Attempto Controlled English — Not Just Another Logic Specification Language[*]

Norbert E. Fuchs, Uta Schwertel, Rolf Schwitter
Department of Computer Science, University of Zurich
{fuchs, uschwert, schwitter}@ifi.unizh.ch
http://www.ifi.unizh.ch/~fuchs/

1 Introduction

Specifications are a set of precisely stated properties or constraints which a software system must satisfy to solve a problem. Specifications describe the interface between the problem domain and the software system [Jackson 95], i.e. they define the purpose of the software system and its correct use [Le Charlier & Flener 98]. In which language should we express these specifications?

Formal specification languages – often logic languages or languages based on logic – have been advocated because they have an unambiguous syntax and a clean semantics, and promise substantial improvements of the software development process [cf. www.comlab.ox.ac.uk/archive/formal-methods]. In particular, formal specification languages offer support for the automatic analysis of specifications such as consistency verification, and the option to validate specifications through execution [Fuchs 92]. However, experience has shown that formal specification languages suffer from major shortcomings – they are hard to understand and difficult to relate to the application domain, and need to be accompanied by a description in natural language that "explains what the specification means in real-world terms and why the specification says what it does" [Hall 90]. Similar observations were made earlier by [Balzer 85]. Deville develops logic programs from primarily formal specifications [Deville 90]. Interestingly, he states the specified relation itself in natural language.

Traditionally, specifications have been expressed in natural language. Natural language as the prototypical means of human communication needs not to be learned, is easy to use and to understand. Enhanced by appropriate domain-specific notations, natural language can be used to express any problem. Nevertheless, again experience has shown that uncontrolled use of natural language can lead to ambiguous, imprecise and unclear specifications with possibly disastrous consequences for the subsequent software development [Meyer 85].

It seems that we are stuck between the over-flexibility of natural language and the potential incomprehensibility of formal languages. While some authors claim that specifications need to be expressed in natural language and that formal specifications are a contradiction in terms [Le Charlier & Flener 98], we are convinced that the advantages of natural and formal specification languages should be and can be combined.

The solution lies in the insight that natural language can be used very precisely. Examples are legal language and the so-called controlled languages used for technical documentation and machine translation [CLAW 96]. However, all these languages are usually ad hoc defined and rely on rather liberal rules of style and on conventions to be enforced by humans. Taking these languages as a lead we have defined the specification language Attempto Controlled English (ACE) – a subset of standard English with a domain-specific vocabulary, a restricted grammar, and a small set of construction and interpretation rules [Schwitter 98; Fuchs et al. 98]. ACE allows users to express specifications precisely, and in the terms of the application domain. ACE specifications are computer-processable and can be unambiguously translated into a logic language. Though it may seem informal, ACE is a formal language with the semantics of the underlying logic language.

There have been several projects with similar aims as the Attempto project, but in most cases the subsets of English to be used as specification languages were not systematically and clearly defined. Macias and Pulman, however, developed a system to write natural language specifications which resembles ours with the important difference that their system restricts only the form of composite sentences, but leaves the form of the constituent simple sentences completely free [Macias & Pulman 95]. As a consequence, the thorny problem of ambiguity remains and has to be resolved by the users post factum.

[*] Extended Abstracts of LOPSTR'98, Eighth International Workshop on Logic-Based Program Synthesis and Transformation, 15-19 June 1998, Manchester, UK. Technical Report Series, Department of Computer Science, University of Manchester, ISSN 1361-6161. Report number UMCS-98-6-1. http://www.cs.man.ac.uk/cstechrep/titles98.html

The rest of the paper is organised as follows. In section 2 we motivate the transition from full English to Attempto Controlled English and present a glimpse of ACE. Section 3 describes the translation of ACE specifications into discourse representation structures – a syntactical variant of first-order predicate logic – and into Prolog. Section 4 overviews querying and executing specifications. Finally, in section 5 we conclude and address further points of research.

2 From English to Attempto Controlled English

Specifications written in full natural language tend to be vague, ambiguous, incomplete or even inconsistent. Take an extract from the specification of a simple electronic watch:

```
A watch has two buttons A and B. When A and B are pressed during the normal
display of the time then the stop-watch is started and displayed.
```

Most readers will probably not notice the deficiencies in this short text because they interpret the specification in the context of their knowledge about watches. But in absence of this additional knowledge would you be able to definitely answer questions like:

- Are `A` and `B` pressed at the same time, or one after the other?
- Where is the stop-watch displayed? What is displayed?
- Who or what starts the stop-watch?

There is no definitive answer to the first question due to a structural ambiguity inherent in coordinated structures. Other typical sources for structural ambiguities in full natural language are overlapping scopes, attachment of prepositional phrases, attachment of relative sentences, and anaphora.

The uncertainty in answering the last two questions results from the incompleteness or vagueness of the specification. For example, the specification uses verbs in the passive with the result that no agents are made explicit.

Imagine a software developer who has no knowledge of the problem domain and is confronted with this – or with a much more complex – natural language specification: it is difficult, or even impossible, to derive a formal specification or a program.

To eliminate the most salient deficiencies of full natural language we propose the specification language Attempto Controlled English (ACE). ACE is a computer-processable subset of full English, and can be unambiguously translated into an executable logic language. To achieve this goal the language ACE is constructively defined on the basis of a relatively small set of construction and interpretation rules. All admissible ACE sentences are grammatically correct English; however, not all English sentences can be generated. Furthermore, unlike full English, ACE sentences are never ambiguous. The interpretation rules guarantee for each sentence exactly one translation into the logic language. Different interpretations can only be achieved by reformulating a sentence. In contrast to rules of formal languages the construction and interpretation rules of ACE are easy to use and remember since they are similar to English grammar rules and only presuppose basic grammatical knowledge.

More concretely, the vocabulary of ACE comprises predefined function word classes (e.g. determiners, conjunctions, prepositions) and a user-defined, domain-specific subset of content word classes (nouns, verbs, adjectives, adverbs). A specification is a sequence of ACE sentences. ACE sentences come as simple sentences, composite sentences, and query sentences. Simple sentences have the form *subject + predicate + (complements + adjuncts)* and express a true state of affairs. Composite sentences are built from simpler sentences with the help of predefined constructors: coordination (`and`, `or`), subordination (`if-then`, `who/which/that`), negation (`not`, `no`), quantification (`every`, `for every`, `there is a`). Query sentences allow users to pose *yes/no*-questions and *wh*-questions.

Here is one possible formulation of the extract of the watch specification in ACE

```
A watch has a button 'A' and a button 'B'.

If
  the watch shows the current time on the display
  and the user presses the button 'A' and the button 'B' at the same time
then
  the watch starts the stop-watch
  and shows the elapsed time of the stop-watch on the display.
```

This ACE specification employs – among others – the following construction rules:

- All verbs are used in the simple present tense, the active voice, the indicative mood, the third person singular.

- Content words can be simple (`watch`) or compound (`stop-watch`).
- Complements of full verbs can be noun phrases and prepositional phrases. Adjuncts are optional and can be realised as prepositional phrases (`on the display`) or adverbs.
- Composite sentences are built from simple sentences with constructors like `if-then`, `and`, `not`, `every` etc.
- Coordination is only possible between equal constituents (e.g. `the button 'A' and the button 'B'`)

The Attempto system analyses the ACE specification according to the following interpretation rules and reflects the analysis in a paraphrase:

- Verbs denote events (`press`) or states (`have`).
- The textual order of verbs determines the default temporal order of associated events and states.
- The default temporal order can be overridden by predefined discourse signals, e.g. `at the same time`.
- In case of noun phrase conjunction elided verbs are distributed to each conjunct. Example: `A watch has a button 'A' and [has] a button 'B'.`
- Prepositional phrases in adjunct position always modify the verb, e.g. give additional information about the location of an event. Example: `The watch {shows the current time on the display}.`
- Anaphoric reference is possible via pronouns or definite noun phrases: the antecedent is the most recent suitable noun phrase that agrees in number and gender, e.g. `the watch ⇒ a watch, the button 'A' ⇒ a button 'A'.`
- Definite noun phrases that are not used anaphorically (`the current time`) are treated like indefinite noun phrases, i.e. the existence of at least one object is stated.

Additionally, ACE comes with rules of style helping users to formulate the intended specification.

We claim that

- domain specialists need less effort to learn and to apply the rules of ACE than to cope with an unfamiliar formal language,
- ACE specifications are much easier to understand and to relate to the problem domain than specifications in an manifestly formal notation.

However, notwithstanding its natural character ACE is indeed a *formal* specification language as we will show next.

3 From ACE to Discourse Representation Structures

ACE specifications are analysed and processed deterministically by a unification-based phrase structure grammar enhanced by linearised feature structures written in GULP [Covington 94]. Unification-based grammars are declarative statements of well-formedness conditions and do not pin down a parsing strategy in advance. Prolog's built in top-down recursive-descendent parser would use strict chronological back-tracking to search a proof for an ACE sentence. Top-down parsing is very fast for short sentences but for longer composite ACE sentences the exponential costs of back-tracking can devour speed. It turns out that we can do better using a hybrid top-down chart parser that remembers analysed constituents of an ACE sentence. Actually the chart is only used if it is complete for a particular constituent type in a specific position of an ACE sentence – otherwise the Attempto system parses the sentence conventionally.

Correct understanding of an ACE specification requires not only processing of individual sentences and their constituents, but also taking into account the way sentences are interrelated to express complex propositional structures. It is well-known that aspects such as anaphoric reference, ellipsis and tense cannot be successfully handled without taking the preceding discourse into consideration. Take for example the discourse

```
A watch has a button 'A' and [has] a button 'B'. If the watch shows ...
```

In classical predicate logic the first sentence would be written as

```
(∃X) (∃Y) (∃Z) (watch(X) ∧ button_A(Y) ∧ have(X,Y) ∧ button_B(Z) ∧ have(X,Z))
```

The definite noun phrase `the watch` in the antecedent of the conditional sentence must somehow relate to the indefinite noun phrase `a watch` in the first sentence. But if the two sentences are treated as separate propositions – as predicate logic does – then their variables cannot be under the scope of the same quantifiers. We solve this problem by employing discourse representation theory that resolves anaphoric references in a systematic way combining the two propositions into one [Kamp & Reyle 93].

In our case, ACE sentences are translated into Discourse Representation Theory with Eventualities (DRT-E). DRT-E is a structured form of predicate logic that represents a multisentential specification discourse in a single logical unit called a discourse representation structure (DRS). Eventualities are events or states that are derived from verbal information.

The hybrid chart parser gets the ACE specification in tokenised form from the dialogue component of the Attempto system and generates concurrently a syntax tree as syntactic representation, a discourse representation structure as a formal representation and a paraphrase as feedback to inform the user in ACE how the specification text is being interpreted.

As far as the formal representation is concerned, the whole specification text is translated into one DRS to which each part of an ACE sentence contributes some logical conditions using the preceding sentences as context to resolve anaphoric references and ellipsis. A DRS is represented by a term of the form `drs(U,Con)` where `U` is a list of discourse referents – i.e. quantified variables – and `Con` is a list of conditions – i.e. terms containing the discourse referents as arguments. The discourse referents stand for entities in the real world, while the conditions constitute constraints that the discourse referents must fulfil to make a DRS true. Simple DRS conditions are logical atoms, while complex DRS conditions are built up from other DRSs and have one of the forms `ifthen(DRS1,DRS2)`, `or(DRS1,DRS2,...)`, `not(DRS)`, `ynq(DRS)` and `whq(DRS)` representing conditional sentences, disjunctive constituents, negated constituents, *yes/no*-questions, and *wh*-questions. These complex conditions are associated with function words and query words of ACE and build the logic scaffolding for the construction of the DRS during the parsing process. For the specification example we get the following (pretty-printed) DRS as output:

```
A B C D E
-----------------------------------------------------------------
watch(A)
button(B)
string(B,'A')
state(C,have(A,B))
button(D)
string(D,'B')
state(E,have(A,D))

 F G H I J K                        L M N O
 -------------------------          -------------------------
 current(F)                         stop_watch(L)
 time(F)                            event(M,start(A,L))
 event(G,show(A,F))                 elapsed(N)
 location(on(G,H))           ⇒      time(N)
 display(H)                         of(N,L)
 user(I)                            event(O,show(A,N))
 event(J,press(I,B))                location(on(O,H))
 event(K,press(I,D))
 at_the_same_time(J,K)
```

The first ACE sentence leads to five existentially quantified discourse referents (`[A,B,C,D,E]`) and seven conditions. The coordinated verb phrase `has a button 'A' and [has] a button 'B'` results in two different states (`state(C,have(A,B))` and `state(E,have(A,D))`). The appositions `'A'` and `'B'` are so-called quoted strings that distinguish the two buttons. Quoted strings are translated into predefined DRS conditions of the form `string(B,'A')` and `string(D,'B')` that give further syntactic information about the entities denoted by the variables `B` and `D`.

The second ACE sentence is analysed in the context of the first sentence. Its translation introduces a complex `if-then` DRS with two subordinated DRSs inside. While the new discourse referents (`[F,G,H,...]`) occurring in the `if`-part of the DRS are universally quantified, the new discourse referents (`[L,M,N,...]`) in the `then`-part of the DRS are existentially quantified. In the `if`-part of the DRS we find three events (`event(G,show(A,F))`, `event(J,press(I,B))`, `event(K,press(I,D))`) that have been derived from the verbal information in the sentence. The expression `at the same time` is a predefined discourse signal that makes explicit that two events occur simultaneously (`at_the_same_time(J,K)`). The verb modifying prepositional phrase `on the display` leads to two different conditions (`location(on(G,H))`, `display(H)`) where the first condition is a predefined one indicating the location of the event. The phrase `the elapsed time of the stop-watch` leads to a binary relation (`of(N,L)`) in the `then`-part of the DRS that is not further analysed.

Anaphoric references, e.g. `the watch` ⇒ `a watch`, `the button 'A'` ⇒ `a button 'A'` have been resolved by the resolution algorithm. References are only possible to discourse referents in superordinated DRSs. The resolution algorithm always picks the closest referent that agrees in gender and number with the anaphora.

The DRS can be automatically translated into Prolog. During the translation existentially quantified discourse referents are replaced by Skolem constants or Skolem functions. Conjunctive consequences of an `if-then` DRS lead to sets of Prolog clauses, one clause for each conjunct.

4 Deductions from Discourse Representation Structures

On the basis of a proposal by [Reyle & Gabbay 94] we have developed a correct and complete theorem prover for discourse representation structures. To prove that a discourse representation structure `DRS2` is logically implied by a discourse representation structure `DRS1`

```
DRS1 |- DRS2
```

the theorem prover proceeds in a goal-directed way without any human intervention. In the simplest case an atomic condition of `DRS2` is a member of the list of conditions of `DRS1` – after suitable substitutions. In other cases, the left or the right side of the turn-stile are reformulated and simplified, e.g. we replace

```
L |- (R1 ∨ R2)
```

by

```
L |- R1  ∨  L |- R2
```

or

```
L |- (R1 ⇒ R2)
```

by

```
(L ∪ R1) |- R2
```

This theorem prover forms the kernel of a general deduction system for DRSs derived from ACE specifications and queries. The deduction system answers queries, performs hypothetical reasoning ('What happens if ... ?'), does abductive reasoning ('Under which conditions does ... occur?'), and executes specifications. All interactions with the deduction system are in ACE.

Queries

A specification in a logic language describes a certain state of affairs within a problem domain. We can examine this state of affairs and its logic consequences by querying the specification in ACE. ACE allows two forms of questions

- *yes/no*-questions asking for the existence or non-existence of a fact,
- *wh*-questions (`who`, `what`, `when`, `where`, `how`, etc.) asking for specific details of the state of affairs.

Here is an example of a *wh*-question. Assuming that our watch specification has been translated into the above discourse representation structure `S`, the ACE query sentence

```
What has a button 'A'?
```

is translated into the query DRS `Q`

```
A B C
-----------------------
what(A)
button(B)
string(B,'A')
state(C,have(A,B))
```

and answered by deduction

$S \vdash Q$

Query words – like `what` – are replaced during the proof and answers returned to the user in ACE, i.e.

```
[The watch] has a button 'A'.
```

Hypothetical Reasoning

When a logic specification partially describes a state of affairs there can be various possibilities to extend it. These extensions can lead to diverse logical consequences, some of which are desirable, while others are not. That is, we may want to ask the question 'What if?'.

'What if?' questions mean that we test a particular hypothesis H by deriving its implied consequences C in the context of a logic specification S.

$S \vdash (H \Rightarrow C)$

With the help of the deduction theorem this can be restated as

$S \cup H \vdash C$

i.e. we derive the logical consequences of the conjunction of S and H.

Abductive Reasoning

Once we have devised a logical specification we may want to investigate under which conditions certain phenomena occur. If the conditions are already described by the logic specification we have the situation of a simple query. If the specification does not yet contain the conditions we can extend it by abduction.

Abduction investigates under which conditions A we can derive particular consequences C from the logic specification S.

$S \cup A \vdash C$

In other words, we try to find an explanation A for the phenomenon C.

Executing the Specification

Model-oriented logic specifications build a behavioural model of the future program [Wing 90], and one might be interested in executing this model to demonstrate its behaviour, be it for validation, or for prototyping. Formally, this form of execution is based on the reflexivity of the deduction relation.

$S \vdash S$

The derivation succeeds trivially. However, the derivation can be conducted in a way that the logical and temporal structure of the specification are traced, and users can convince themselves that the specification has the expected behaviour. Furthermore, if predicates have side-effects – i.e. operations that modify the state of the system such as input and output – these side-effects can be made visible during the derivation. The concrete side-effects are defined by interface predicates as part of the execution environment.

Executing the watch specification starts with the trace:

```
state:   A has B
    A:   watch
    B:   button, 'A'

state:   A has D
    A:   watch
    D:   button, 'B'
```

To execute the `if-then` part of the specification the execution environment queries the user about the truth of the preconditions:

```
Does the watch show the current time on the display?
```

Assuming that the user answers positively, the trace continues with

```
event:   A shows F on H
    A:   watch
    F:   current time
    H:   display
```

The rest of the trace follows the same pattern.

Validating a specification can be difficult since users may find it hard to relate its logical consequences to the – possibly implicit or incomplete – requirements. The Attempto system eases this task by expressing the execution trace in the terms of the problem domain.

5 Conclusions

We have developed Attempto Controlled English (ACE) as a specification language that combines the familiarity of natural language with the rigor of formal specification languages. Furthermore, we have implemented the Attempto specification system that allows users to compose, to query and to execute specifications.

Though ACE has reached a degree of maturity many research opportunities remain, e.g.

- extending ACE with plurality,
- adding complementary notations, e.g. graphics for user interfaces,
- structuring large ACE specifications.

6 References

[Balzer 85] R. M. Balzer, A 15 Year Perspective on Automatic Programming, IEEE Transactions Software Engineering, 11(11), pp.1257-1268, 1985

[Le Charlier & Flener 98] B. Le Charlier, P. Flener, Specifications Are Necessarily Informal, or: The Ultimate Myths of Formal Methods, Journal of Systems and Software, Special Issue on Formal Methods Technology Transfer 40(3), pp. 275-296, March 1998

[CLAW 96] Proceedings CLAW 96, First International Workshop on Controlled Language Applications, Centre for Computational Linguistics, Katholieke Universiteit Leuven, Belgium, March 1996

[Covington 94] M. A. Covington, GULP 3.1: An Extension of Prolog for Unification-Based Grammars, Research Report AI-1994-06, Artificial Intelligence Center, University of Georgia, 1994

[Deville 90] Y. Deville, Logic Programming, Systematic Program Development, Addison-Wesley Publishing Company, 1990

[Fuchs 92] N. E. Fuchs, Specifications Are (Preferably) Executable, Software Engineering Journal, 7, 5 (September 1992), pp. 323-334, 1992

[Fuchs et al. 98] N. E. Fuchs, U. Schwertel, R. Schwitter, Attempto Controlled English (ACE), Language Manual, Version 2.0, Institut für Informatik, Universität Zürich, 1998

[Hall 90] A. Hall, Seven Myths of Formal Methods, IEEE Software, 7, 5, pp. 11-19, 1990

[Jackson 95] M. Jackson, Software Requirements & Specifications: A Lexicon of Practice, Principles and Prejudices, Addison-Wesley, 1995

[Kamp & Reyle 93] H. Kamp, U. Reyle, From Discourse to Logic, Introduction to Modeltheoretic Semantics of Natural Language, Formal Logic and Discourse Representation Theory, Studies in Linguistics and Philosophy 42, Kluwer, 1993

[Macias & Pulman 95] B. Macias, S. G. Pulman, A Method for Controlling the Production of Specifications in Natural Language, The Computer Journal, vol. 38, no. 4, pp. 310-318, 1995

[Meyer 85] B. Meyer, On Formalism in Specifications, IEEE Software, vol. 2, no.1, pp. 6-26, 1985

[Reyle & Gabbay 94] U. Reyle, D. M. Gabbay, Direct Deductive Computation on Discourse Representation Structures, Linguistics and Philosophy, 17, August 94, pp. 343-390, 1994

[Schwitter 98] R. Schwitter, Kontrolliertes Englisch für Anforderungsspezifikationen, Dissertation, Universität Zürich, 1998

[Wing 90] J. M. Wing, A Specifiers's Introduction to Formal Methods, IEEE Computer, 23, 9, pp. 8-24, 1990