificial Intelligence.

[23] G. Ferguson, J. Allen, and B. Miller. TRAINS-95: Towards a mixed-initiative planning assistant. In *Proceedings of the Conference on Artificial Intelligence Planning Systems*, pages 70–77, Edinburgh, Scotland, 1996.

[24] R. Fikes, P. Hart, and J. Nils. Learning and executing generalized robot plans. *Artificial Inteligence*, 3:251–288, 1972.

[25] R. Fikes, P. Hart, and N. Nilsson. Learning and executing generalized robot plans. In B. Buchanan and D. Wilkins, editors, *Readings in Knowledge Acquisition and Learning: Automating the Construction and Improvement of Expert Systems*, pages 485–503. Kaufmann, San Mateo, CA, 1993.

[26] C. Gardent and E. Jacquey. Lexicalization as a description logic inference task. *Actas del IOCS'03 (Inference in Computational Semantics)*, 2003.

[27] C. Green. Application of theorem proving to problem solving. In *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 219–240, 1969.

[28] B. Grosz. The representation and use of focus in a system for understanding dialogs. In *Proc. of the International Joint Conference on Artificial Intelligence*, pages 67–76, 1977.

[29] V. Haarslev and R. Möller. RACER system description. In *Proceedings of International Joint Conference on Automated Reasoning (IJCAR 01)*, number 2083 in LNAI, pages 701–705, Siena, Italy, 2001. `http://www.racer-systems.com/technology/contributions`.

[30] V. Haarslev and R. Möller. *RACER User's Guide and Reference Manual*, april 2004.

[31] E. Hinkelman and J. Allen. Two constraints on speech act ambiguity. In *ACL*, pages 212–219, 1989.

[32] I. Horrocks. FaCT and iFaCT. In P. Lambrix, A. Borgida, M. Lenzerini R. Möller, and P. Patel-Schneider, editors, *Proc. of the 1999 International Workshop on Description Logics (DL'99)*, pages 133–135, 1999.

[33] A. Joshi and Y. Schabes. Tree-adjoining grammars. In *Handbook of formal languages. Vol 3: Beyond words*, pages 69–123. Springer-Verlang New York, Inc., 1997.

[34] H. Kautz and J. Allen. Generalized plan recognition. In *AAAI*, pages 32–37, 1986.

[35] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In H. Shrobe and T. Senator, editors, *Proc. of the 13th National Conference on Artificial Intelligence and the Eighth Innovative*

Applications of Artificial Intelligence Conference, pages 1194–1201, Menlo Park, California, 1996. AAAI Press.

[36] H. Kautz and B. Selman. BLACKBOX: A new approach to the application of theorem proving to problem solving. In *Workshop Planning as Combinatorial Search, AIPS-98, Pittsburgh*, 1998.

[37] H. Kautz and B. Selman. Unifying SAT-based and graph-based planning. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI 99)*, pages 318–325, Stockholm, Sweden, 1999. `http://www.cs.washington.edu/homes/kautz/satplan/blackbox/index.html`.

[38] J. Koehler, B. Nebel, J. Hoffmann, and Y. Dimopoulos. Extending planning graphs to an adl subset. In S. Steel and R. Alami, editors, *Proc. 4th European Conference on Planning*, pages 273–285, London, UK, 1997. Springer-Verlag.

[39] A. Koller, R. Debusmann, M. Gabsdil, and K. Striegnitz. Put my galakmid coin into the dispenser and kick it: Computational linguistics and theorem proving in a computer game. *Journal of Logic, Language and Information*, 13(2):187–206, 2004.

[40] V. Lifschitz. On the semantics of STRIPS. In M. Georgeff and A. Lansky, editors, *Proc. of the 1986 Workshop: Reasoning about Actions and Plans*, pages 1–9, Timberline, Oregon, 1986. M. Kaufmann.

[41] R. MacGregor and R. Bates. The Loom knowledge representation language. Technical Report ISI/RS-87-188, Information Science Institute, University of Southern California, Marina del Rey, California, 1987.

[42] D. McDermott. Pddl - the planning domain definition language, 1998.

[43] M. Minsky. A framework for representing knowledge. In R. Brachman and H. Levesque, editors, *Readings in Knowledge Representation*, pages 246–262. Morgan-Kaufmann, Los Altos, 1974.

[44] R. Perrault and J. Allen. A plan-based analysis of indirect speech acts. *Computational Linguistics*, 6(3-4):167–182, 1980.

[45] J. Quantz and C. Kindermann. Implementation of the BACK system, version 4. Technical Report KIT-Report 78, FB Informatik, Technische Universität Berlin, Berlin, Germany, 1990.

[46] M. Quillian. Word concepts: a theory and simulation of some basic capabilities. *Behavioral Science*, 12:410–430, 1967.

[47] E. Reiter and R. Dale. Building applied natural language generation systems. *Journal of Natural Language Engineering*, 3(1):57–87, 1997.

[48] N. Rychtyckyj. DLMS: An evaluation of KL-ONE in the automobile industry. In *KR*, pages 588–596, 1996.

[49] J. Searle. *Speech Acts: An Essay in the Philosophy of Language*. Cambridge University Press, Cambridge, London, 1969.

[50] A. García Serrano and J. Calle Gómez. A cognitive architecture for the design of an interaction. *Lectures Notes in Computer Science*, 2446:82–89, 2002.

[51] A. García Serrano, P. Martínez, and J. Hernández. Using AI techniques to support advanced interaction capabilities in a virtual assistant for e-commerce. *Expert Systems with Applications*, 26(3):413–426, 2004.

[52] A. García Serrano, P. Martínez, and L. Rodrigo. Adapting and extending lexical resources in a dialogue system. In *Proceedings of the workshop on Human Language Technology and Knowledge Management*, pages 1–7. Association for Computational Linguistics, 2001.

[53] J. Sowa, editor. *Principles of Semantic Networks: Explorations in the Representation of Knowledge*. Morgan Kaufmann, Los Altos, 1991.

[54] Kristina Striegnitz. *Generating Anaphoric Expressions. Contextual Inference in Sentence Planning*. PhD thesis, Loria at INRIA, Nancy, France, November 2004.

[55] M. Strube. Never look back: An alternative to centering. In *COLING-ACL*, 1998.

[56] Gerald Jay Sussman. *A Computer Model of Skill Acquisition*. Elsevier Science Inc., New York, NY, USA, 1975.

[57] R. H. Thomason, M. Stone, and D. DeVault. Enlightened update: A computational architecture for presupposition and other pragmatic phenomena. In *Workshop in Presupposition Accomodation*, page To Appear. The Ohio State University, 2006.

[58] D. Warren. WARPLAN: A system for generating plans. Memo 76, Department of Computational Logic, University of Edinburgh, June 1976.

[59] RACER's Website. `http://www.racer-systems.com/`. Visitado por última vez en Agosto, 2005.

[60] RICE's Website. `http://www.blg-systems.com/ronald/rice/`. Visitado por última vez en Agosto, 2005.

[61] J. Weizenbaum. ELIZA: a computer program for the study of natural language communication between man and machine. *Commun. ACM*, 9(1):36–45, 1966.

[62] D. Weld. Recent advances in ai planning. *AI Magazine*, 20(2):93–123, 1999.

[63] R. Wilensky, D. Chin, M. Luria, J. Martin, J. Mayfield, and D. Wu. The berkeley UNIX consultant project. *Comput. Linguist.*, 14(4):35–84, 1988.

[64] W. Woods. What's in a link: Foundations for semantic networks. In D. G. Bobrow and A. Collins, editors, *Representation and Understanding*, pages 35–82. Academic Press, New York, 1975.

Appendix A

Implementation Issues

The aim of this appendix is to document the implementation of FrOzA. And it is mainly intended to assist those that need to mantain this version of the system.

A.1 The Implemented Architecture

Both FrOz and FrOzA are implemented in the Oz functional language[1]. The architecture of FrOzA is presented in Figure 4.1 and is repeted here:

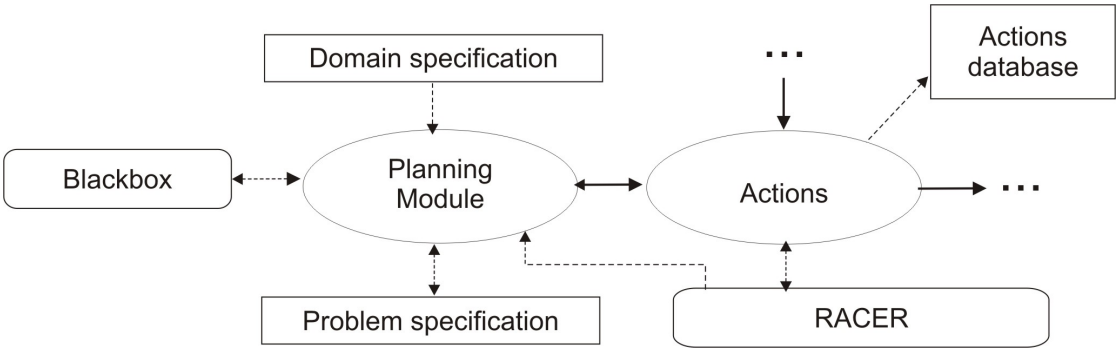

Figure A.1: FrOzA architecture

The directory where FrOzA code is stored has a structure that mimic the system architecture and is depicted in the Figure A.2.

We will fucus our attention in the Actions Module and in the Dynamic Planning Module which are the ones involved in this thesis.

The dynamic planning module is stored in the Actions directory because in the general architecture, this is the only other natural language processing module it interacts with.

[1] `http://www.mozart-oz.org/`

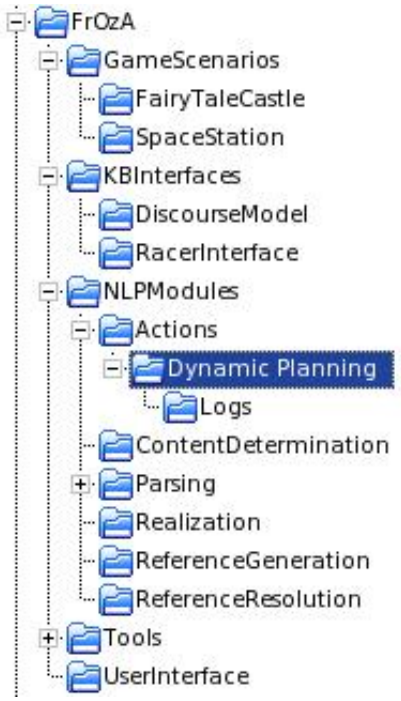

Figure A.2: FrOzA directory structure

Let us analyse the implementation of these two modules.

A.2 Actions Module

A.2.1 Code Files

The files that take part of the Actions Module are the following

A.2.2 Input

The input is a list of lists of action names. E.g. for the command "take the green frog" the input would be:

```
[[take(patient:[priesemut])]]
```

And for the command "drop it and take the brown frog" it would be:

```
[[drop(patient:[priesemut])
  take(patient:[priesemut2])]]
```

The outer list collects different possible "interpretations" of the command. If the command had referential or syntactic ambiguities which could not be resolved by the module for referent resolution. Interpretations are again lists, representing sequences of actions.

This input is the output of the interpretation modules (i.e., parsing and resolution of referring expressions) or the dynamic planning module.

A.2.3 Output

The output is a list of readings corresponding to the different interpretations that came in. Each interpretation is a sequence of actions. The output shows which of these actions can be executed, which of these actions fail (because its preconditions are not satisfied), and which of these actions were not tested (as soon as the first failing action is found the rest of the sequence of actions is not evaluated any more).

This is what the output looks like for an interpretatation that consists of only one action which can be executed.

```
[reading(executable:[conjunct(action:ACTION
                              failPre:FAILED_PRECONDITIONS
                              succPre:SUCCESSFULL_PRECONDITIONS
                              untestedPre:UNTESTED_PRECONDITIONS
                              uk:UPDATES_ON_PLAYER_KNOWLEDGE)]
         failed:nil
         untested:nil)]
```

This output is then passed on to the content determination module.

A.2.4 Main function: Executable.oz

Here is what the function Executable (the main function of this module) does.

Input is a list of lists of actions that is interpreted as a disjunction of conjunctions of actions. The outer list corresponds to alternative interpretations while the inner list of actions is a sequence of actions that all have to be carried out.

1. NP-conjunctions are being expanded by the auxiliary function Preprocessing.oz.

2. Get all actions from the action database using LookupAction.oz. If one action name corresponds to several entries in the action database all of these are returned as a list.

3. So, what we have then is a list of lists of lists of entries. Where the innermost list corresponds to ambiguities in the actions database (several actions with the same name; they should be disambiguated by their preconditions). Multiply these ambiguities out so that we again have a list of lists of entries.

4. For each disjunct (element of the outer most list), test whether it can be executed in a clone of the current game A-Box. This is done by function ExecutableDisjunct. The result is a list of alternative action sequences annotated for whether they are executable or not.

5. In this step we have tree possible situations:

   - If there is only one executable sequence of actions, execute it.
   - If there is no executable sequence of actions, then we can have two possible situations:
     - If the input is the output of the interpretation modules then pass the control to the dynamic planning module.
     - If the input is the output of the dynamic planning module then raise an error. The generation module will then generate an error message.
   - If there are several executable sequences, pass the information on to the content determination. This is actually a mistake in the original FrOz implementation, an error should be raised here as well.

A.2.5 Auxiliary functions

The auxiliary functions used by this module are:

InitActions.oz: Reads the action database and puts them into a dictionary. The action slot of each entry being the key.

Preprocessing.oz: Function for multiplying out plural NPs and coordinate NPs. I.e., something like `<take([a,b])>` will become `<take(a), take(b)>`.

A.3 Dynamic Planning Module

A.3.1 Code Files

The files that take part of the Dynamic Planning Module are the following:

A.3.2 Input

The input of the dynamic planning module is the output of the actions module when there is no executable sequence of actions. I.e., some action in the conjunction has failed.

This is what the output looks like for an interpretation that contains a failed action.

```
[reading(executable: EXECUTABLE_ACTIONS
         failed:[conjunct(action:FAILED_ACTION
                          failPre:FAILED_PRECONDITIONS
                          succPre:SUCCESSFULL_PRECONDITIONS
                          untestedPre:UNTESTED_PRECONDITIONS
                          uk:UPDATES_ON_PLAYER_KNOWLEDGE)]
         untested: UNTESTED_ACTIONS)]
```

A.3.3 Output

The output of the dynamic planning module is the input of the actions module which only includes one conjunction of actions. The general format would be:

```
[[ACTIONS]]
```

This list of list of actions is then passed on to the actions module.

A.3.4 Main function: FindPlan.oz

Here is what the function FindPlan (the main function of this module) does.

1. The planning problem file is created using the function ProblemFile. This function queries the player ABox through RACER and specifies the problem initial state and the goal.

2. Blackbox is invoqued with the problem and domain files. All files (the planning problem, the planning domain and the plan) are labeled with a time stamp for logging purposes.

3. Blackbox output is parsed by the function FormatPlan into the output expected by the Actions module:

```
[[ACTIONS]]
```

4. Append the plan before the failed action and pass the control to the actions module. The output schema would be:

```
[[EXECUTABLE_ACTIONS PLAN FAILED_ACTION UNTESTED_ACTIONS]]
```

A.3.5 Auxiliary functions

The auxiliary functions used by this module are:

ProblemFile: This function job is as follows:

1. Create the problem file `problemTIMESTAMP.PDDL`.
2. All the objects (instances) in the player ABox are retrieved and then, for each object, all the dynamic and static concepts they belong to (as well as the dynamic concepts each object does nott belong to –in order to calculate the Close World Assumption for the predicates afterwards).
3. The role assertions in the player ABox and retrieved and then all the possible two-element combinations of objects are calculated (for the Close World Assumption).
4. The objects with their types are printed in the PDDL problem file. As well as the initial state assertions (positive and negated dynamic concepts and role assertions).
5. The goal is printed in the file: the instantiated preconditions of the `FAILED_ACTION`.

FormatPlan: The input of this function is the plan file generated by Blackbox planTIMESTAMP.PDDL. If the file content is `"NO SOLUTION"` then returns `PLAN = nil`. Otherwise, the file content has the general format:

```
((ACTION=SEMANTIC_ROLES PARAMETERS)+)
```

This is then reformated into the expected actions module input, a list of lists of records:

```
[[ACTION(SEMANTIC_ROLE:[PARAMENTER]*)+]]
```