Playing With Description Logic
Application Description

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

1 Introduction

Text adventures are a classical form of computer games which were most popular in the eighties. The goal of these games is to solve puzzles by interacting with the game world (e.g. the rooms and objects in a space station). This interaction takes place by typing in natural-language commands; the player receives written feedback from the computer, e.g. as follows:

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

In this paper, we describe an implementation of a text adventure engine which has a description logic (DL) knowledge base at its core and employs techniques from computational linguistics to interpret the player input and to generate the game's responses. Our system shows some interesting interactions between the DL knowledge base and the language processing components. It turns out that the DL infrastructure is useful in both the analysis of the player input (resolution of referring expressions) and the generation of system utterances (content determination and realization modules).

A more detailed account of the NLP modules and their interaction with the reasoner can be found in (Gabsdil et al. 2001). Here we try to give a flavour of the implementation by showing more of the low-level details necessary for processing one complete example. The implementation was carried out as a student project at Saarland University. We used the concurrent constraint programming language Oz (Mozart Consortium 1999), which allowed us to reuse existing modules for parsing and realization (Duchier and Debusmann 2001; Striegnitz 2001). As a DL prover we used the RACER system (Haarslev and Möller 2001) because it provides support for A-Box reasoning, which is essential for us. In addition, its standalone version can connect to our system via sockets.

In the following section, we will give an overview over the architecture of the system. We then describe the DL knowledge bases modeling the status of the game in Section 3. In Section 4 we go through an example illustrating the contribution of each module and its interaction with the DL knowledge bases. Finally, in Section 5, we provide some preliminary discussion of our decision to use description logic and how well the prover could deal with the inference tasks that are required by our application.

2 Architecture

The general architecture of the game engine, shown in Fig. 1, consists of a number of language-processing modules (drawn as ellipses), which interface with a world model provided by a DL knowledge base (drawn as rectangles). The arrows indicate flow of information.

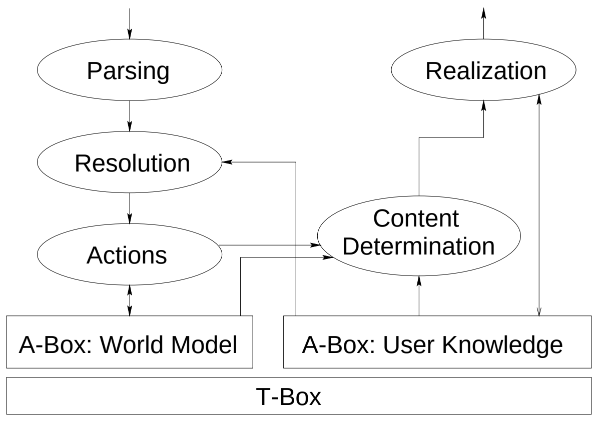

Figure 1: Architecture

The world model consists of one common T-Box defining the concepts and roles we use, as well as some useful derived concepts, such as the concept of all objects the player can see. We use two different A-Boxes: One for representing the true 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 interpreted and generated with respect to the user's knowledge –, and is updated by the realization module with the information that was verbalized and therefore conveyed to the player.

The user's input is first parsed using an efficient parser for dependency grammar (Duchier and Debusmann 2001). Next, referring expressions are resolved to individuals in the A-Boxes. The result is a ground term or a sequence of ground terms that indicates the action(s) the user wants to take. In case of syntactic ambiguities which cannot be resolved by the parser and the resolution module the alternative sequences of actions are all passed on to the module executing the actions, where unsatisfied preconditions may resolve the ambiguity.

The ground terms are then used to retrieve action descriptions from a database. Action descriptions are STRIPS-like operators defining preconditions and effects of an action. In addition, they specify how the user knowledge has to be updated when the action is performed. If the preconditions of an action are satisfied, the world model is updated according to the action's 'effects' slot, and the instantiated contents of its 'user-knowledge' slot are passed on to the content determination component, which retrieves the information that has to be verbalized from the knowledge bases. The realization component then produces an English text conveying this information to the player and updates the user model.

3 The DL Knowledge Bases

The T-Box establishes a concept taxonomy. The most general concepts are 'room', 'object', 'player', and 'property'. By requiring that the concepts 'room', 'object', and 'player' be pairwise disjoint, the individuals of the world are partitioned in three groups. (Note that the individual 'myself' is the only instance that we ever define of the concept 'player'.) T-Box axioms, such as 'toolbox $\sqsubseteq$ object', 'colour $\sqsubseteq$ property', 'silver $\sqsubseteq$ colour', specify subsumption relations between concepts. The T-Box furthermore defines roles. In particular, it defines the

room(scooter-bridge)	player(myself)
hammer(hammer1)	toolbox(toolbox1)
id-card(id1)	silver(hammer1)
closed(toolbox1)	unlocked(toolbox1)
white(id1)	
has-location(toolbox1, scooter-bridge)	has-location(myself, scooter-bridge)
has-location(hammer1, toolbox1)	has-location(id1, toolbox1)

Figure 2: A fragment of the world A-Box.

'has-location' role, which connects players and objects to exactly one location (i.e. a room, a container object, or the player). Finally, the T-Box contains definitions of more complex concepts which are essential for checking the preconditions of actions and for processing the natural language input and output. 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. As 'player' contains just one individual and 'has-location' is a feature, the concept 'here' contains precisely one individual as well.

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

Another useful concept is 'accessible', which contains all individuals which the player can manipulate. All objects in the same location as the player are accessible, and 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.

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

A concept 'visible' containing those individuals that the player can see can be defined in a similar, but more complicated way.

The two A-Boxes for modeling the current state of the game world and the user's knowledge share this T-Box. They contain information about specific individuals in the world, stating which concept they belong to and by which roles they are related to other individuals. A fragment of such an A-Box, which we will also use in the subsequent example, is shown in Fig. 2. The contents of the two A-Boxes will typically be different. For instance, the world A-Box will usually contain more individuals than the player A-Box because the player will not have explored the world completely and will therefore not have seen all the individuals. It can also 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.

Finally, we should mention that inside the action processing module, we create multiple temporary A-Boxes. They can be necessary when the user types an ambiguous sentence, such as "put the apple in the box on the table". In this case, we pursue all possible meanings of the sentence in parallel. We do this by creating copies of the current world A-Box for each reading, and then carrying out the different actions denoted by the readings on these copies. We currently clone an A-Box by resending the history of all calls made to it to a new A-Box; this is a source of inefficiency that will be discussed later.

4 Example

We now go through an example to illustrate the precise contributions of the particular modules and the queries that they typically send to the DL prover. Assume that we are in the game partially specified by the A-Box given in Fig. 2, and that the user has just unlocked the toolbox (which is still closed, however). This means he has not yet seen the contents of the toolbox; in particular, the player A-Box does not yet contain any assertions for the individuals ‘id1’, ‘hammer1’, etc. Now the user types

```
> open the toolbox
```

Parsing. First of all, the input is parsed as described in (Duchier and Debusmann 2001), producing a simple semantic representation which specifies the action that is to be executed and descriptions of the objects this action should be performed on:

```
open(patient:[toolbox(agr:[gender:[neut] number:[sing] spec:[def]])])
```

That is, we want to perform an ‘open’ action on an object which is a toolbox that was specified by a definite description (“the toolbox”). The term also includes some agreement information which is needed for the resolution of pronouns (cf. Gabsdil et al. 2001).

Resolution. Before the action can actually be executed, such object descriptions must first be resolved to individuals in the game world. The module for resolving referring expressions retrieves the instances matching this description from the player A-Box. To this end, the semantic content of the description is represented as a DL concept term (‘toolbox’). As actions can only be performed on objects that are accessible or visible, we assume that those are the objects that the player will want to talk about, and ignore all other objects. This avoids confusion of two objects that would both fit the description but are in different rooms, for example. Here, the query that is sent to RACER is

`(concept-instances` $\text{toolbox} \sqcap \text{visible}$`).`

In the example, there is only one visible individual which belongs to the concept ‘toolbox’, so resolution succeeds, and we get the output

```
open(patient:[toolbox1]).
```

In case a definite description is resolved to more that one instance the input is rejected with an error message stating that the reference cannot be resolved.

Actions. This ground term is now used to select and instantiate an action description, which then looks like this.

```
entry(action: open(patient:toolbox1)
      pre:[closed(toolbox1) unlocked(toolbox1) accessible(toolbox1)]
      effects: effects(add:[open(toolbox1)]
                       delete:[closed(1)])
      uk: uk(add:[open(toolbox1) describe(toolbox1)]
             delete:[closed(toolbox1)]))
```

It specifies that its patient must be accessible to the player and still has to be closed, but must not be locked. Checking these preconditions involves testing whether an instance belongs to a certain concept or (though it doesn't occur in this example) whether two instances are related by a certain role in the world A-Box. This can be done by `individual-instance?` and `individuals-related?` queries in RACER. In our example, all preconditions are satisfied in the world A-Box, so the actions module updates it with the information that 'toolbox1' is no longer closed, but now open. We "forget" what's specified in the delete list of the effects slot and then add what's specified in the add slot.

The following record is passed on to Content Determination:

```
open(patient:toolbox1
     failPre:nil
     succPre:[instance(toolbox1 accessible)
              instance(toolbox1 unlocked) instance(toolbox1 closed)]
     uk:uk(add:[instance(toolbox1 open) describe(toolbox1)]
           delete:[instance(toolbox1 closed)])
```

This record specifies what effect the action should have on the user knowledge, and it also records which of the preconditions were evaluated successfully and which failed. If there were failed preconditions, this would be used by the generation modules to produce informative error messages explicitly stating which are the reasons why the action cannot be carried out.

Content Determination. The next step is to determine what information should be prompted to the user, as specified by the user-knowledge ('uk') slot of the instantiated action. The elements of the 'add' slot are treated one after the other. For elements introducing concept or role assertions, such as 'instance(toolbox1 open)', the main task of content determination is to compute the information that is necessary to construct expressions referring to the individuals mentioned in these assertions. In the case of *old* individuals the content determination module just collects what the player already knows about this individual and passes it on to the realization module which constructs a suitable referring expression. New individuals are introduced to the player by giving their type and colour. The instruction 'describe(toolbox1)' is expanded into a description of the toolbox and its contents. Retrieving the necessary information about an individual from the world A-Box involves accessing its direct types (i.e. the most specific concepts it is an instance of), other individuals it is linked to via a certain role, and, in general, role assertions it is mentioned in.

The following list of records is passed on to the realization module. Each element will be realized as one sentence.

```
[content(new:new(entities:nil
                 properties:[open(toolbox1)])
         old:old(entities:[toolbox1]
                 properties:[toolbox(toolbox1) unlocked(toolbox1) open(toolbox1)])
         uk:uk(add:[instance(toolbox1 open)]
               delete:[instance(toolbox1 closed)]))
 content(new:(entities:[id1 hammer1]
              properties:[contains(toolbox1 [hammer1 id1])
                          id-card(id1) white(id1)
                          hammer(hammer1) silver(hammer1)])
         old:old(entities:[toolbox1]
                 properties:[toolbox(toolbox1) unlocked(toolbox1) open(toolbox1)])
         uk:uk(add:[related(id1 toolbox1 has-location)
                    related(hammer1 toolbox1 has-location)
                    instance(id1 id-card) instance(id1 white)
                    instance(hammer1 hammer) instance(hammer1 silver)]
               delete:nil))]
```

If there had been failed preconditions, we would have had to generate an error message explaining the failure. In such a case, the Content Determination takes the slot recording the failed preconditions (instead of 'uk') as input, adds the information that the statements in this list have to be negated, and then proceeds in the same way, assembling object descriptions as described above. If, for instance, the toolbox had not been unlocked, the content determination would return the following record, which would be realized as 'The toolbox is not unlocked'.

```
[content(new:new(entities:nil
                 properties:[not(unlocked(toolbox1))])
         old:old(entities:[toolbox1]
                 properties:[toolbox(toolbox1) locked(toolbox1)])
         uk:uk(add:nil
               delete:nil))]
```

Realization. Finally, the realization module produces a text which conveys this information and updates the player A-Box. Everything that was specified by content determination to be new information has to be expressed, while the old information is only needed to build successful references. The above information gets verbalized as follows:

```
The toolbox is open.
The toolbox contains a silver hammer and a white ID card.
```

Note that the hammer and the ID card are realized as indefinites because they are new. The next time they are mentioned, they will automatically be realized as definites ("the hammer").

Definite descriptions are built following Dale and Reiter (1995) by incrementally adding information to the noun phrase until it uniquely identifies the individual it is supposed to refer to. Whether a definite description is indeed uniquely identifying is checked using the same techniques as the resolution module for computing the referent of a referring expression.

5 Discussion

Our initial motivation for representing the world using description logic was that we had heard that the most recent DL provers (Horrocks and Patel-Schneider 1998; Haarslev and Möller 2001) were able to deal surprisingly efficiently with very expressive logics. Although computational linguists often employ logics that are even far more expressive, and this is certainly necessary for the general case, we wanted to give DL provers a try in our very restricted setting.

In hindsight, this was a very good idea. The inferences we needed were much simpler than full first-order entailment tests: Concept subsumption and in particular automatic assignment of (complex) concepts to individuals were sufficient. This made first-order provers look too powerful for our needs. In addition, we needed some simple functions for knowledge base management, such as adding and forgetting information. Such functionality, along with queries for retrieving information from the knowledge base such as `concept-instances` and `individual-direct-types`, is offered by DL provers, but not by first-order ones.

It seems that while DL is certainly not expressive enough to capture the meanings of natural language in all their complexity, the setting in an adventure game entails some powerful simplifications that make the task of the semanticist easy enough to use DL. The world model only talks about the *state* of the world, and not about any events that have a temporal aspect. In addition, the user's inputs only refer to singular individuals that are concretely "present" at the current location, and not about abstractions or properties. It may even help that the user is used to talking in very simple syntax to a computer game.

From a perspective of efficiency, most of the queries that we send to the inference system are answered very quickly; the vast majority take under 10 milliseconds, and only a handful of queries ever takes more than 100 ms. Unfortunately, the expensive queries can be *really* expensive. The primary culprit here is the realization of A-Boxes, which sometimes takes several seconds to come back even for a very small knowledge base; we can easily build bigger versions whose realization takes minutes. But this problem is largely caused by the fact that we create a new world A-Box for each action that the player wants to perform, and then explicitly copy every piece of information from the old world A-Box to the new one, which then has to be realized from scratch. We hope to gain much efficiency here by exploiting the A-Box cloning mechanism in the new version 1.6.1 of RACER. Further, we want to look into the extent to which our naive use of recursively defined concepts and inverse roles counteract RACER's optimizations. These are certainly part of the problem: The second most time is spent in answering the query `(concept-instances here)`, and the definition of 'here' is simple but involves an inverse role.

One aspect where the logic supported by RACER is not expressive enough for our needs is the construction of concepts from individuals. The $\mathcal{SHOQ}(\mathbf{D})$ logic (Horrocks and Sattler 2001), for instance, admits *nominals*, concept terms $\{o\}$ whose denotation contains precisely the one individual o. The absence of such terms in our application forces us into workarounds in several places. For example, some of the action effects could be stated more naturally by simply *defining* the concept 'player' as containing precisely 'myself'. As it is now, some action definitions contain subqueries for `concept-instances`, which hurts the efficiency. On the other hand, we of course acknowledge that the combination of nominals and inverse roles is even more explosive from a computational point of view.

6 Conclusion

In this paper, we have discussed a text adventure engine that employs techniques from computational linguistics to deal with natural-language input and output. The world is modeled as a description logic knowledge base, and we make use of DL inference services throughout the system. As the language-processing modules have to reason about the state of knowledge of the player, we have to use two different A-Boxes, one of which describes the true state of the world, while the other describes the player's knowledge. We have gone through an example in some detail to illustrate the queries we actually send to the inference engine, and have discussed the suitability of today's DL provers for our application.

The current system does allow the interaction with a small game world. However, the language modules are still very simplistic and the theorem-proving task becomes quite inefficient for more complex worlds. We plan to address the second problem by a careful investigation of some aspects of our T-Box that counteract the prover's optimizations, e.g. the use of inverse roles. As for the first problem, we plan to put a cleaned-up version of the system on the Web and invite colleagues and students to contribute their own (more sophisticated) modules to replace ours. As part of this process, we want to develop a clear specification of the modules and the interfaces between them and provide a mechanism which allows users of other programming languages to plug in their modules as well.

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 and background knowledge about DL. 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

Dale, R. and E. Reiter (1995). Computational Interpretations of the Gricean Maxims in the Generation of Referring Expressions. *Cognitive Science 18*, 233–263.

Duchier, D. and R. Debusmann (2001). Topological Dependency Trees: A Constraint-based Account of Linear Precedence. In *Proceedings of the 39th ACL*, Toulouse, France.

Gabsdil, M., A. Koller, and K. Striegnitz (2001). Building a Text Adventure on Description Logic. In *Proceedings of KI-2001 Workshop on Applications of Description Logics*, Vienna. Available at `http://www.coli.uni-sb.de/~koller/papers/dlws01.html`.

Haarslev, V. and R. Möller (2001). RACER System Description. In *Proceedings of IJCAR-01*, Siena.

Horrocks, I. and P. F. Patel-Schneider (1998). Optimising Propositional Modal Satisfiability for Description Logic Subsumption. In J. Calmet and J. Plaza (Eds.), *Artificial Intelligence and Symbolic Computation: International Conference AISC'98*, Number 1476 in Lecture Notes in Artificial Intelligence, pp. 234–246. Springer-Verlag.

Horrocks, I. and U. Sattler (2001). Ontology Reasoning in the $\mathcal{SHOQ}(\mathbf{D})$ Description Logic. In B. Nebel (Ed.), *Proc. of the 17th Int. Joint Conf. on Artificial Intelligence (IJCAI-01)*, pp. 199–204. Morgan Kaufmann.

Mozart Consortium (1999). The Mozart Programming System web pages. `http://www.mozart-oz.org/`.

Striegnitz, K. (2001). Model Checking for Contextual Reasoning in NLG. In P. Blackburn and M. Kohlhase (Eds.), *Proceedings of ICoS-3*, Siena.