I2-D9

Attempto Controlled English 5: Language Extensions and Tools I

Project title:	Reasoning on the Web with Rules and Semantics
Project acronym:	REWERSE
Project number:	IST-2004-506779
Project instrument:	EU FP6 Network of Excellence (NoE)
Project thematic priority:	Priority 2: Information Society Technologies (IST)
Document type:	D (deliverable)
Nature of document:	R/P (report and prototype)
Dissemination level:	PU (public)
Document number:	IST506779/Zurich/I2D9/D/PU
Responsible editor:	Norbert E. Fuchs
Reviewers:	Lars Kirchhoff (I2), Gerd Wagner (I1)
Contributing participants:	University of Zurich
Contributing workpackages:	I2
Contractual date of delivery:	3 September 2006
Actual date of delivery:	31 August 2006

Abstract

This report presents four tracks of research on Attempto Controlled English (ACE). First, we present the new features of version 5 of ACE: modality, sentence subordination, negation as failure, macros, passive voice, numbers and strings as objects. Second, we describe NP ACE as a sublanguage of ACE used to flexibly and concisely verbalise discourse representation structures in ACE. Third, we discuss translating ACE into OWL DL. Fourth, we introduce ACERules, a powerful implementation of prioritised rules expressed in ACE. Version 5 of ACE and its associated tools have been prototypically implemented.

Keyword List

Attempto Controlled English, ACE, NP ACE, DRACE, OWL DL, ACERules

Project co-founded by the European Commission and the Swiss State Secretariat for Education and Research within the Sixth Framework Programme

Attempto Controlled English 5: Language Extensions and Tools I

Norbert E. Fuchs, Kaarel Kaljurand, Tobias Kuhn

Department of Informatics

&

Institute of Computational Linguistics

University of Zurich

Email: {fuchs, kalju, tkuhn} @iifi.unizh.ch

3 September 2006

Abstract

This report presents four tracks of research on Attempto Controlled English (ACE). First, we present the new features of version 5 of ACE: modality, sentence subordination, negation as failure, macros, passive voice, numbers and strings as objects. Second, we describe NP ACE as a sublanguage of ACE used to flexibly and concisely verbalise discourse representation structures in ACE. Third, we discuss translating ACE into OWL DL. Fourth, we introduce ACERules, a powerful implementation of prioritised rules expressed in ACE. Version 5 of ACE and its associated tools have been prototypically implemented.

Keyword List

Attempto Controlled English, ACE, NP ACE, DRACE, OWL DL, ACERules

Contents

1. Introduction

The original title “ACE Language Extensions I” of this deliverable – determined quite some time ago in the updated I2 workplan – does not do justice to the contents of the deliverable that covers a whole range of topics:

- version 5 of Attempto Controlled English
- NP ACE: Verbalising DRSs with ACE Noun Phrases
- updates to the ACE-to-OWL Conversion
- ACERules: Courteous Logic Programs in ACE

Thus we decided to change the title into "Attempto Controlled English 5: Language Extensions and Tools I"

Though the topics of this deliverable address diverse issues, they serve one common goal, namely to turn Attempto Controlled Engish into a user-friendly, powerful and flexible knowledge representation language specifically for the semantic web.

2. Version 5 of Attempto Controlled English

2.1. Motivation

We have extended ACE 4 by

- modality
- sentence subordination (`that`-subordination)
- negation as failure
- macros
- passive voice
- numbers and strings as objects

Furthermore, we have generalised some ACE 4 constructs, clarified some ACE 4 constructs, and removed a small number of ACE 4 constructs that could lead to ambiguity or to inconsistency.

The result of all these activities is version 5 of ACE. Please note that there will be further extensions of ACE 5 in deliverable I2-D11.

The transition from ACE 4 to ACE 5 was partially motivated by requests from our REWERSE partners, partially by suggestions from external users of ACE, and last but not least by our intention to develop ACE into a full-blown knowledge representation language specifically for the semantic web.

2.2. Major ACE 5 Language Extensions

2.2.1. Modality

ACE 5 provides the alethic modalities of possibility and necessity and their negations – both sentence-internally with modal auxiliaries, and sentence-initially with predefined modal phrases.

Modal Auxiliaries

The modal auxiliaries are `can` and `must` and their negations. Here are some examples.

```
A customer (can/cannot/can not/can't) enter a card.
A customer (must/has to/does not have to) enter a card.
```

Note that `must` has the alias `have to` that is used to express the negation of `must` as `does/do not have to`. (In English `must not` is not interpreted as the negation of `must`.)

To logically represent alethic modalities we extended the DRS language by the unary prefix operators <> (possibility) and [] (necessity) that take a DRS as argument. For the sentence

```
A customer must enter a card.
```

we get the (pretty-printed) DRS

```
[A]
object(A, atomic, customer, person, cardinality, count_unit, eq, 1)-1
   [#]
   [B, C]
   object(B, atomic, card, object, cardinality, count_unit, eq, 1)-1
   predicate(C, unspecified, enter, A, B)-1
```

Note that the modal operator [] is pretty-printed as `[#]` to distinguish it from an empty list `[]` of discourse referents.

The scope of a sentence-internal modality is the complete verb phrase that immediately follows the modal auxiliary.

```
A customer {must enter a card} and {types a code}.
```

The scope of a modal auxiliary cannot be extended. Instead a predefined modal phrase (see below) has to be used.

Noun phrases introduced in the scope of a modal auxiliary are not accessible for anaphoric references.

```
A customer must enter a card. *It is correct.
```

Note that sentences prefixed with * are not accepted by the ACE parser.

Predefined Modal Phrases

The predefined modal phrases are `it is (not) possible that` and `it is (not) necessary that`. The word `that` must be followed by a complete ACE sentence. Here is an example.

```
It is necessary that a customer enters a card.
```

with the DRS

```
[]
   [#]
   [A, B, C]
   object(A, atomic, customer, person, cardinality, count_unit, eq, 1)-1
   object(B, atomic, card, object, cardinality, count_unit, eq, 1)-1
   predicate(C, unspecified, enter, A, B)-1
```

Please note, that the sentence `It is necessary that a customer enters a card.` does not have the same meaning as the sentence `A customer must enter a card.` as can be seen by their different DRSs. In the first sentence `a customer` is included in the scope of the modality, while in the second sentence it is not included.

The scope of a predefined modal phrase is the sentence that immediately follows the word `that` of the predefined phrase.

```
It is necessary that {a customer enters a card} and {the ATM is broken}.
```

`To` extend the scope the word `that` has to be repeated.

```
It is necessary that {a customer enters a card and that the ATM is
broken}.
```

Noun phrases introduced in the scope of a predefined modal phrase are not accessible for anaphoric references.

```
It is necessary that a customer enters a card. *It is correct.
```

Other Constructs of Modality

Following suggestions by ACE users we considered, but decided not to adopt, some other constructs to express modality. Here are two suggestions, and the reasons why we did not adopt them.

Sentence-initial adverbs like `possibly`, `necessarily`, `eventually`, `always` etc. modify sentences as in

```
Possibly a customer enters a card.
```

However, native English speakers judged only some of these adverbs acceptable in sentence-initial position, while others were deemed not acceptable. Thus sentence-initial adverbs do not offer a general and generally accepted solution.

Other ACE users proposed to define some sentence-internal adverbs as sentence modifying, for instance

```
A customer eventually enters a card.

A customer always enters a card.
```

This proposal, however, conflicts with the standard interpretation of adverbs as verb modifying. Even if we defined, for example, `always` as sentence modifying, how about its synonyms `ever`, `forever`, `permanently`, `invariably`, `perpetually`? Should they be treated as `always`, or rather not? Altogether, the proposal would certainly confuse users, and thus we decided not to adopt it.

2.2.2. Sentence Subordination

The sentence subordination (`that`-subordination) of ACE 5 allows us to use sentences as objects of transitive and ditransitive verbs. Here are some examples.

```
A customer believes that his own card is correct.
A customer tells a clerk that the ATM is broken.
```

The word `that` must be followed by a complete ACE sentence.

To logically represent sentence subordination we added to the DRS language the binary infix operator : that takes a label as left argument and a DRS as right argument. The label is used as argument for the predicate/[5,6] conditions derived from (di-) transitive verbs. For the sentence

```
A customer believes that his own card is correct.
```

we get the (pretty-printed) DRS

```
[A, B, C]
object(A, atomic, customer, person, cardinality, count_unit, eq, 1)-1
predicate(B, unspecified, believe, A, C)-1
   C
   [D, E, F]
   relation(F, card, of, A)-1
   property(D, correct)-1
   predicate(E, state, be, F, D)-1
   object(F, atomic, card, object, cardinality, count_unit, eq, 1)-1
```

Scoping is similar to that of predefined modal phrases. The scope of sentence subordination is the sentence that immediately follows the word `that` of the predefined phrase.

```
A customer believes that {his own card is correct} and {the ATM is broken}.
```

To extend the scope the word `that` has to be repeated.

```
A customer believes that { his own card is correct and that the ATM is broken}.
```

Noun phrases introduced in the scope of sentence subordination are not accessible for anaphoric references.

```
A customer believes that his own card is correct. *It is not correct.
```

Adverbs and prepositional phrases modifying the main verb have to be inserted before the word `that`.

```
A customer believes correctly in the bank that his own card is correct.
```

If the complementiser `that` could be confused with the relative pronoun `that`, then the complementiser has to be written as `_that_`.

```
A manager sees in the morning that a clerk oversleeps. (= A manager sees in the morning. A clerk oversleeps the morning.)

A manager sees in the morning _that_ a clerk oversleeps. (= A manager sees in the morning the fact that a clerk oversleeps.)
```

2.2.3. Negation As Failure

ACE 5 complements its logical negation (`is not/does not/no/not/it is false that`) by negation as failure using the sentence-initial phrase `it is not provable that`. The word `that` must be followed by a complete ACE sentence. Note that there is no sentence-internal negation as failure. Here is an example.

```
It is not provable that a card is correct.
```

To logically represent negation as failure we added to the DRS language the unary prefix operator ~ that takes a DRS as argument. For the example sentence we get the (pretty-printed) DRS

```
[]
   NAF
```

```
[A, B, C]
object(A, atomic, card, object, cardinality, count_unit, eq, 1)-1
property(B, correct)-1
predicate(C, state, be, A, B)-1
```

in which ~ is represented as `NAF`. Scoping is similar to that of predefined modal phrases and of sentence subordination. The scope of negation as failure is the sentence that immediately follows the word `that` of the phrase `it is not provable that`.

It is not provable that {a card is correct} `and {the ATM is broken}.`

`To` extend the scope the word `that` has to be repeated.

It is not provable that {a card is correct `and that the ATM is broken}.`

Noun phrases introduced in the scope of negation as failure are not accessible for anaphoric references.

`It is not provable that a card is correct. *It is not correct.`

2.2.4. Macros

ACE 5 allows the definition of sentential macros and references to these macros. A macro definition introduces the name of the macro and its defining sentence. ACE sentences refer to a macro via its name. Macro expansion is the substitution of the name of a macro by its defining sentence; any anaphoric references occurring in the defining sentence are resolved after the substitution.

The definition of a macro consists of the word `proposition` followed by the name of the macro followed by a colon. After the colon follows the defining ACE sentence.

`Proposition P: A card is correct.`

Note that the ACE sentence does not have a truth-value.

Defined macros can be referred to by their names. A macro is referred to when its name is inserted after the word `that` of any of the sentence-initial phrases or of sentence subordination. Here are some examples.

`It is true that P.`

`It is false that P.`

`It is possible that P.`

`It is not provable that P.`

`A customer believes that P.`

Each occurrence of a macro name is replaced by its defining sentence; then any anaphors occurring in the defining sentence are resolved.

For the often occurring case

`Proposition P: A card is correct.`

`It is true that P.`

the abbreviation

`Fact P: A card is correct.`

can be used.

Definitions of macros can refer to *previously defined* macros.

`Proposition P: A card is correct.`

`Proposition Q: A customer believes that P.`

Note that macro names and variables associated with noun phrases cannot be confused since they are defined and called in syntactically different contexts. Thus with the above definitions the sentence

`P is true.`

would cause an error message “Unresolved variable: P.”.

What are macros good for?

The simplest application of a macro is to clearly identify a recurring phrase – and possibly to save some typing. Instead of

```
A customer assumes that a card is valid. A clerk checks that the card
valid. The machine determines that it is valid.
```

one can write

```
There is a card. Proposition CardIsValid: The card is valid. A customer
assumes that CardIsValid. A clerk checks that CardIsValid. The machine
determines that CardIsValid.
```

A more elaborate application of macros is to use them to define a set of theories or ontologies of a domain, and to easily switch between the theories/ontologies.

```
Proposition TheoryA:                     Proposition TheoryB:
  Sentence1OfTheoryA                        Sentence1OfTheoryB
  and                                       and
  …                                         …
  and                                       and
  SentenceNOfTheoryA.                       SentenceMOfTheoryB

If it is true that TheoryA then … .
If it is true that TheoryB then … .
```

ACE users will certainly find many other applications of macros.

2.2.5. Passive Voice

To increase the expressivity of ACE 5, and to allow a more flexible verbalisation of DRSs in ACE we decided to add the passive voice.

Passive is available for both transitive and ditransitive verbs. Here are some examples:

active	passive
`A customer enters a card.`	`A card is entered by a customer.`
`A customer gives a clerk a card.`	`A clerk is given a card by a customer.`
`A customer gives a card to a clerk.`	`A card is given to a clerk by a customer.`

Note that an active sentence and its passive counterpart generate the same DRS.

Note further that ACE 5 passives require a `by …` phrase indicating the agent of an action. If this agent is unknown a phrase `by somebody` or `by something` must be used.

2.2.6. Numbers and Strings as Objects

In ACE 4 numbers could only be used as determiners of nouns (`2 apples`) and strings only as appositions to noun phrases (`a message "errors"`). ACE 5 introduces numbers (integers, reals) and strings as independent objects. Here are examples:

```
The temperature is -12.

The address is "Paris".
```

generating the DRS

```
[A, B, C, D, E, F]
object(A, atomic, integer, object, cardinality, count_unit, eq, 1)-1
integer(A, -12)-1
predicate(B, state, be, C, A)-1
object(C, atomic, temperature, object, cardinality, count_unit, eq, 1)-1
object(D, atomic, string, object, cardinality, count_unit, eq, 1)-2
string(D, Paris)-2
predicate(E, state, be, F, D)-2
```

```
object(F, atomic, address, object, cardinality, count_unit, eq, 1)-2
```

Note that reals are not yet implemented.

2.3. Generalisations of ACE 4 Constructs

Taking into account our own experience and the feedback of ACE users we generalised some ACE 4 constructs. These are small changes that, nevertheless, contribute essentially to the expressivity of the language.

2.3.1. Indefinite Pronouns Can Have Variables in Apposition

Indefinite pronouns (`somebody`, `something`, `everybody`, ...) can have a variable in appositions, for instance

```
If somebody X has a card then X enters the card.
```

```
Everybody X has a card that X enters.
```

2.3.2. Proper Names Can Have Relative Phrases

Proper names can have relative phrases, as for instance in

```
John who has a card enters it.
```

2.3.3. Variables Can Have Relative Phrases

Variables can have relative phrases, as for instance in

```
There is a customer X. X who has a card enters it.
```

2.3.4. New Predefined Phrases `it is true/false that`

We added to the already existing predefined phrases two new predefined phrases `it is true that` and `it is false that` (with the alias `it is not the case that` taken over from ACE 4 for backward compatibility). See section 2.2.4.

2.3.5. Numbers as Words

We added `zero/null`, `one`, ..., `twelve` as alternatives for `0`, `1`, ..., `12`.

2.4. Clarifications of ACE 4 Constructs

We clarified some ACE 4 constructs that users may have found confusing.

2.4.1. Quantifier `no`

The quantifier `no` is interpreted as `every ... not` unless the construction is existential (`there is no`). This can be best seen by the DRSs generated.

```
No man waits.
[]
   [A]
   object(A, atomic, man, person, cardinality, count_unit, eq, 1)-1
   =>
   []
      NOT
      [B]
      predicate(B, unspecified, wait, A)-1

There is no man who waits.
[]
   NOT
   [A, B]
   object(A, atomic, man, person, cardinality, count_unit, eq, 1)-1
   predicate(B, unspecified, wait, A)-1
```

2.4.2. Saxon Genitives

Noun phrases NP introduced in phrases "proper name's NP" and "possessive pronoun NP" are interpreted as existentially quantified, for instance

```
John's card = a card of John
his card = a card of him
```

2.5. Removal or Restrictions of ACE 4 Constructs

To avoid possible ambiguities and inconsistencies we removed or restricted some constructs available in ACE 4.

2.5.1. Proper Names in Apposition

Proper names in apposition, as for instance in

```
a man John
```

could lead to ambiguities and are disallowed in ACE 5.

2.5.2. `there is`-Phrases

The phrases `there is` & definite noun phrase, `there is` & proper name and `there is` & variable, as for instance in

```
there is the man
there is John
there is X
```

could lead to inconsistencies and are disallowed in ACE 5.

2.5.3. Multi-Words

While in ACE 4 multi-words could be juxtaposed with intervening blanks and with hyphens, in ACE 5 only hyphens are allowed.

```
a persona-non-grata
```

2.5.4. Saxon Genitive of Coordinations and Coordinations of Possessive Pronouns

The Saxon genitive of noun phrase coordinations (`John and Mary's`) and coordinations of possessive pronouns (`his and her`) are no longer allowed.

Instead of

```
John and Mary need some money. *John and Mary's card is correct.
```

use for instance

```
John and Mary need some money. Their card is correct.
```

Instead of

```
John and Mary need some money. *His and her card is correct.
```

use for instance

```
John and Mary need some money. Their card is correct.
```

A general alternative is to use the `of`-genitive, for example

```
John and Mary need some money. The card of John and Mary is correct.
```

3. NP ACE: Verbalising DRSs with ACE Noun Phrases

3.1. Introduction

We describe a subset of ACE called NP ACE. In NP ACE there is no sentence coordination or sentence negation, instead ACE noun phrases (NPs) are used to encode the same meaning using relative clauses. We show how NP ACE can be used to verbalise DRSs which only contain implication boxes, i.e. DRSs of the form *drs(_, _) => drs(_, _)*. Even though NP ACE has weaker expressive power than full ACE, most statements found in ontology and rule languages can be expressed in it. Furthermore, the verbalisations are easy to read *every*-sentences.

We also describe the implementation of a verbaliser (DRACE NP) which translates DRSs into NP ACE, but do not discuss how different ways of verbalising DRSs can be combined so that the end-user experiences just one verbalisation and all the choices for the best possible verbalisation are done for him.

In the following discussion, when describing the input to the verbaliser we mostly avoid examples of DRSs but represent them as ACE sentences as a semantically equivalent from of the DRSs. I.e. instead of describing how DRSs can be verbalised in NP ACE, we describe how full ACE sentences can be paraphrased in NP ACE.

3.2. Relation to Core ACE

A verbalisation of Attempto DRSs into an ACE subset called Core ACE was described in [Fuchs et al. 05]. When designing Core ACE, the goal was to deterministically handle any DRS, less attention was paid to the compactness and readability of the resulting verbalisation. E.g. Core ACE uses simple NPs which do not include relative clauses. Whenever a discourse referent is reused in the DRS, a definite NP must be used in Core ACE to account for the argument sharing. In order to express the coordination and negation of predicates, Core ACE uses sentence coordination and negation. As a result, Core ACE sentences are often long and contain multiple anaphoric references. This was pointed out by the cross-reader of deliverable I2-D5, who was especially concerned about the use of sentence negation instead of NP and VP (verb phrase) negation, e.g. the DRS equivalent to the ACE sentence 'No child drives a car.' is verbalised in Core ACE as 'If there is a child then it is false that the child drives a car.'. In NP ACE the verbalisation is 'No child drives a car.'.

3.3. Syntax and Semantics of NP ACE

The syntax of NP ACE is otherwise equivalent to the syntax of ACE *every*-sentences, but lacks support for NP modifiers (apart from relative clauses), VP modifiers, possessive constructions, ditransitive verbs, NP-conjunctions, data items (strings and integers) and embedded *every*-constructions. While full ACE allows relative pronouns to be either 'who', 'which' and 'that', NP ACE uses only 'that'.

The semantics of NP ACE is the same as for full ACE.

3.4. Representing Sentences with Noun Phrases

ACE has been designed with syntactic expressivity in mind. This means, among other things, that instead of applying logical operators (*and*, *if-then*, *not*, *or*) to sentences, one can choose to apply those operators on NP or VP level.

Let's look at how ACE sentences can be equivalently expressed as NPs with relative clauses.

```
A man owns a car. # sentence
There is a man that owns a car. # 'there is' + NP

A man owns a car. The man feeds a cat.
There is a man that owns a car and that feeds a cat.

A man owns a car. The car runs-on some gas.
There is a man that owns a car that runs-on some gas.


A man owns a car. A cat sees the car.
```

```
There is a man that owns a car that a cat sees. # rel obj
There is a man that owns a car that is seen by a cat. # passive

A man owns a car. A cat sees the man.
There is a man that owns a car and that a cat sees. # rel obj
There is a man that owns a car and that is seen by a cat. # passive
There is a cat that sees a man that owns a car.
```

Relative clauses provide a compact syntax to express argument sharing between sentences, either the subject or the object of consecutive sentences can be shared. To make the following discussion simpler we represent the sentences as a list of pairs of nouns. A pair doesn't have a name, i.e. the name (i.e ACE verb) is not important in the context of this discussion. Deciding whether a list of sentences can be represented as a complex NP amounts to checking whether the list of pairs can be organised into a connected graph and traversed.

The sentences are now represented in the following way.

```
(man car)

(man car) (man cat)

(man car) (car gas)

(man car) (cat car)

(man car) (cat man)
```

To organise and traverse the list of pairs we can apply the following operations on the list.

First, we can reorder the elements in the list because the corresponding sentences are coordinated and in ACE coordination is symmetric.

```
(man car) (cat man) == (cat man) (man car)
```

Secondly, we can invert the elements in the pair, as ACE sentences can be expressed in passive voice which is treated equivalently to the active voice. This inversion is marked by the character i.

```
(man car) (cat man) == (man car) i(man cat)

A man owns a car. A cat sees the man. # Original sentences
There is a man that owns a car and that is seen by a cat. # Passivised
relative clause
There is a man that owns a car and that a cat sees. # relative clause with
verb-arg inversion
```

Third, in order to connect pairs to each other, we have two possibilities, either the two pairs have matching subjects ('and that' construction), or the object of the first pair matches the subject of the second pair ('that' construction).

```
(man car) (car gas)
There is a man that owns a car THAT runs-on some gas.

(man car) (man cat)
There is a man that owns a car AND THAT feeds a cat.
```

The following Prolog program organises and traverses the list of pairs. The program essentially generates all verbalisation plans, or fails if there are none. The main subject is the starting point of the traversal.

```
% ---------------------------------------------------------------
% np_path(MainSubject, Pairs, OrganisedPairs)
% ---------------------------------------------------------------

np_path(_, [], []).

% Select and finish
np_path(Subject, [(Subject, Object)], [(Subject, Object)]).

% Select, invert, and finish
```

```
np_path(Subject, [(Object, Subject)], [i(Subject, Object)]).

% Select and continue with subject ('and that')
np_path(Subject, [A, B | Pairs], [(Subject, Object) | OrganisedPairs]) :-
        select((Subject, Object), [A, B | Pairs], RestPairs),
        np_path(Subject, RestPairs, OrganisedPairs).

% Select and continue with object ('that')
np_path(Subject, [A, B | Pairs], [(Subject, Object) | OrganisedPairs]) :-
        select((Subject, Object), [A, B | Pairs], RestPairs),
        np_path(Object, RestPairs, OrganisedPairs).

% Select, invert, and continue with subject ('and that')
np_path(Subject, [A, B | Pairs], [i(Subject, Object) | OrganisedPairs]) :-
        select((Object, Subject), [A, B | Pairs], RestPairs),
        np_path(Subject, RestPairs, OrganisedPairs).

% Select, invert, and continue with object ('that')
np_path(Subject, [A, B | Pairs], [i(Subject, Object) | OrganisedPairs]) :-
        select((Object, Subject), [A, B | Pairs], RestPairs),
        np_path(Object, RestPairs, OrganisedPairs).
```

An example:

```
?- np_path(Subject, [ (man, car), (car, gas)], OrganisedPairs).

Subject = car
OrganisedPairs = [ (car, gas), i(car, man)] ;

Subject = man
OrganisedPairs = [ (man, car), (car, gas)] ;

Subject = car
OrganisedPairs = [i(car, man), (car, gas)] ;

Subject = gas
OrganisedPairs = [i(gas, car), i(car, man)] ;

No
```

It is clear that not all lists of pairs can be represented as a connected graph, consider:

```
A man owns a car. A cat drinks some milk.

(man car) (cat milk)
```

In this example, there exists a pair *(cat milk)* which neither shares the subject nor the object with the other sentences. Therefore, there are ACE texts which cannot be "packed" into complex NPs.

Note, that there are also configurations which form a connected graph but which cannot be traversed given the third rule. The simplest of such graphs is:

```
o -- o -- o -- o -- o
          |
          o
          |
          o
```

3.5. Adding Support for Negation and Disjunction

The previous section discussed only conjoined sentences. In ACE, sentences can also be negated and disjoined. In case of sentence negation, there is a corresponding list of pairs which is under negation. This list of pairs can be treated independently, but its main subject must match a subject in an upper list level. Similarly, in case of disjoined sentences there is a corresponding disjunction of lists of pairs. Those lists must have the same subject and match a subject in an upper list.

3.6. Implications

Implications, or DRSs of the form *drs(_, _) => drs(_, _)* can be expressed in ACE either by *if-then* sentences or by *every*-sentences. NP ACE uses the latter which has the form:

```
'Every' NP SentenceWithoutSubject .
```

where SentenceWithoutSubject is a normal sentence (just one sentence) without subject. E.g.

```
Every man that owns a car has a job and has a driving-license.
```

In order to use an *every*-sentence, the subjects of the left-hand side and the right-hand side must match, otherwise both sides can be verbalised independently. Or in terms of lists of pairs:

```
implication(Pairs1 => Pairs2, OPairs1 => OPairs2) :-
  np_path(Subject, Pairs1, OPairs1),
  np_path(Subject, Pairs2, OPairs2).
```

3.7. Usage Examples

If-then expressions (implications) are the main component in both rule languages and ontology languages. Such expressions usually have a high degree of argument sharing. To handle this, NP ACE offers relative clauses, which are arguably easier to "consume" for average users as the number of explicit anaphoric references in different locations in the formula is reduced.

Another good property of NP ACE is that it is easier to "read off" from the syntax of NP ACE statements the semantic expressivity of the underlying logic. See [Pratt-Hartmann 03] where a language similar to NP ACE is defined and the complexity (decidability) of reasoning about the satisfiability of the statements is measured. The presence of explicit anaphoric references plays an important role.

We now look at a few examples on how NP ACE can be used to capture statements in other formal languages.

NP ACE can verbalise the following Description Logic statement

```
Male ⊓ Person ⊓ ¬ Husband ⊑ Bachelor
```

by

```
Every male that is a person and that is not a husband is a bachelor.
```

or the following statement

```
∃ live-in {London} ⊔ ∃ live-in {Paris} ⊑ ∃ live-in {Europe}
```

by

```
Everybody that lives-in London or that lives-in Paris lives-in Europe.
```

By using explicit anaphoric references (definite NPs with or without variables), the following rule

```
animal(X) ∧ animal(Y) ∧ food(Z) ∧ eat(X, Z) ∧ eat(Y, Z) → hate(X, Y)
```

can be captured by:

```
Every animal that eats some food that is eaten by a animal is hated by the
animal.
```

3.8. Implementation

3.8.1. Introduction

Verbalisation of Attempto DRSs into NP ACE is implemented as a SWI-Prolog program DRACE NP which is part of the Attempto tools and used to paraphrase ACE sentences.

3.8.2. Input

DRACE NP accepts only a subset of the DRS language as input. The constraints are the following.

- The top-level DRS can contain only implications, i.e. DRSs of the form *drs(_, _) => drs(_, _)* and objects, i.e. conditions of the form *object/8*.

- The embedded DRSs must not contain the following conditions: *named*, *modifier*, *proper_part_of*, *property*, *relation*, *query*, *quoted_string*, *string*, *integer*, *predicate/6*, *drs(_, _) => drs(_, _)*, *Label:drs(_, _)* and *~drs(_, _)*. Note that the ACE 5 extensions of necessity ([]) and possibility (<>) are handled similarly to the way negation is handled, but in this case using 'must' and 'can' instead of 'does/do not'.
- Each DRS must contain a predicate condition, either *predicate/4* or *predicate/5*. The only exception is the left implication box which is allowed to contain just a single *object/8*.
- The top-level implication boxes must share at least one discourse referent. This referent cannot map to a top-level object (e.g. a proper name) nor to a plural noun (e.g. 'at most 2 men'). Note also that DRSs of the form *drs(_, [- drs(_, _)]) => drs(_, _)* cannot be verbalised as in this case the sides of the implication cannot share any discourse referents.
- All predicates must share arguments so that a connected graph forms (cf. section 3.4).
- Each embedded DRS must share at least one referent with a box on an upper level. Disjunction boxes (*drs(_, _) v drs(_, _)*) must share this argument also with each other.

Given those restrictions, the DRSs which correspond to the following ACE sentences, do not belong to the input subset.

- A man waits.
- Everybody knows a dog of a man.
- Everybody runs fast.
- It is false that a man waits. (Note that the semantically equivalent 'No man waits.' is allowed.)
- If John drinks then he eats.
- If a man eats then a woman drinks.
- If there is a man then he waits and a woman owns a cat.

3.8.3. Output

The output is generated in NP ACE. Read more in section 3.4. In case the input is rejected, an error message is output.

As a special case, the DRS of the form *drs(_, _) => drs(_, -[drs(_, _)])* is verbalised as:

```
'No' NP SentenceWithoutSubject .
```

e.g.

```
No child drives a car.
```

3.9. Shortcomings

Although NP ACE can handle most argument sharing with relative clauses, sometimes explicit anaphoric references are needed. ACE offers various ways to make anaphoric references, the most flexible being variables. The current implementation uses variables when verbalising conditions that correspond to indefinite pronouns ('somebody', 'nothing', 'everyone', ...) and definite NPs when verbalising conditions that correspond to common nouns ('man', ...). This causes sometimes "ugly" output (e.g. 'Everybody X waits.') and sometimes incorrect output (e.g. the DRS corresponding to 'There is a man X. Every man likes X.' is incorrectly verbalised as 'There is a man. Every man likes the man.'). A better and more readable solution would introduce variables only when they are needed, i.e. only when an object is being referred to at some point in the DRS and this reference cannot be handled by a relative clause.

The current implementation never uses *comma-and*. This causes sometimes an incorrect verbalisation (e.g. the DRS that corresponds to 'Every man works, and eats or sleeps.' is verbalised as 'Every man works and eats or sleeps.').

Some combinations of modality and negation cannot be expressed, e.g. the DRS corresponding to 'If there is a man then it is possible that it is false that he sees a cat.'

In general, many different verbalisations are possible. DRACE NP currently does not apply any particular selection but delivers just the first solution that is generated. E.g. the following NP ACE sentences are semantically equivalent and it is not clear which of them should be preferred.

```
  Every man who loves a woman who loves the man smiles.
  Every man that is loved by a woman and that loves the woman smiles.
```

One could also argue that in the presence of discourse referents which cannot be verbalised by relative clauses only, the Core ACE verbalisation is more suitable, e.g.

```
  If a woman loves a man and the man loves the woman then the man smiles.
```

3.10. Future Work

In the future we intend to handle a larger input language by adding support for *relation/4*, data items (*number*, *string*), modifiers (*modifier*, *property*), and embedded implications.

4. Updates to the ACE-to-OWL Conversion

4.1. Data-Valued Properties

Exploiting the support for integers and strings in ACE 5, partial support for data-valued properties was added to the ACE-to-OWL conversion. Now it is possible to describe data-valued properties between individuals and integers/strings. E.g.

```
John's age is 31.
The latitude of Greenwich is 0.
There is some ice. Its temperature is -2.
```

Also, data-valued properties can be used to generate the *hasValue* construction in OWL's class descriptions.

```
Every person whose age is 10 is a child.
```

At this point, there is no support for describing properties (such as domain and range) of data-valued properties.

4.2. Error Messages

As the conversion from ACE to OWL DL accepts only a subset of the DRS language as its input, we output error messages in case the input of the conversion is not in this subset. The error messages refer to ACE constructs instead of the DRS constructs as most users of the conversion will work on the ACE level and the preprocessing step of translating ACE texts into the DRS language takes place in the background, hidden from the user.

There are two kinds of error messages: messages about illegal (or currently unsupported) conditions in the DRS and messages about illegal argument sharing in *if-then* conditions.

4.2.1. Illegal or Unsupported Conditions

In most cases the DRS conditions contain the original ACE words (in a lemmatised form). They also include sentence identifiers. This additional information is reported along with the error message. In case the DRS is not generated from an ACE text, but by some other means, the sentence identifier can be used as a general location identifier.

Condition	**Message**
modifier(_, _, none, Adverb)-Id	Adverbs not supported.
modifier(_, _, Preposition, _)-Id	Prepositional phrases not supported.
proper_part_of(_, _)-Id	Noun phrase conjunction not supported.
property(_, Adjective)-Id	Attributive adjectives not supported.
query(_, _, QueryWord)-Id	Queries not supported.
query(_, QueryWord)-Id	Queries not supported.
predicate(_, _, Verb, _, _, _)-Id	Ditransitive verbs not supported.
relation(_, Noun, of, _)-Id	Possessive constructions not supported. (Only in case the possessive was not used in a data-valued property construction.)
[](DRS)	Necessity not supported.
<>(DRS)	Possibility not supported.
~(DRS)	Negation as failure not supported.
Label:DRS	Sentence subordination not supported.
DRS1 v DRS2	Disjunction not supported. (Only in case of top-level disjunction.)

4.2.2. Illegal Argument Sharing

Message	**Example**
The if-part and the then-part must share arguments.	If a man owns a dog then a woman owns a cat.
The then-part must refer to exactly one noun phrase in the if-part.	Every man who owns a cat likes a dog that likes the cat.
Unsafe anaphoric reference in the if-part.	Every man who likes himself owns a dog.
Unsafe anaphoric reference in the then-part.	Every man who owns a dog likes himself.

5. ACERules: Courteous Logic Programs in ACE

5.1. Introduction

Rule-based systems play an important role in different applications (e.g. business rule systems, policy rule systems). Verbalisation of such rules in natural language has been considered an important feature (e.g. [Demey et al. 02]). In most cases, the rules have to be validated by people who are not familiar with formal languages, and thus natural descriptions of the formal rules are very helpful. But this means that we have at the end two descriptions of the same rule system: one in a formal language, and one in natural language. Thus, we have to translate from one to the other, and we have to make sure that both are equivalent. Depending on the size of the rule-set, this can require a lot of additional work, and it is a potential source of errors.

In order to solve this problem, we provide the ACERules system. It allows to write rule-sets in Attempto Controlled English (ACE). In this way, we only need a single language to write rules that are formal and human readable at the same time. Thus, we can eliminate the overhead of verbalising formal rules and of creating formal rules from descriptions in natural language.

As theoretical background, we use courteous logic programs (CLP) [Grosof 97]. They provide classical negation, negation as failure (NAF), priorities, and mutual exclusion.

5.2. Courteous Logic Programs

As most rule systems, courteous logic programs are divided into facts and rules. Each rule has a head and a body, where the body reflects the *if*-part and the head stands for the *then*-part of the rule. A fact has no conditions and thus consists only of a head (we can also say: a fact has the condition *true* as its body).

Horn clauses [Horn 51] were one of the first attempts to get an efficiently tractable subset of first-order logic. They disallow any form of negation. General logic programs [Lloyd 84] extend Horn clauses by negation as failure which makes them more practical. The programming language *Prolog* can be seen as an implementation of general logic programs. Extended logic programs [Gelfond & Lifschitz 91] are an extension of general logic programs and allow to use both negation as failure and classical negation. This has the downside that rule-sets can get inconsistent (which is not possible with Horn clauses and with general logic programs). In order to overcome this problem, courteous logic programs are introduced [Grosof 97]. They allow to define priorities of the rules in order to resolve inconsistencies. If the priorities do not allow to resolve an inconsistency between $\neg p$ and p then the program behaves "courteously", i.e. skeptical: neither $\neg p$ nor p is seen as a consequence of the program. Despite all these extensions, there are still strong restrictions for the rules and facts of courteous logic programs:

- No disjunctions are allowed.
- Classical negation and negation as failure can only be applied on atomic propositions.
- Every head of a rule has to be a literal.

A literal is meant to be an atomic proposition or the (classical) negation of an atomic proposition. To show how courteous logic programs work and how they are represented in ACE, let us take a look at the following example [Grosof 97].

```
Every quaker is a pacifist.
No republican is a pacifist.
Nixon is a quaker.
```

This program consists of two rules and one fact. With a forward-chaining interpreter, we can create the answer set of this program (see [Dörflinger 05] for details about forward and backward interpretations of CLPs). An answer set is the set of all the facts that are a consequence of the program. It our case we get:

```
Nixon is a quaker.
Nixon is a pacifist.
```

This answer set is consistent, and there was no conflict during the inference. By adding the following fact to our program, we force a conflict.

```
Nixon is a republican.
```

Now we can conclude both: Nixon is a pacifist and he is not a pacifist. Since we did not yet introduce priorities, the program behaves "courteously" and does not conclude one or the other. Thus, we get an answer set with no information about Nixon being a pacifist or not.

```
Nixon is a quaker.
Nixon is a republican.
```

In order to get a result concerning Nixon's pacifism, we can introduce priorities and define which of the two rules has priority over the other. This has to be done by assigning labels to the rules. With *overrides*-statements, we can then define the priority structure. For ACERules, we use double colons (::) for the declaration of labels. We can redefine our rules as follows.

```
Quaker-Rule :: Every quaker is a pacifist.
Republican-Rule :: No republican is a pacifist.
Republican-Rule overrides Quaker-Rule.
```

With these modifications, we can conclude that Nixon is not a pacifist, since the rule saying that every quaker is a pacifist has lower priority. Thus, the answer set is now:

```
Nixon is a quaker.
Nixon is a republican.
It is false that Nixon is a pacifist.
```

The "courteous" interpretation makes sure that the answer set is always consistent. Furthermore, the labels and overrides-statement are a useful feature to update existing knowledge-bases (see [Grosof 97] for details and examples).

5.3. Negation as Failure

Negation as failure is a concept that is often found in rule systems and similar applications. It implies that we consider a statement wrong if we cannot prove its truth. For classical negation – in contrast – a statement is only considered wrong if we can prove its falsity. For some applications, only one form of negation is available (usually NAF); in other cases, negation as failure and classical negation can be used together. As already mentioned above, we allow both forms of negation in ACERules.

For classical negation, we use the usual English negation forms: "no", "nobody“, "nothing“, "does not“, "is not“, and the phrase "it is false that“. For negation as failure we use the phrase "it is not provable that“. One has to carefully decide which negation has to be used in a certain case. The two examples below show the differences.

```
If someone X is not a criminal then X is trustworthy.
If it is not provable that someone X is a criminal then X is trustworthy.
```

In the first case, we consider someone trustworthy only if we can *prove* that this person is not a criminal. In the second case, we consider everyone trustworthy as long as there is *no evidence* that this person is a criminal. Depending on the application and on the information available, both representations can be useful (but not both of them at the same time, since the second one implies the first one). In order to show how the two forms of negation can work together, we take a look at the following example.

```
If someone X is not a criminal and it is not provable that X is indebted
then X is trustworthy.
Bill is not a criminal.
John is not a criminal.
John is indebted.
Mary is not indebted.
```

Using this program, we can conclude that Bill is trustworthy since we can prove that he is not a criminal and there is no evidence that he is indebted. In the case of John and Mary, we cannot say whether they are trustworthy or not. The proof that John is trustworthy fails because we can prove that John is indebted. And the proof that Mary is trustworthy fails because we cannot prove that she is not a criminal.

5.4. Technical Details

This section takes a look at the technical details of the transformations that have to be applied. First, the rules and facts have to be translated into a CLP-compliant format that can be used by the CLP-interpreter. Next, the answer set is computed using the algorithms for courteous logic programs. Finally, we have to back-translate the answer set into an ACE text. These three operations are described in detail in the next subsections.

5.4.1. Transformation into Facts and Rules

It takes five steps to transform a rule system into facts and rules that comply with the restrictions of courteous logic programs. These five steps are:

1. Meta-preprocessing
2. Parsing (APE)
3. Condensation of the DRS
4. Grouping of predicates
5. Creation of the rules and facts

The DRS representation created by APE is relatively rich. This has the consequence that the transformation into a CLP-compliant format is not trivial. Let us consider the following exemplary rule throughout this subsection.

```
R1 :: Every customer has a card.
```

Since the Attempto parser APE does not support labels, we need a preprocessing step to handle the labels and *overrides*-statements. In this preprocessing step, the rule system is split up into label–sentence pairs (a label may be empty). In the next step, each sentence is parsed separately with APE. After these two steps, our exemplary rule leads to the following DRS.

```
[]
   [A]
   object(A,atomic,customer,person,cardinality,count_unit,eq,1)
   =>
   [B,C]
   object(B,atomic,card,object,cardinality,count_unit,eq,1)
   predicate(C,unspecified,have,A,B)
```

Unfortunately, we cannot translate this DRS directly into a rule, since there would be a conjunction of two predicates in the *then*-part of the rule. Courteous logic programs disallow such rules. In this case, we could split the rule into two rules, but there are cases where this does not work. For that reason, we try to condense the DRS by merging predicates that always occur together on the same level in the DRS. E.g. we can generally merge the representations for an adverb and its corresponding verb, since they always occur on the same level.

In some cases (like in our example), we still do not get a CLP-compliant rule. As a last opportunity, we can simply group the predicates in order to remove the critical conjunctions. We have to mention that this step can lead to unexpected results in some special cases. Thus, the user should be warned whenever this grouping can be harmful. In the current ACERules system such a warning is not yet generated, but it will be provided in the future. After grouping the predicates of the *then*-part, our example looks as follows.

```
[]
   []
   object(A,atomic,customer,person,cardinality,count_unit,eq,1)
   =>
   []
   group([pred_mod(unspecified,have,A,B,[]),
          object(B,atomic,card,object,cardinality,count_unit,eq,1)])
```

Finally, we have a DRS that complies with the restrictions of courteous logic programs. This DRS is now translated into an internal rule format, and then it can be used by the CLP-interpreter.

5.4.2. Generation of the Answer Set

The generation of the answer set for a courteous logic program can be divided into two steps. As a preprocessing step, we have to eliminate negation as failure. Then we can calculate the answer set.

1. Elimination of negation as failure
2. Computation of the answer set

In the first step, negation as failure is replaced by equivalent constructs without negation as failure. See [Antoniou et al. 00] for a detailed description of this transformation. In order to prevent an explosion in the number of rules, we introduce auxiliary facts that contain unbound variables. The ACERules engine makes sure that these facts are treated correctly.

In the second step, the answer set is calculated. The facts of the initial program are the starting point. This set of facts is then iteratively extended by forward reasoning until the final answer set is reached. During this process, inconsistencies have to be resolved when they occur. A detailed description of this procedure can be found in [Grosof 97] and [Dörflinger 05]

5.4.3. Transformation back to ACE

In order to get a nice representation of the results, we have to back-translate the answer set into ACE. For this purpose, we can use the verbaliser DRACE [Fuchs et al. 05] which translates DRSs into ACE texts. In order to do so, we first have to generate a valid DRS representation from the answer set. This means that we have to undo the transformations that we applied in the beginning. Thus, we have the following four steps to perform.

1. Generation of a preliminary DRS
2. Ungrouping
3. Expansion (i.e. undo condensation)
4. Verbalisation (DRACE)

The first step simply puts the facts into a DRS structure. The second step removes the group structures and places the contained predicates instead. As a third step, the condensation of the predicates has to be undone. Finally, we get a valid DRS that can be used by DRACE to create an ACE text.

5.5. Conclusions

We presented ACERules as a rule system application that allows us to define programs that contain rules and facts in the controlled natural language ACE 5. In the background, the programs are executed as courteous logic programs, and thus all inconsistencies are automatically resolved. The result of ACERules (i.e. the answer set of the program) is back-translated into ACE 5, and thus it is understandable for everyone.

ACERules makes verbalisation unnecessary, since the formal descriptions (input and output) are already in a natural and easy-understandable form. This makes life easier for rule system designers. Furthermore, it allows to simplify the fault-prone process of validation, since the formal rules are self-explanatory.

6. One More Thing

ACE 5 and the associated tools presented in this report have been prototypically implemented, and are already available, or will soon be available, via the Attempto web-interface or via web-services.

For lack of time there are some temporary restrictions, though. For instance, APE does not yet generate the FOL output for modality, sentence subordination, and negation as failure. Queries containing modal auxiliaries and passives are not yet fully supported. Neither are passives in user provided lexicons. Also, real numbers are not yet implemented.

Our testing and regression testing revealed some bugs that we did not yet find the time to fix, for example `his card` is still translated as `the card of him`.

The on-line documentation is up-to-date with the exception of the abstract ACE 5 grammar that needs some more work.

7. Conclusions

The WG I2 Updated Workplan for the months 30 – 48 (Deliverables concerning Controlled Natural Language) states:

> The following set of requirements for ACE will be addressed in future deliverables:
>
> - negation as failure
> - prioritised rules
> - decidable subsets of ACE
> - modality
>
> While working on verbalisation (deliverables I2-D5 and I2-D7), we noticed that ACE – though it is in general more expressive and flexible than the languages currently used or proposed for the semantic web – does not, or does not conveniently, provide some important features of web languages, for instance:
>
> - URIs
> - Unicode support
> - constructs similar to OWL's allValuesFrom and oneOf
> - data types similar to OWL's data type properties
> - data structures similar to RDF's Bag, Alt, Seq
> - procedural attachments
> - built-ins like in Prolog or SWRL
>
> We decided to extend ACE by these or equivalent features which resulted in a second set of additional requirements.

The description of deliverable I2-D9 reads:

> A first stab at the two sets of requirements above will address prioritised rules, negation as failure, forward and backward execution of prioritised rules, data structures and operations on them, procedural attachments, 'that' subordinated sentences, modality constructs based on 'that' sentences, translation of ACE into and from OWL and other web languages. The definition of the extended version of ACE and of prototypes of the associated tools (ACE parser, interpreters for rules, translators etc.) are to be delivered.

The present deliverable I2-D9 shows that we addressed most, though not all, of the requirements listed above. But we also addressed some other important issues that we encountered during our work and provided solutions for them.

The remaining work will be done in I2-D11 whose description reads:

> Remainder of the two sets of requirements above, imperative mood for interactive operations. The definition of the extended version of ACE and of prototypes of the associated tools (ACE parser, interpreters for rules, translators etc.) are to be delivered.

Research on ACE continues to be demanding, exciting and gratifying.

8. References

[Antoniou et al. 00] G. Antoniou, M.J. Maher, D. Billington. *Defeasible logic versus Logic Programming without Negation as Failure*. The Journal of Logic Programming 42, 47-57, 2000

[Demey et al. 02] Jan Demey, Mustafa Jarrar, Robert Meersman. *A Conceptual Markup Language That Supports Interoperability between Business Rule Modeling Systems*. Lecture Notes in Computer Science, Vol. 2519, 19-35, Springer, 2002

[Dörflinger 05] Marc Dörflinger, *Interpreting Courteous Logic Programs*, Diploma Thesis. Department of Informatics, University of Zurich, 2005

[Fuchs et al. 05] Norbert E. Fuchs, Kaarel Kaljurand, Gerold Schneider. *Verbalising Formal Languages in Attempto Controlled English I*, Deliverable I2-D5. Technical report, REWERSE, 2005

[Gelfond & Lifschitz 91] Michael Gelfond, Vladimir Lifschitz. *Classical negation in logic programs and disjunctive databases*. New Generation Computing, 9:365-385, 1990

[Grosof 97] Benjamin N. Grosof. *Courteous Logic Programs: Prioritized Conflict Handling For Rules*. IBM Research Report RC 20836. Technical report, IBM T.J. Watson Research Center, 1997

[Horn 51] Alfred Horn. *On Sentences Which are True of Direct Unions of Algebras*. The Journal of Symbolic Logic, Vol. 16, No. 1, 14-21, 1951

[Lloyd 84] John Lloyd. *Foundations of Logic Programming*. Springer, 1984

[Pratt-Hartmann 03] Ian Pratt-Hartmann. *A Two-Variable Fragment of English*. Journal of Logic, Language and Information, 12(1), 2003, pp. 13--45.