Put my galakmid coin into the dispenser and kick it: Computational Linguistics and Theorem Proving in a Computer Game

ALEXANDER KOLLER

Saarland University, Saarbrücken, Germany.
E-mail: koller@coli.uni-sb.de

RALPH DEBUSMANN

Saarland University, Saarbrücken, Germany.
E-mail: rade@ps.uni-sb.de

MALTE GABSDIL

Saarland University, Saarbrücken, Germany.
E-mail: gabsdil@coli.uni-sb.de

KRISTINA STRIEGNITZ

Saarland University, Saarbrücken, Germany.
E-mail: kris@coli.uni-sb.de

ABSTRACT: We combine state-of-the-art techniques from computational linguistics and theorem proving to build an engine for playing text adventures, computer games with which the player interacts purely through natural language. The system employs a parser for dependency grammar and a generation system based on TAG, and has a component for resolving referring expressions. Most of these modules make heavy use of inferences offered by a modern theorem prover for description logic. Our game engine solves some problems inherent in classical text adventures, and is an interesting test case for the interaction between natural language processing and inference.

KEYWORDS: text adventures, description logic, theorem provers, parsing, generation, reference resolution, dependency grammar

1 Introduction

Text adventures are a form of computer games that was very 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.

```
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.
```

FIG. 1. An example interaction with a text adventure, taken from [14].

In this paper, we combine state-of-the-art techniques from computational linguistics and theorem proving to implement a text-adventure engine. We use Description Logic (DL) to represent the state of the game world and what the player knows about it. A DL reasoning system is used to update, maintain, and query these knowledge bases. This reasoning system is used heavily throughout the game. In particular, the modules for the resolution of referring expressions, for executing actions, and for generation access the inference engine; only the modules for parsing and surface realization don't.

Our system is interesting from the perspective of the game player, from the perspective of computational linguistics, and from the perspective of DL theorem proving. The first motivation is maybe most obvious. Classical text adventures typically produced very high quality texts (which were hard-coded into the games), but the parsers they employed, while usable, were still very simple. This led to some irritating limitations, for example the *identification problem*: Sometimes the games would 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 [6]). By closing the gap between the quality of the parser and the generator, our system allows in principle to avoid this sort of problem.

From the perspective of computational linguistics, the computer game setting is

```
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.
```

FIG. 2. The identification problem. The user plays a teddybear in this game.

interesting because it naturally restricts what utterances the user will produce. For example, players will typically only refer to objects they can "see" in the simulated world. This simplifies the language processing tasks, constrains the way in which ambiguous inputs are to be interpreted, and allows the inference problems to scale rather well to larger game worlds. There is a natural notion of a context (both with respect to what has been said before and in the sense of being situated in the game world), and the world can be freely specified and tailored to different levels of complexity. The system can serve as a testbed for more advanced NLP modules, which have automatic access to the inference infrastructure. It is also attractive as a teaching environment. In fact, our current implementation was created in an advanced programming project for students. This is why the individual NLP modules described in this paper are all necessarily quite simplistic.

Finally, our system is interesting from the theorem proving perspective because it is an application that makes heavy use of A-Box reasoning. Traditionally, applications of description logic have focused on T-Box inferences such as concept subsumption. Our system, in contrast, needs to enumerate instances of concepts, concepts to which an instance belongs, etc. What's more, our system needs to deal with A-Boxes that have to be changed in every turn of the game. Modern DL systems support this sort of inference, which allows us to store the world model entirely in the DL knowledge base and retrieve relevant information by querying the inference system. The demands of our system towards the theorem prover have already motivated optimizations of the

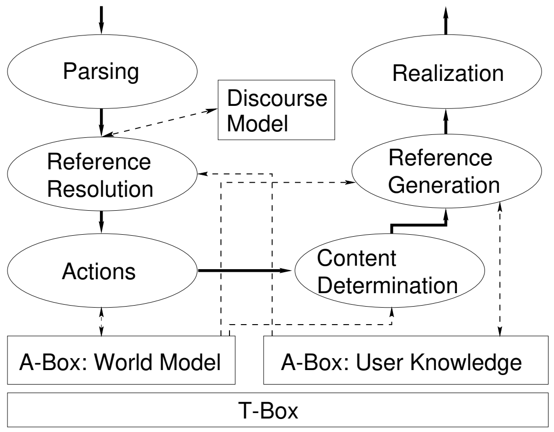

FIG. 3. The Architecture.

prover we use.

The paper is organized as follows: Section 2 sketches the general architecture of our systems and its components. In the following sections (Sections 3 – 7), we describe each of the system's components in more detail. In Section 8, we take a look at the performance of the DL theorem prover in the game setting, and Section 9 concludes the paper.

2 Architecture

The general architecture of the game engine is shown in Fig. 3. Underlying the system are two description logic knowledge bases, which share a set of common definitions: One represents the true state of the world and the other keeps track of what the player knows about the world. These knowledge bases are accessed by all language processing modules (drawn as ellipses) except for parsing and realization, as is indicated by the dashed arrows. The solid arrows show the flow of information when processing a player input and generating a response.

The user's input is first parsed. This yields a semantic representation specifying the action that the user wants to execute and describing the objects that this action involves. Next, these object descriptions are resolved to individuals of the game world, based on the knowledge that the player has about the world and on the discourse model, which keeps track of when and how individuals were mentioned in the previous dialog. The result is a ground term or a sequence of ground terms that indicates the action(s) the user wants to take. The Actions module looks up these actions in a database (where they are specified in a STRIPS-like format), checks whether the action's preconditions are met in the world, and, if yes, updates the world state with the effects of the action.

The action can also specify effects on the user's knowledge, i.e. information that should be conveyed to the hearer through a natural language text. The generation component, which produces this text, consists of three modules: The Content Determination module further enriches the information that is specified as effects on the user knowledge in the action description; for example, this module chooses which information to include in detailed descriptions of objects the player wants to look at. The Reference Generation module translates the internal names of individuals into descriptions that can be verbalized. In the last step, this assembled information is realized as a natural language text. The player knowledge is updated after Reference Generation because some information which is new to the user may be added by this module, as e.g. in the case of indefinite NPs. The same generation modules are also used to generate error messages.

The system is implemented in the programming language Mozart [15] and communicates with the DL reasoning system RACER [10] to access the knowledge bases.

In the following sections, we will describe each of the game's components in more detail. In Section 3, we give a brief introduction to Description Logic and describe the different knowledge bases we need to model the game world. We then discuss how the player input is analyzed, describing the modules for parsing (Section 4) and resolution of referring expressions (Section 5). Next, we show how actions are performed in the game world (Section 6) and finally, we describe how the output texts informing the player about the changing state of the game world are generated (Section 7).

3 The World Model

Before we go into the details of the language-processing modules, we will first explain how we model the world and the user knowledge. We start with an introduction to description logic (DL), and then show a fragment of an actual knowledge base used in a game.

3.1 Description Logic

Description logic (DL) is a family of logics in the tradition of knowledge representation formalisms such as KL-ONE [17]. DL is a fragment of first-order logic which only allows unary and binary predicates (called *concepts* and *roles* in this context), Boolean connectives, and only very restricted quantification. Correspondingly, the syntactic objects it is concerned with are *concept terms*, *role terms*, and *constants*. Concept terms denote sets of individuals, role terms denote binary relations, and constants denote individuals; they are defined as in Figure 4.

A knowledge base consists of a *T-Box*, which contains axioms relating the concepts and roles, and an *A-Boxes*, which states that individuals belong to certain concepts, or are related by certain roles. The axioms in a T-Box typically have either the form

concept terms:

C	an *atomic concept*, denotes a set of individuals
$C \sqcup C'$	a *disjunction* denotes the set union of C and C'
$C \sqcap C'$	a *conjunction* denotes the intersection of C and C'
$\neg C$	all individuals that are not in C
$\exists R.C$	the concept containing all individuals that are connected via R to an individual in C
$\forall R.C$	the concept containing all individuals such that every individual to which they are related through R is in C
$\bot$	the empty concept, containing no individuals

role terms:

R	an *atomic role*, denotes a binary relation
R^{-1}	the *inverse role*, denotes the inverse relation of R

FIG. 4. DL concept and role terms.

$C \sqsubseteq C'$, stating that C denotes a subset of C', or $C \doteq C'$, expressing that the denotations of C and C' are equal. A-Box axioms are of the form $C(a)$ and $R(a, b)$.

Theorem provers for description logics support a range of different reasoning tasks. Among the most common are *consistency* checking, *subsumption* checking, and *instance* and *relation* checking. Consistency checks decide whether a combination of T-Box and A-Box can be satisfied by some model, subsumption is to decide of two concepts whether all individuals that belong to one concept must necessarily belong to another, and instance and relation checking test whether an individual belongs to a certain concept and whether a certain relation holds between a pair of individuals, respectively. In addition to these basic reasoning tasks, description logic systems usually also provide some *retrieval functionality* which e.g. allows to compute all concepts that a given individual belongs to, or all individuals that belong to a given concept.

There is a wide range of different description logics today which add different extensions to a common core. Of course, the more expressive these extensions become, the more complex the reasoning problems are. "Traditional" DL systems have concentrated on very weak logics with simple reasoning tasks. In the last few years, however, new systems such as FaCT [12] and RACER [10] have shown that it is possible to achieve surprisingly good average-case performance for very expressive (but still decidable) logics. In this paper, we employ the RACER system, mainly because it allows for A-Box inferences. We will state the DL queries in quite general terms throughout this paper; more technical details can be found in [8].

3.2 The World Model

The T-Boxes we use games specify the concepts and roles in the world and defines some useful complex concepts, e.g. the concept of all objects the player can see. Such a T-Box is shared by two different A-Boxes representing the state of the world and what the player knows about it respectively.

The player A-Box will typically be a sub-part of the game A-Box because the player will not have explored the world completely and will therefore not have seen all the individuals or know about all of their properties. Sometimes, however, it may 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. In this case, the player A-Box can contain information that is inconsistent with the world A-Box.

A fragment of an example A-Box describing a state of the world is shown in Fig. 5; Fig. 6 gives a graphical representation.

room(kitchen)	player(myself)
table(t1)	apple(a1)
apple(a2)	worm(w1)
red(a1)	green(a2)
bowl(b1)	bowl(b2)
has-location(t1, kitchen)	has-location(b1, t1)
has-location(b2, kitchen)	has-location(a1, b2)
has-location(a2, kitchen)	has-detail(a2,w1)
has-location(myself, kitchen)	...

FIG. 5. A fragment of a world A-Box.

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, container objects, or players) via the 'has-location' role; the A-Box also specifies what kind of object an individual is (e.g. 'apple') and what properties it has ('red'). The T-Box then contains axioms such as 'apple $\sqsubseteq$ object', 'red $\sqsubseteq$ 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.

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

In the example in Fig. 5, 'here' denotes the singleton set $\{\text{kitchen}\}$: It is the only individual to which an instance of 'player' is related via the role 'has-location'.

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

$$\begin{aligned}\text{accessible} \;\doteq\; & \forall \text{has-location.here} \sqcup \\ & \forall \text{has-location.}(\text{accessible} \sqcap \text{open})\end{aligned}$$

All objects in the same room as the player are accessible; if such an object is an open

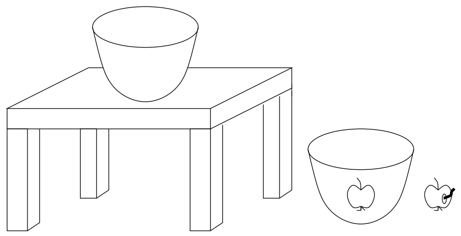

FIG. 6. Graphical representation of the A-Box fragment.

container, its contents are also accessible. The T-Box contains axioms that express that all instances of certain concepts (e.g. ‘table’, ‘bowl’, and ‘player’) are always ‘open’. This permits access to the player’s inventory. In the simple above scenario above, ‘accessible’ denotes the set $\{$myself, t1, a1, a2, b1, b2$\}$. Finally, we can define the concept ‘visible’ 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.

4 Parsing

In this and the next sections, we will now go through the modules that were shown in Fig. 3 in more detail. First of all, we discuss the parsing module.

The parsing module uses a parser for Topological Dependency Grammar (TDG) to perform the syntactic analysis. TDG is a rather new formalism based on dependency grammar. The formalism and some underlying linguistic theory are described in [3, 4]; an implementation of the parser as a constraint program is available freely on the web, and is described in [5].

The output of the TDG parser is a *syntactic dependency tree* and a *topological dependency tree*. The interesting structure for us is the syntactic dependency tree, which represents the syntactic analysis of the sentence. We do not talk about the topological tree here, which is used to constrain word order. From the syntactic dependency tree, we compute the desired *semantic representation* of the input sentence. This semantic representation is passed on to the later stages of processing; the dependency tree is discarded.

We begin with explaining the notion of a syntactic dependency tree, and then how to transform it into a semantic representation. For this transformation of the syntactic dependency tree to the semantic representation, we have developed a syntax-semantics interface for TDG. This is in fact the first syntax-semantics interface for the TDG grammar formalism and has been developed specifically for the game engine.

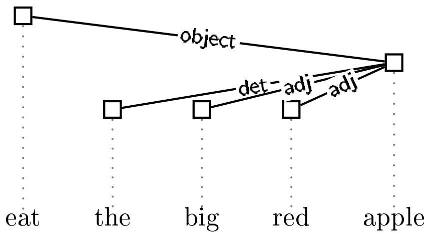

FIG. 7. A syntactic dependency tree.

4.1 Syntactic Dependency Trees

Fig. 7 shows an example of a syntactic dependency tree. As is characteristic for dependency trees, the nodes of the tree (boxes) are associated with words of the input sentence (dotted lines) and the edges are labeled with syntactic relations. Essentially, an edge from node w to node w' labeled ρ expresses that w' is a ρ-dependent of w. In Fig. 7, for example, *apple* is the object(-dependent) of *eat*, and *big* and *red* are adjective-dependents of *apple*.

The lexicon assigns to each word a set of lexical entries. In a dependency tree, one such entry must be picked for each node. The lexical entry specifies constraints on the incoming and outgoing edges of the node. Fig. 8 shows some examples of lexical entries for a grammar that accepts imperative sentences.[1] The lexical entry for *eat* specifies that this node must not have any incoming edges (i.e. it must be the root of the tree), that it must have exactly one object (marked with !), and may have at most one subject (for sentences like "John, eat the apple!" – marked with ?). The entry for *apple* says that it can have either a subj or an obj role coming into it; it does not require any outgoing edges, but allows one determiner and arbitrarily many adjectives (marked with $*$). *The*, *big*, and *red* can fill these roles, and do not allow any outgoing edges.

A syntactic dependency tree is well-formed if it is a tree and satisfies all the constraints on incoming and outgoing edges specified by the lexical entries. Given the lexicon in Fig. 8, the tree in Fig. 7 is a well-formed syntactic dependency tree for the imperative sentence *Eat the big red apple*.

4.2 Semantic Dependency Trees

Given a lexicon like the one in Fig. 8, the TDG parser computes a syntactic dependency tree for the input sentence. To obtain a semantic representation for the sentence, we transform this syntactic tree into a *semantic dependency tree*. Semantic dependency trees have edge labels corresponding to thematic roles (e.g. agent and patient)

[1] Notice that these lexical entries only specify syntactic information. Information concerning word order (topological dependency tree) and the syntax-semantics-interface is left out for clarity.

$$
\begin{aligned}
eat &= \begin{bmatrix} \mathsf{in} & : & \{\} \\ \mathsf{out} & : & \{(\mathsf{subj},?),(\mathsf{obj},!)\} \end{bmatrix} \\
the &= \begin{bmatrix} \mathsf{in} & : & \{\mathsf{det}\} \\ \mathsf{out} & : & \{\} \end{bmatrix} \\
big, red &= \begin{bmatrix} \mathsf{in} & : & \{\mathsf{adj}\} \\ \mathsf{out} & : & \{\} \end{bmatrix} \\
apple &= \begin{bmatrix} \mathsf{in} & : & \{\mathsf{subj},\mathsf{obj}\} \\ \mathsf{out} & : & \{(\mathsf{det},?),(\mathsf{adj},*)\} \end{bmatrix}
\end{aligned}
$$

FIG. 8. An Example Lexicon.

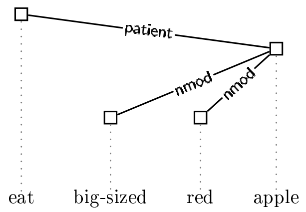

FIG. 9. A semantic dependency tree.

and a subset of the nodes of the syntactic tree. As an example, Fig. 9 shows the semantic dependency tree corresponding to the syntactic dependency tree of Fig. 7.

The relation between the syntactic and semantic tree is specified in semantics features of the lexical entries. For instance, in addition to the features specified in Fig. 8, the entry for *apple* contains the following features:

$$
apple = \begin{bmatrix} \mathsf{sem} & : & \text{‘apple’} \\ \mathsf{nmod} & : & \{\mathsf{adj}\} \end{bmatrix}
$$

This means that whenever the node corresponding to the word *apple* in the syntactic tree has an outgoing adj edge, the corresponding node in the semantic tree has an equivalent outgoing nmod edge (for "noun modification").

We compute the semantic dependency tree by going top-down through the syntactic tree and mapping syntactic to semantic roles. Starting at the root, we map the object role going out of the *eat* node to a patient role, make a new node for *apple* in the semantic tree, and proceed there.

Now *apple* in the syntactic tree has three outgoing edges, two of which are adj edges and are thus mapped to nmod edges in the semantic tree according to the above lexical entry. However, the edge with label det isn't mapped to anything. This means the node for *the* in the syntactic tree gets lost in the transformation.

For the purposes of reference resolution, we record some further information whenever we hit a noun node, namely agreement, linear position within the sentence, and (in)definiteness. The agreement information annotated to each noun is the unification of the agreement features of the determiner, the adjectives and the noun; and we take a noun to be (in)definite if it has a child over a det edge in the syntactic tree that leads to an (in)definite determiner.

The end product of semantics construction is then the following enriched semantic tree corresponding to the syntactic tree in Fig. 7 (given in record notation):

```
eat(patient:[apple(agr:[unit(gender:[neut]
                              number:[sing]
                              spec:[def])]
                   nmod:['big-sized' red]
                   pos:[5])])
```

5 Resolution of Referring Expressions

Records like this are valid semantic representations of the player's input, but still refer to individuals in the world through formalized versions of the NPs in the input sentence. The next step is now to map these representations to constants in the knowledge base, which can be used internally.

The resolution module is responsible for relating the user input to the game world by mapping definite and indefinite noun phrases and pronouns to individuals in the description logic knowledge bases. We make use of RACER's inference system to retrieve individuals that match the player's descriptions and employ a simple discourse model which keeps track of available referents to resolve pronouns and ambiguous definite NPs.

5.1 Definite and Indefinite Descriptions

The resolution of definite and indefinite descriptions is simplified in the adventure setting by the fact that the communication is situated in a sense: Players will typically only refer to objects which they can "see" in the virtual environment, as modeled by the concept 'visible' above. Furthermore, they won't try to refer to objects they haven't seen yet. We can therefore perform all RACER queries needed for resolution on the player knowledge A-Box, avoiding unintended ambiguities when the player's expression would for example not refer uniquely with respect to the true state of the world.

The resolution of a *definite* description means to find a unique entity which, according to the player's knowledge, is visible and matches the description. To compute such an entity, we construct a DL concept expression corresponding to the description and then send a query to RACER asking for all instances of this concept. In the case of *the apple*, for instance, we would retrieve all instances of the concept

$$\text{apple} \sqcap \text{visible}$$

from the player A-Box. 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 work for us. For example, *the apple with the worm* translates to the query

$$\text{apple} \sqcap (\exists\text{has-detail.worm}) \sqcap \text{visible}$$

If such a query yields only one entity ($\{\text{a2}\}$ for *the apple with the worm* for the A-Box in Fig. 5), the reference has been unambiguous and succeeds. It may, however, also be the case that more than one entity is returned. For instance, the query for *the apple* would return the set $\{\text{a1,a2}\}$. In such a situation, we try to filter out all potential referents which are completely unsalient according to our discourse model (see below). If this narrows down the candidate set to one we are done. Otherwise we assume that the definite description wasn't unique, and return an error message to the user, indicating the ambiguity.

We resolve *indefinite* NPs, such as *an apple*, by querying the player knowledge in the same way as described above for definites. Unlike in the definite case, however, we do not require unique reference. Instead we assume that the player did not have a particular object in mind and arbitrarily choose one of the possible referents. The reply of the game will automatically inform the player which one was chosen, as a unique definite reference will be generated (see Section 7).

5.2 *Pronouns and the Discourse Model*

To resolve pronouns we make use of a discourse model (DM) inspired by Strube's [16] salience list approach, due to its simplicity. The DM is a data structure that stores an ordered list of the most salient discourse entities according to their "information status" and text position and provides methods for retrieving and inserting elements. Following Strube, *hearer-old* discourse entities (which include definites) are ranked higher in the DM (i.e. are more available for reference) than *hearer-new* discourse entities (including indefinites). Within these categories, elements are sorted according to their position in the currently processed sentence. For example, the ranking of discourse entities for the sentence *take a banana, the red apple, and the green apple* would look as follows:

$$[red\ apple \prec green\ apple]_{old} \prec [banana]_{new}$$

The DM is built incrementally and updated after each input sentence. Updating removes all discourse entities from the DM which are not realized in the current utterance. That is, there is an assumption that referents mentioned in the previous utterance are much more salient than older ones.

Given the current DM, pronouns are simply resolved to the most salient entity that matches their agreement constraints. The restrictions our grammar imposes on the player input (no embeddings, no reflexive pronouns) allow us to analyze sentences including intra-sentential anaphora like *take the apple and eat it*. The incremental construction of the DM ensures that by the time we encounter the pronoun *it*, *the apple* has already been processed and can serve as a possible antecedent.

6 Performing Actions

The output of the resolution module is a list of lists of instantiated action descriptions – in the example above, `[[take(patient:a2)]]`. The outer list contains one entry for each reading of a (syntactically or referentially) ambiguous input sentence. These entries are themselves lists, which represent sequencing of actions which are to be performed one after another ("take the apple *and* eat it").

We will now explain how these actions are performed. We shall first look at how a single action is executed, and then we shall explain how an ambiguous list of action sequences is interpreted.

6.1 Performing a single action

An instantiated description of a single action, such as `take(patient:a2)`, is first of all matched against a list of action representations in a database. Such action representations are STRIPS-like operators [7] that specify an action's preconditions and effects, as in the following example:

`take(patient:X)`	
preconditions	`accessible(X), takeable(X),` `not(inventory-object(X))`
effects	add: `related(X myself has-location)` delete: `related(X indiv-filler(X has-location) has-location)`
user knowledge	add: `related(X myself has-location)` delete: `related(X indiv-filler(X has-location) has-location)`

The term X in the action representation is a variable that gets bound to the actual argument that the resolution module computed. In the example, X would be bound to the constant `a2`, and thus the preconditions and effects of the operator will become ground terms as well.

An instantiated action representation is applicable if all preconditions are satisfied by the current world A-Box. These preconditions assert that individuals belong to certain concepts, or that they are linked by certain roles. In the example, we require that we can actually touch the object, that it is small enough to be picked up, and that we don't carry it already. These questions can be answered by queries to RACER.

If the preconditions are satisfied, the world A-Box is updated. First the atoms in the "delete" branch of the "effects" slot are removed from the A-Box, and then the atoms in the "add" branch are added to it.

Most interesting actions cannot be specified completely in advance as they depend on the current state of the game. In these cases, we embed special terms of the forms `individual-filler(X R)` and `concept-instance(C)` in the action specification. These terms trigger further RACER queries which provide the information for fully specifying the effects of the action. The first term evaluates to the unique individual to which `X` is connected via the role `R`; the second term evaluates to the unique individual denoted by the concept `C`. The uniqueness assumptions require careful synchronization of the world model and the actions; if they are violated, the action fails. In the example, we first delete the information that `X` is in its original location, and then we add that it's in our inventory.

Finally, the "user-knowledge" slot represents the information that should be communicated to the user to indicate the action has been performed successfully, and what has changed in the world as a result. It is passed on to the generation module, where it serves as the input for the content determination component.

6.2 Ambiguity and Sequences

If the input sentence was ambiguous (either syntactically or referentially), the actions module tries to decide which of the readings the user intended by trying each action sequence in parallel. If only one possible sequence succeeds, it assumes that this is the command the player had in mind, and commits to the end result of this sequence. If more than one sequence is possible, it reports a true ambiguity; if none is, it outputs an error message.

The first step towards the parallel tests is to instruct the theorem prover to create an identical copy of the current world A-Box for each reading. Then the actions in each reading are performed in sequence on its own copy. As long as the single actions in each sequence succeed, the effects of an action are first incorporated into the A-Box, and the preconditions of the next action in the sequence are evaluated with respect to the updated A-Box. When an action fails, the entire reading it belongs to is discarded.

7 Generation

The task of the generation component is to produce texts in reaction to the user input to let the user 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 slot of the action we just performed, and it computes textual output in three steps, which we will discuss now.

7.1 Content Determination

In general, Content Determination is the task to decide *what* to say. In our setting, this is very easy in some cases, as we can simply extract the information from the user knowledge slot. More precisely, we verbalize just the "add" branch, assuming that the player can infer the "delete" information from the positive branch.

In the above case of the *take the apple* action, this branch contains the following list, which can simply be passed on to the next module without change:

```
[has-location(a2 myself)]
```

However, there are actions for which we do have to do some work here. In particular, the user knowledge slots of some actions contain literals of the form `describe(X)`. Such literals are interpreted as an instruction to Content Determination to generate a detailed description of the individual `X`. They are useful for actions like *look*, which have no effects on the world at all and only update the player's knowledge, as well as for actions like *move*, which move the player into a new room and should describe the new location. The user knowledge slot of an instantiated *look* action might thus look as follows:

```
add: [describe(a2)]
delete: nil
```

The Content Determination module now replaces the `describe` literal by a list of properties that `a2` has. It queries RACER to return all most specific concepts that `a2` belongs to in the world A-Box, as well as all role assertions in which `a2` participates. It will then group this information into different sentences, one for each concept and role; if `a2` is connected to more than one individual through the same role, these target individuals are aggregated into a plural list. The result for the running example would be as follows:

```
[content(goal:l1
         sem: [l1#apple(a2) green(a2)])
 content(goal:l2
         sem: [l2#has-location(a2 kitchen)])
 content(goal:l3
         sem: [l3#has-detail(a2 w1)])]
```

The `goal` labels indicate the main message of each sentence and is needed by the realization module.

7.2 Reference Generation

The output of Content Determination specifies what information we want to communication to the player. Unfortunately, it refers to individuals with names like `a2` and `w1`, which are meaningful to the theorem prover, but not to the player. The task of the Reference Generation module is to generate natural-language NPs that refer to these individuals. It enriches the semantics lists that are passed from the Content Determination by some further literals, which then form the input for the Realization component below.

The reference generation task is quite simple 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, we generate an indefinite NP containing the type and (if it has one) colour of the object, as in *the bowl contains a red apple*. We use RACER's retrieval functionality to extract this information from the world A-Box.

To refer to an object that the player already has encountered, we construct a definite description that, given the player knowledge, uniquely identifies this object. For this purpose we use a variant of Dale and Reiter's [2] incremental algorithm, extended to deal with relations between objects [1]. The properties of the target referent are looked at in some predefined order (e.g. first its type, then its colour, its location, parts it may have ...). 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, we use RACER queries to the player A-Box 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. The last message of the running example would e.g. be enriched as follows, if the player knows about both apples, but not about the worm:

```
[content(goal:l1
         sem:[l1#has-detail(a2,w1), indef(w1), worm(w1),
              def(a2), apple(a2), green(a2)])]
```

The message now contains the information that an indefinite reference to `w1` should be built, referring to it as *a worm*. `a2` should be referred to by the definite description *the green apple*. The colour was added to distinguish it from the other apple, `a1`, which is red.

7.3 Realization

Now all the information that has to be expressed is assembled and has to be cast into a text. This is done sentence by sentence, using a tree-adjoining grammar. Each elementary tree in the grammar is associated with a non-empty list of semantic atoms, which can be matched to a part of the semantics we want to verbalize. The goal is to select trees that completely cover the input semantics and can be composed into a valid TAG derivation.

To solve this problem, we use the surface realization system described in [13]. This system transforms the problem of verbalizing a semantics according to a TAG grammar into the problem of parsing a sentence according to an (abstract) dependency grammar. It then uses the same parser that we use for parsing the user input (see Section 4) for realization. Although realization is still an NP-complete problem [13], the realization component performs rather well in the game engine.

8 Performance

All the mechanisms we have laid out so far would be not very useful for the task of actually playing a game if they were so inefficient that the system's response time was more than a second or two. Fortunately, it turns out that it works quite efficiently.

The first surprise in this respect is in the parsing and realization components, both of which solve NP-complete problems. But the constraint programming techniques in the dependency parser seem to be very good at keeping the average complexity down, and indeed both components perform in polynomial time with the grammars and inputs we use [13].

Even more interesting, however, is the performance of the inference system. The inefficiency of theorem proving systems is one of the main bottlenecks of using inference in computational linguistics, and indeed RACER solves EXPTIME-complete problems.

In our application, it turns out that this is no problem for the most part. On the knowledge base we've worked with mostly, the vast majority of queries returns after less than 10 milliseconds, and only a handful of queries take more than 100 ms. In earlier versions of RACER, there were some queries that took several seconds to compute. Since then, RACER has been optimized for faster A-Box reasoning – in part specifically to improve the performance of our game engine [11]. For example, a new mechanism for cloning A-Boxes has been added to the prover. The slowest queries now return after about 500 milliseconds, and the average time spent on queries in each turn is 546ms with 39 queries per turn on average.

On the modeling side, description logic of course affords us much less expressive power than e.g. first-order logic, and it seems rather improbable that one could use it to capture the meanings of natural language in all their complexity. However, the adventure game setting entails some powerful simplifications that make the use of DL suitable. To give an example, the world model currently only talks about the *state* of the world. Events are not represented within it and, consequently, the player cannot talk about previous actions, as in the referring expression *the apple that I just picked up* for example. This doesn't seem to be an unnatural restriction, though.

Furthermore, we can work around some of the limitations in expressivity by splitting a reasoning task that cannot be formulated directly as a RACER query into a sequence of queries using the answers to retrieval commands in the formulation of the following query. This has proved to be an extremely powerful technique in the game context.

One further simplification that greatly helps the system to scale to larger knowledge bases is that the user's inputs only refer to individuals that are currently "present" at the player's location. This means that we generally have a natural restriction on the concept whose instances we need to enumerate.

9 Conclusions and Future Work

We have described an engine for text adventures which is based on techniques from computational linguistics and theorem proving. The input is analyzed using a dependency parser and a simple reference resolution module, and the output is produced by a small generation system. Information about the world and about the player's knowledge is represented in Description Logic knowledge bases, which are accessed through the RACER inference system.

All language-processing components (except for the parser and the surface realization module) use the inference infrastructure. The majority of queries to the inference system are A-Box queries that are concerned with the extensions of concepts. This is a challenge for the theorem prover because efficient A-Box reasoning is a comparatively new development, and some optimizations are not available because our A-Boxes change after each turn. However, our experience so far is that the performance offered by RACER is good enough for fluent gameplay. It remains to be seen how well the performance scales to bigger game worlds, but as queries are generally restricted to the "location" of the player, we are quite optimistic. The lesson we take from this is that the recent progress in optimizing inference engines for expressive description logics is beginning to make them useful for applications in natural-language processing.

The language-processing modules that we have implemented so far are all rather simplistic. We can get away with this, again because the situatedness of the player in a virtual location naturally restricts the player's utterances. (The precise extent of this, of course, remains to be evaluated.) With respect to the processing of the player's input, our system exceeds traditional text adventures by far. In particular, the focus on the player's location and surroundings can be exploited to resolve ambiguities, which we have shown in more detail elsewhere [9].

Unlike the input, the output that our game generates is far away from the quality of the commercial text adventures of the eighties, which produced canned texts. A possible solution could be to combine the full generation with a template based approach, to which the TAG-based generation approach we take lends itself well. The identification problem, which we have quoted as one of the main annoyances in classical games, is easy to resolve in practice so far. An interesting question for the future is whether the grammars for parsing and generation can be synchronized in order to *guarantee* that it does not occur.

More generally, we believe that the prototype we have described can serve as a starting point for an almost unlimited range of interesting extensions, ranging from adding a speech recognition and synthesis system to the integration of a planner to perform trivial tasks for the player. At the same time, it should be possible to replace the modules one by one by more sophisticated ones doing the same tasks. We plan to make our system available over the web shortly, and invite everybody to contribute.

References

[1] R. Dale and N. Haddock. Generating referring expressions involving relations. In *Proceedings of the 5th EACL*, 1991.

[2] R. Dale and E. Reiter. Computational interpretations of the gricean maxims in the generation of referring expressions. *Cognitive Science*, 18, 1995.

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

[4] Denys Duchier. Lexicalized syntax and topology for non-projective dependency grammar. In *Eighth Meeting on Mathematics of Language*, Helsinki/FIN, 2001.

[5] Denys Duchier. Configuration of labeled trees under lexicalized constraints and principles. To appear in the Journal of Language and Computation, 2002.

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

[7] Richard E. Fikes, Peter E. Hart, and Nils J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3:251–288, 1972.

[8] M. Gabsdil, A. Koller, and K. Striegnitz. Playing with description logic. In *Proceedings of the Second Workshop on Methods for Modalities (Application Description)*, Amsterdam, 2001.

[9] Malte Gabsdil, Alexander Koller, and Kristina Striegnitz. Natural language and inference in a computer game. In *Proceedings of COLING*, Taipei, 2002. To appear.

[10] V. Haarslev and R. Möller. RACER System Description. In *Proceedings of IJCAR-01*, 2001.

[11] Volker Haarslev and Ralf Möller. Optimization strategies for instance retrieval. In *Proc. of the International Workshop on Description Logics*, Toulouse, France, 2002.

[12] I. Horrocks, U. Sattler, and S. Tobies. Practical reasoning for expressive description logics. In H. Ganzinger, D. McAllester, and A. Voronkov, editors, *Proceedings of LPAR'99*, number 1705 in LNAI. Springer-Verlag, 1999.

[13] Alexander Koller and Kristina Striegnitz. Generation as dependency parsing. In *Proceedings of ACL-02*, Philadelphia, 2002. To appear.

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

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

[16] M. Strube. Never Look Back: An Alternative to Centering. In *COLING-ACL*, 1998.

[17] W. Woods and J. Schmolze. The KL-ONE Family. *Computer and Mathematics with Applications*, 23(2–5), 1992.

Received 4 June 2002