Building a Text Adventure on Description Logic

Malte Gabsdil, Alexander Koller, Kristina Striegnitz
Dept. of Computational Linguistics
Saarland University, Saarbrücken, Germany
{gabsdil|koller|kris}@coli.uni-sb.de

Abstract

We describe an engine for a computer game which employs techniques from computational linguistics and theorem proving based on description logic. We show how we represent a world model as a DL knowledge base and then illustrate how we use it in the computational linguistics modules with the examples of analyzing and generating referring expressions.

1 Introduction

In this paper, we describe an engine for text adventures which employs techniques from computational linguistics and theorem proving based on description logic. The system is being developed at Saarland University as a student project. Its purpose is twofold: Players should be able to interact more naturally with the game, and we envisage a use as a testbed for computational linguistics modules.

Text adventures are a classical form of computer games which were most popular in the eighties. The player interacts with the game world (e.g. the rooms and objects in a space station) by typing natural-language commands and the computer provides feedback in the form of natural-language descriptions of the world and of the results of the player's actions. Typically, the user has to solve puzzles to win the game; an example interaction is shown in Fig. 1.

Text adventures have since gone somewhat out of fashion. One reason for this was the advent of more powerful graphics hardware, but another is that even the most advanced games of the eighties, which accepted input that went well beyond simple two-word sentences, suffered from some irritating limitations. Maybe most striking is what we call the *identification problem*: Sometimes the game does not allow the user to refer to an object with the exact same words that the game itself used for it (Fig. 2, taken from [3]). This is unsurprising, since the output of the game is hard-coded and elaborate, whereas the input has to be analyzed by a very simple parser.

```
Observation Lounge
This is where the station staff and visitors come to relax. There
are a lot of tables and chairs here, a large observation window,
and a plush carpet. In the corner you can see an AstroCola dispenser.
A tube leads up to the station's main corridor.

>put my galakmid coin into the dispenser
Click.
The dispenser display now reads "Credit = 1.00".

>push diet astrocola button
You hear a rumbling noise in the dispenser, but nothing appears in the
tray.

>kick dispenser
A can drops into the tray. Amazing! The oldest trick in the book, and
it actually worked.
```

Figure 1: An example interaction with a text adventure, taken from [7].

```
Cupboard
When you aren't lying on the bed, you usually stay in here, snug and
safe with your friends atop the warm pile of clothes. Your warm
winter jacket is here, which may be just as well, it's a little chilly.

>take the warm winter jacket
You can't see any such thing.

>take the winter jacket
You can't see any such thing.

>look at the jacket
A smart green jacket with big pockets, teddy bear sized.

>take the smart green jacket
You can't see any such thing.

>take the jacket with big pockets
I only understood you as far as wanting to take the green jacket.

>take the green jacket
Taken.
```

Figure 2: The identification problem.

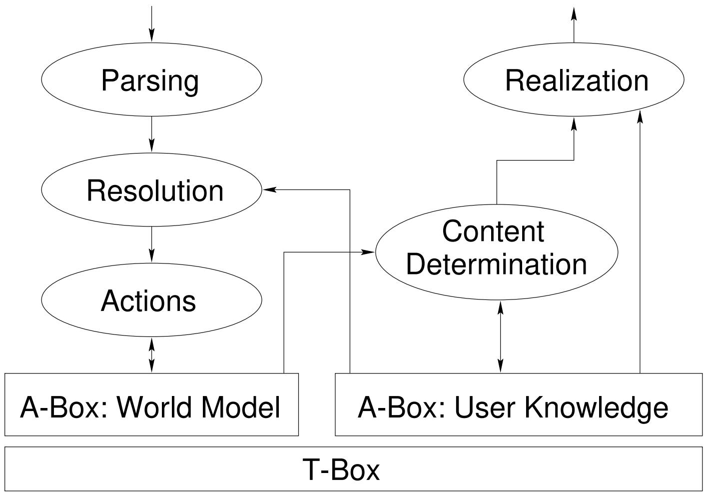

Figure 3: Architecture

Our system attempts to overcome this and other limitations by employing state-of-the-art techniques from computational linguistics, such as a real parser for English and a component for the automatic generation of the system's answers. Underlying the system is a world model based on description logic, which is used by almost every component of the NLP system. In our implementation we use the RACER system [5] because it provides support for A-Box reasoning, which is essential for us.

The paper is organized as follows: We will first sketch the general architecture of the system and its components (Section 2) and describe the DL world model (Section 3). Then we will briefly illustrate how we make use of DL inferences in the NLP modules by first showing how to analyze the meaning of referring expressions (Section 4), and then how to generate such referring expressions (Section 5). Section 6 concludes the paper and presents some ideas for future work.

2 Architecture

The general architecture of the game engine is shown in Fig. 3. The user's input is first parsed – that is, its syntactic structure is determined, using an efficient parser for dependency grammar [2]. Next, *referring expressions* (such as *the toolbox*) in the input are resolved to objects in the world. The result is a ground term that indicates the action the user wants to take.

This term is used to retrieve action descriptions from a database; the entry for "open" is shown in Fig. 4. Action descriptions are STRIPS-like operators defining preconditions and effects of the action. In addition, they specify in the 'uk' slot how the user knowledge has to be updated when the action is performed.

open(pat:X)	
pre:	closed(X), unlocked(X)
effects:	add: open(X) delete: closed(X)
uk:	add: open(X), describe(X) delete: closed(X)

Figure 4: The operator for the “open” action.

The term that was produced by the resolution component is matched with the head of the operator, binding the variables in the action description.

If the preconditions are satisfied, the world model is updated according to the ‘effects’ slot, and the instantiated contents of the ‘uk’ slot are passed on to the content determination component, which computes what information has to be verbalized by the generation module. This verbalization process is then carried out by a realization component based on Lexicalized Tree Adjoining Grammar [6, 10], which produces English text.

3 The World Model

The world model of the game engine is encoded as a DL knowledge base. The T-Box specifies the concepts and roles which are available in the world and defines complex concepts used e.g. by the resolution module (see below). The A-Boxes state which concepts and roles hold of the individuals in the world.

In the system, we use two different A-Boxes. One stores the current state of the world; it is used to determine whether the preconditions of an action are satisfied in the world, and, if this is the case, is updated with the action's effects. The function of the other A-Box is to keep track of the player's knowledge. It is used in the language-processing modules – for instance, referring expressions must be evaluated with respect to the user's knowledge –, and is updated by the content determination when it has determined what new information should be verbalized. The two A-Boxes share the same T-Box, but will typically be different. For instance, the world A-Box will usually contain more individuals than the user A-Box because the player will not have explored the world completely and will therefore not have seen all the individuals. On the other hand, it can be useful to deliberately hide effects of an action from the user, e.g. if pushing a button has an effect in a room that the player cannot see.

A fragment of the A-Box describing the state of the world is shown in Fig. 5. The T-Box specifies that the world is partitioned into three parts: rooms, objects, and players. The individual ‘myself’ is the only instance that we ever define of the concept ‘player’. Individuals are connected to their locations (i.e. rooms,

room(scooter-bridge)	toolbox(t1)
hammer(h1)	player(myself)
saw(s1)	silver(t1)
closed(t1)	unlocked(t1)
has-location(t1, scooter-bridge)	has-location(h1, t1)
has-location(myself, scooter-bridge)	has-location(s1, t1)
...	

Figure 5: A fragment of the world A-Box.

container objects, or players) via the 'has-location' role; the A-Box also specifies what kind of object an individual is (e.g. 'toolbox') and what properties it has ('closed', 'silver'). The T-Box then contains axioms such as 'toolbox $\sqsubseteq$ object', 'silver $\sqsubseteq$ colour', etc., which establish a taxonomy among concepts.

These definitions allow us to add axioms to the T-Box which define more complex concepts. One is the concept 'here', which contains the room in which the player currently is – that is, every individual which can be reached over a has-location role from a player object.

$$\text{here} \doteq \exists \text{has-location}^{-1}.\text{player}$$

Another useful concept is 'accessible', which contains all individuals which the player can manipulate.

$$\text{accessible} \doteq \forall \text{has-location}.\text{here} \sqcup \forall \text{has-location}.(\text{accessible} \sqcap \text{open})$$

All objects in the same room as the player are accessible; if such an object is an open container, its contents are also accessible. As the player itself is by definition 'open', this includes the player's inventory.

Finally, we should mention that inside the action processing module, we create multiple temporary A-Boxes to allow for a more benevolent handling of ambiguity. Imagine the player types an ambiguous sentence, such as "put the apple in the box on the table". We will explain below how the resolution module can sometimes filter out some readings of such an ambiguity, but in this case, let's assume that it cannot decide whether the user meant putting "the apple" into "the box on the table", or "the apple in the box" onto "the table". It will hand both alternatives down to the action processing component.

Here we pursue all possible meanings of the sentence in parallel. For each reading, we create a copy of the current world A-Box, and then attempt to perform the action on the copy. This can be nontrivial because it may be possible to express sequences of actions with a single sentence, and these have to be performed one after another. If it turns out that we can only successfully perform the actions in one of the readings (e.g. because the player does not hold the apple in the box), we can commit to this reading without the user ever noticing that we had trouble understanding what he meant. Otherwise, we have to report an error.

4 Resolution of Referring Expressions

Referring expressions, such as *the toolbox*, *it*, or *a hammer*, link linguistic forms to objects in the world (the *referents* of the referring expressions). The player in our application will typically use definite descriptions (*the toolbox*) or pronouns (*it*) to refer to the objects on which he wants to perform an action. It is therefore essential to resolve these expressions to the actual individuals in the player-knowledge A-Box. As an example reconsider the A-Box in Fig. 5: We first have to resolve *the toolbox* to the RACER individual t1 before any action on this object can be carried out.

Resolving Definite Descriptions Definite descriptions of the form *the toolbox*, *the green apple*, or *the hammer in the toolbox* refer to an object that matches their restriction (*toolbox*, *green apple*, etc.). In a first approximation, we also take them to refer *uniquely*: That is, there must be exactly one object in the world that matches the restriction [9]. For *the toolbox*, this restriction is simply the concept 'toolbox'. We furthermore assume that the player will only try to refer to 'accessible' objects. This avoids confusion with other objects that would match the same description but are not in the same room as the player, i.e. objects that are not locationally salient. Thus we can retrieve a list of all potential referents for *the toolbox* by evoking the RACER query

$$\texttt{(concept-instances}\ \text{toolbox} \sqcap \text{accessible)}$$

Assuming that t1 is actually already present in the player A-Box, this returns the list `(t1)`. As it contains exactly one element, the reference succeeds; otherwise we would have rejected the command with an error message. Note that we always interpret reference with respect to the player's knowledge: The presence of toolboxes unknown to the player does not lead to an ambiguous reference.

More complicated definites are simply translated into more complex concepts. Our general strategy here is to push as much of the work into the DL inference problems and let RACER's optimizations work for us. For example, *the hammer in the toolbox* translates to the query

$$\texttt{(concept-instances}\ \text{hammer} \sqcap \text{accessible} \sqcap \exists\text{has-location.toolbox)}$$

The fact that definite references may fail (i.e. no referent in the world model can be found that matches the restriction) can be very helpful when we are faced with more than one possible syntactic derivation for an input. For example, the sentence *Unlock the toolbox with the key* is ambiguous: *The key* could either be an instrument used in the unlocking, or it could modify *the toolbox*, as in *the toolbox with the red handle.* In this example, we will not be able to find a referent for the constituent *the toolbox with the key* in the second parse, and will therefore only pass on the (resolved) first reading to the actions module.

Resolving Pronouns Unlike definite descriptions, pronouns do not provide much information about the object they refer to. However, the linguistic restrictions on which objects can be referred to by pronouns are much stricter.

We make use of a discourse model inspired by Strube's S-list [12] to determine the objects pronouns might refer to (their *antecedents*). The idea behind the S-list is to keep an ordered record of salient objects that have been introduced during a discourse and which are therefore most likely to be antecedents for pronouns. We associate every element in the S-list with agreement features (gender and number), its information status and text position (both needed to determine list-order; see [12]), as well as the RACER individual it refers to in the player A-Box. Resolving a pronoun then comes down to a lookup in the S-list: We simply take the first element that matches the pronoun's agreement restrictions.

The discourse model is updated incrementally and can be accessed by both the resolution and the generation module (see below). We can therefore resolve inter-sentential pronouns like *Take the apple and eat it* as well as simple cases of cross-speaker anaphora [4] as in the following short dialogue:

GAME:	*There is an apple on the table.*
PLAYER:	*Take it.*

5 Generation of Referring Expressions

The purpose of the generation module is to describe the environment the player is in and how his actions affect the game world. As is common practice in natural language generation systems, it consists of the two submodules *content determination* and *realization* (see e.g. [13]). Content determination assembles the information that has to be communicated to the player, and then passes it on to the realization module to cast it into a text.

The information that should be communicated to the user is essentially the value of the 'uk' slot of the instantiated action schema, with two notable differences. First, individuals can of course not be called by their internal names (such as t1), so we must again generate a referring expression that names them. Second, special atoms like 'describe(t1)' are taken as requests to generate a description of t1.

ds:	l1, l2
new:	l1:open(t1), l2:contains(t1,[h1,s1]), hammer(h1), saw(s1)
old instances:	t1

Figure 6: Output of Content Determination (Example: *Open the toolbox*)

Fig. 6 shows an example output of the content determination module for the action *open the toolbox*, as specified in Fig. 4. It tells the realization component to generate two sentences, called internally 'l1' and 'l2': one expresses the fact that the toolbox t1 is open now, and the other one introduces the objects that are contained in the toolbox. Depending on contextual factors, a possible output could be *The toolbox is now open. It contains a hammer and a saw.*

Referring to Objects Reference to *old* individuals, i.e. individuals the player already knows about, is mainly taken care of by the realization module (see below) as there is interaction with the surface form that is chosen for the referring expression.

In case the action schema asserts a fact about an individual r the player has *not* encountered before, this object will be introduced to the player by a *mini-description* stating the most obvious properties of the object – e.g., its (most specific) type, such as 'toolbox', and its colour. We can find out whether the individual is new to the player by checking whether it is an instance of the universal concept $\top$ in the player A-Box.

Suppose we want to generate a mini-description for the toolbox t1 in Fig. 5. The query

```
(individual-direct-types t1)
```

will return the list `(toolbox silver)`. Using concept subsumption checks, we can find out that 'toolbox' is the type of t1, and 'silver' is its colour; so both concepts go into the mini-description. (Subsumption checks are inexpensive because we can completely classify the T-Box when we start the system.) The realization component can use this information later to generate the expression like *a silver toolbox.*

Describing Objects There are two main situations in which object descriptions have to be produced. First, the player may ask explicitly for a description, for instance by saying *look at the toolbox.* In this case, a full detailed description of the object is required. Therefore, all (most specific) concepts of which the object is an individual as well as all role assertions in which the concept takes part are retrieved from the world model. Mini-descriptions (as above) are provided for the objects introduced through role assertions.

The second type of object description is intended for situations in which an action has changed the world in such a way that new information becomes

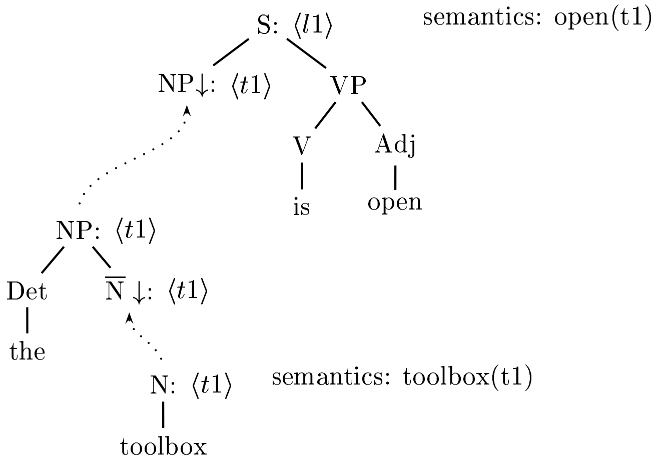

Figure 7: Realizing 'open(t1)'

accessible to the player. An example is the 'open' action, which makes the objects in the opened container visible. In this case, only new information, i.e. facts about the object that can be retrieved from the world model but don't follow from the user knowledge, should be contained in the description. At the moment, we derive this information from the specific action types. In the case of an 'open' action, not all role assertions are retrieved, but only those 'has-location' relations which point to objects included in the container. We aim to arrive at a more general solution eventually.

Realization The realization component produces a text to communicate the information assembled by the content determination to the player. In order for the text to be smooth and for the player to be able to correctly resolve references to objects, it is important that appropriate expressions are used for referring. For example, we want to refer to objects the player knows about with definite descriptions (*the toolbox*) and to new objects with indefinites (*a toolbox*). For new objects, we simply verbalize the concepts in the mini-description.

The correct verbalization of old objects is handled inside the main realization algorithm, which is based on [10]. In this framework realization comes down to assembling a sentence from the partial parse trees of a lexicalized tree-adjoining grammar [6]. Fig. 7 shows how *The toolbox is open* is built from fragments of syntax trees. The lexicon entries are associated not only with semantic information – which connects e.g. the word *toolbox* with the concept 'toolbox' –, but also with *pragmatic* information. This allows us to specify in the lexicon entry for *the* that definite descriptions must refer uniquely with respect to the player's knowledge. The resulting sentence (or little text) has to convey the information that content determination selected, it has to be syntactically viable (there must not be any holes in the result) and pragmatically appropriate.

To realize a reference to an old individual, we first compute the `individual-`

`direct-types` of the individual again; then we successively add the members of this list to the definite until the reference is unique, which we can check by computing the number of `concept-instances` as in Section 4. We follow standard practice in generation systems [1] by adding these concepts according to a predefined order of salience; first the type, then the colour, etc.

6 Conclusion and Outlook

In this paper, we have sketched the components of a text adventure engine which employs techniques from computational linguistics to make a more natural interaction with the game possible. The state of the world and the player knowledge are represented as description logic knowledge bases, and almost all language-processing modules utilize (A-Box and T-Box) inferences over these knowledge bases. We have looked more closely at the components for resolving and generating referring expressions, which solve the identification problem.

We are currently implementing the system; we hope to finish a prototype by September. The implementation is being done in the concurrent constraint programming language Oz [8], which allows us to reuse existing modules for parsing and realization [2, 11]. We communicate with the standalone version of RACER via sockets.

The system has much room for improvement, and indeed is designed in a modular fashion that will allow to replace specific components by more sophisticated ones. One line of future work could be to improve the part that resolves referring expressions; likewise, their generation is currently a very active research field in computational semantics, and new ideas could easily be incorporated into the system. One straightforward improvement of the realization component would be to add lexical entries that contain larger chunks of text, which could make the output more interesting to read.

Beyond these local changes, one can imagine many additions to the system's functionality. For instance, one could add speech recognition and generation components. In addition, it would be interesting to allow multiple instances of 'player' and make the game multi-user, or to model the world more realistically e.g. by replacing the room concept by coordinates in the world. But we believe that even the first version as it stands offers an interesting setup for exploring the use of description logic in computational linguistics.

Acknowledgments. We are grateful first of all to our students, without whose enthusiasm in implementing the system the game would have remained an idea. We are indebted to Ralph Debusmann for his contributions to the parsing component and the syntax-semantics interface. Carlos Areces introduced us to the new world of efficient DL provers, and Volker Haarslev and Ralf Möller were wonderfully responsive in providing technical support for RACER. Special thanks go

to Gerd Fliedner, in a discussion with whom the idea for employing techniques of computational linguistics in a text adventure engine came up first.

References

[1] Robert Dale and Ehud Reiter. Computational interpretations of the gricean maxims in the generation of referring expressions. *Cognitive Science*, 18:233–263, 1995.

[2] Denys Duchier and Ralph Debusmann. Topological dependency trees: A constraint-based account of linear precedence. In *Proceedings of the 39th ACL*, Toulouse, France, 2001.

[3] David Dyte. A Bear's Night Out. Text adventure. Available at `http://www.covehurst.net/ddyte/abno/`, 1997.

[4] Nissim Francez and Jonathan Berg. A Multi-Agent Extension of DRT. In H. Bunt, R. Muskens, and G. Rentier, editors, *Proceeding of the* 1^{st} *International Workshop on Computational Semantics*, pages 81–90, 1994.

[5] Volker Haarslev and Ralf Möller. RACER System Description. In *Proceedings of IJCAR-01*, Siena, 2001.

[6] Aravind Joshi and Yves Schabes. Tree-Adjoining Grammars. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, chapter 2, pages 69–123. Springer-Verlag, Berlin, 1997.

[7] David Ledgard. Space Station. Text adventure, modelled after a sample transcript of Infocom's *Planetfall* game. Available at `http://members.tripod.com/~infoscripts/planetfa.htm`, 1999.

[8] Mozart Consortium. The Mozart Programming System web pages. `http://www.mozart-oz.org/`, 1999.

[9] Bertrand Russell. On Denoting. *Mind*, 14:479–493, 1905.

[10] Matthew Stone and Christine Doran. Sentence planning as description using tree adjoining grammar. In *Proceedings of ACL*, pages 198–205, 1997.

[11] Kristina Striegnitz. Model Checking for Contextual Reasoning in NLG. In P. Blackburn and M. Kohlhase, editors, *Proceedings of ICOS-3*, Siena, 2001.

[12] Michael Strube. Never Look Back: An Alternative to Centering. In *COLING-ACL*, pages 1251–1257, 1998.

[13] H.S. Thompson. Strategy and Tactics in language production. In *Papers from the 13th Regional Meeting of the Chicago Linguistic Society*. 1977.