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

Abstract. The specification language Attempto Controlled English (ACE) is a controlled natural language, i.e. a subset of standard English with a domain-specific vocabulary and a restricted grammar. The restriction of full natural language to a controlled subset is essential for ACE to be suitable for specification purposes. The main goals of this restriction are to reduce ambiguity and vagueness inherent in full natural language and to make ACE computer processable. ACE specifications can be unambiguously translated into logic specification languages, and can be queried and executed. In brief, ACE allows domain specialists to express specifications in familiar natural language and combine this with the rigour of formal specification languages.

1 Introduction

Specifications state properties or constraints that a software system must satisfy to solve a problem [IEEE 91], describe the interface between the problem domain and the software system [Jackson 95], and define the purpose of the software system and its correct use [Le Charlier & Flener 98]. In which language should we express specifications to accommodate these demands?

The answer to this question depends on many factors, particularly on the specifiers and their background. Though many programs are specified by software engineers, often domain specialists — electrical engineers, physicists, economists and other professionals — perform this task. There are even situations where software engineers and knowledge engineers are deliberately replaced by domain specialists since they are the ultimate source of domain knowledge [Businger 94]. One major goal of our research is to make formal specification methods accessible to domain specialists in notations that are familiar to them and that are close to the concepts and terms of their respective application domain.

Traditionally, specifications have been expressed in natural language. Natural language as the fundamental means of human communication needs no extra learning effort, and is easy to use and to understand. Though for particular domains there are more concise notations, natural language can be used to express any problem. However, 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 process [Meyer 85].

Formal specification languages — often 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]. Nevertheless, 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] and by [Deville 90].

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], other authors just as vigorously defend the appropriateness of formal specification methods [Bowen & Hinchey 95a; Bowen & Hinchey 95b]. We, however, are convinced that the advantages of natural and formal specification languages should be and can be combined, specifically to accommodate the needs of domain specialists.

Our starting point lies in the observation 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 [cf. www-uilots.let.ruu.nl/~Controlled-languages]. 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 and a restricted grammar in the form of a small set of construction and interpretation rules [Fuchs et al. 98; Schwitter 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 ACE may seem informal, it is a formal language with the semantics of the underlying logic language. This also means that ACE has to be learned like other formal languages.

There have been several projects with similar aims, but in most cases the subsets of English were not systematically and clearly defined. For example, [Macias & Pulman 95] developed a system which resembles ours with the important difference that their system restricts only the form of composite sentences, but leaves the form of the constituent sentences completely free. As a consequence, the thorny problem of ambiguity remains and has to be resolved by the users after the system has translated the specification into a formal representation.

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 and into other logic languages. In section 4 we outline the semantics of ACE specifications. Section 5 overviews querying and executing specifications. Finally, in section 6 we conclude and address further points of research. The appendix contains a complete example of an ACE specification together with its translation into the logic language of discourse representation structures.

2 From English to Attempto Controlled English

First, we will introduce Attempto Controlled English (ACE) by an example and then summarise its main characteristics. More information can be found in [Fuchs et al. 98; Schwitter 98].

2.1 An Example

Specifications, and other technical texts, written in full natural language tend to be vague, ambiguous, incomplete, or even inconsistent. Take for example the notice posted in London underground trains [Kowalski 90]:

> Press the alarm signal button to alert the driver.
>
> The driver will stop immediately if any part of the train is in a station.
>
> If not, the train will continue to the next station where help can be more easily given.
>
> There is a £50 penalty for improper use.

The notice leaves many assumptions and conclusions implicit. Kowalski shows that clarity can be improved, ambiguity reduced, and assumptions and conclusions made explicit when one follows guidelines for good language use. To do so, Kowalski reformulates the notice in the declarative form of logic programs. Take for instance the first part of the third sentence. Filling in the missing condition referred to by not, Kowalski rewrites it as

> The driver stops the train at the next station if the driver is alerted and not any part of the train is in a station.

Though much improved even this sentence is incomplete since the agent who alerts the driver is still implicit.

To eliminate this and other deficiencies of full natural language we have proposed the specification language Attempto Controlled English (ACE) as a well-defined subset of English. Since ACE is computer-processable we go beyond what [Kowalski 90] achieved — ACE specifications are not only clearer and more complete, but can also be automatically translated into a logic language, be queried and executed.

Here is a version of the complete London underground notice in ACE. The third sentence is the one discussed above. Note that the agent who alerts the driver is now made explicit.

If a passenger presses an alarm signal button then the passenger alerts the driver.

If a passenger alerts the driver of a train and a part of the train is in a station then the driver stops the train immediately.

If a passenger alerts the driver of a train and no part of the train is in a station then the driver stops the train at the next station.

If the driver stops a train in a station then help is available.

If a passenger misuses an alarm signal button then the passenger pays a £50 penalty.

In the appendix you can find this ACE text together with its translation into the logic language of discourse representation structures.

2.2 ACE in a Nutshell

In this section we briefly present the components of ACE, viz. the vocabulary and the construction and interpretation rules.

Vocabulary. The vocabulary of ACE comprises

– predefined function words (e.g. determiners, conjunctions, prepositions),
– user-defined, domain-specific content words (nouns, verbs, adjectives, adverbs).

Users can define content words with the help of a lexical editor that presupposes only basic grammatical knowledge. Alternatively, users can import existing lexica.

Construction Rules. The construction rules define the form of ACE sentences and texts, and state restrictions on the lexical and phrasal level. The construction rules are designed to avoid typical sources of imprecision in full natural language.

ACE Specifications. An ACE specification is a sequence of sentences. There are

– simple sentences,
– composite sentences,
– query sentences.

Simple Sentences. Simple sentences have the form

subject + verb + complements + adjuncts

where complements are necessary for transitive or ditransitive verbs and adjuncts are optional. Here is an example for this sentence form:

The driver stops the train at the station.

Composite Sentences. Composite sentences are built from simpler sentences with the help of predefined constructors:

- coordination (and, or),
- subordination by conditional sentences (if ... then ...),
- subordination by subject and object modifying relative sentences (who, which, that),
- verb phrase negation (does not, is not),
- noun phrase negation (no),
- quantification (a, there is a, every, for every).

Query Sentences. There are

- *yes/no*-queries,
- *wh*-queries.

Yes/no-queries are derived from simple sentences by inverting the subject and the verb be (Is the train in the station?), or by inserting do or does if the verb is not be (Does the driver stop the train?). *Wh*-queries begin with a so-called *wh*-word (who, what, when, where, how, etc.) and contain do or does (Where does the driver stop the train?), unless the query asks for the subject of the sentence (Who stops the train?).

Anaphora. ACE sentences and phrases can be interrelated by anaphora, i.e. by references to previously occurring noun phrases. Anaphora can be personal pronouns or definite noun phrases. In

A passenger of a train alerts a driver. He stops the train.

the personal pronoun he refers to the noun phrase a driver and the definite noun phrase the train refers to the indefinite noun phrase a train.

Coordination. Coordination is possible between sentences and between phrases of the same syntactic type, e.g.

A passenger presses an alarm signal button and the driver stops the train.

A passenger presses an alarm signal button and alerts the driver.

A driver stops a train immediately or at the next station.

Coordination of verbal phrases can be simplified. Instead of

A passenger presses a red button or presses a green button.

we can write

A passenger presses a red button or a green button.

Lexical Restrictions. Examples for lexical restrictions and typographical conventions are:

– verbs are used in the simple present tense, the active voice, the indicative mood, the third person singular (presses),
– no modal verbs (may, can, must etc.) or intensional verbs (hope, know, believe etc.),
– no modal adverbs (possibly, probably etc.),
– ACE allows the user to define synonyms (alarm signal button, alarm button) and abbreviations (ASB standing for alarm signal button),
– content words can be simple (train) or compound (alarm signal button, alarm-signal-button).

Phrasal Restrictions. Examples for phrasal restrictions are:

– complements of full verbs can only be noun phrases (press a button) or prepositional phrases (send the ambulance to the station),
– adjuncts can only be realised as prepositional phrases (in a station) or adverbs (immediately),
– *of*-constructions (the part of the train) are the only allowed postnominal prepositional modifiers.

Interpretation Rules. Interpretation rules control the semantic analysis of grammatically correct ACE sentences. They, for example, resolve ambiguities that cannot be removed by the construction rules. The result of this analysis is reflected in a paraphrase. Important rules are:

– verbs denote events (press) or states (be),
– the textual order of verbs determines the default temporal order of the associated events and states,
– prepositional phrases in adjunct position always modify the verb, e.g. give additional information about the location of an event (The driver {stops the train in a station}.),
– 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,
– noun phrase coordination within a verb phrase is interpreted as coordination reduction; the elided verb is distributed to each conjunct (The driver presses a red button and [presses] a green button.),
– the textual occurrence of a quantifier (a, there is a, every, for every) opens its scope that extends to the end of the sentence; thus any following quantifier is automatically within the scope of the preceding one.

Learning ACE. In contrast to rules of formal languages the construction and interpretation rules of ACE are easy to use and to remember since they are similar to English grammar rules and only presuppose basic grammatical knowledge. One should bear in mind, however, that in spite of its appearance ACE is a formal language that — like other formal languages — must be learned. Companies like Boeing and Caterpillar have been using controlled languages for technical documentation for many years. They report that these languages can be taught in a few days, and that users get competent in a few weeks [CLAW 98]. Thus 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.

3 From ACE to Discourse Representation Structures

ACE sentences are translated into discourse representation structures — a syntactical variant of full first-order predicate logic. We will first discuss the translation process, and then the handling of ambiguity.

3.1 Translation

ACE specifications are analysed and processed deterministically by a unification-based phrase structure grammar enhanced by linearised feature structures written in GULP, a preprocessor for Prolog [Covington 94]. Unification-based grammars are declarative statements of well-formedness conditions and can be combined with any parsing strategy. Prolog's built in top-down recursive-descent parser uses strict chronological back-tracking to parse an ACE sentence. Top-down parsing is very fast for short sentences but for longer composite sentences the exponential costs of backtracking can slow down the parsing. It turns out that we can do better using a hybrid top-down chart parser that remembers analysed phrases of an ACE sentence. Actually, the chart is only used if it is complete for a particular phrase 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 passenger enters a train. The train leaves a station.

In classical predicate logic these two sentences would be represented as two separate formulas

$$\exists X \exists Y (passenger(X) \land train(Y) \land enter(X,Y))$$
$$\exists U \exists W (train(U) \land station(W) \land leave(U,W))$$

This representation fails to relate the anaphoric reference of the definite noun phrase the train in the second sentence to the indefinite noun phrase a train in the first sentence. 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 extended by events and states (DRT-E) [cf. Kamp & Reyle 93, Parsons 94]. DRT-E is a variant of predicate logic that represents a multisentential text as a single logical unit called a discourse representation structure (DRS). Each part of an ACE sentence contributes some logical conditions to the DRS using the preceding sentences as context to resolve anaphoric references. A DRS is represented by a term `drs(U,Con)` where `U` is a list of discourse referents and `Con` is a list of conditions for the discourse referents. The discourse referents are quantified variables that stand for objects in the specified domain, while the conditions constitute constraints that the discourse referents must fulfil to make the DRS true. Simple DRS conditions are logical atoms, while complex DRS conditions are built up recursively from other DRSs and have the following forms *ifthen(DRS1,DRS2)*, *or(DRS1,DRS2, . . .)*, *not(DRS)*, *ynq(DRS)* and *whq(DRS)* representing conditional sentences, disjunctive phrases, negated phrases, *yes/no*-queries, and *wh*-queries.

The translation of the two ACE sentences

A passenger enters a train. The train leaves a station.

generates the DRS

```
[A,B,C,D,E]
passenger(A)
train(B)
event(C,enter(A,B))
station(D)
event(E,leave(B,D))
```

The first ACE sentence leads to three existentially quantified discourse referents (`[A,B,C]`) and the first three conditions of the DRS. The second ACE sentence is analysed in the context of the first sentence. It contributes two further discourse referents (`[D,E]`) and the fourth and the fifth condition of the DRS. The two event conditions have been derived from lexical information that classifies the verbs enter and leave as being associated with events.

Analysing the second sentence in the context of the first one allows the Attempto system to resolve the train as anaphoric reference to a train. The search space for antecedents of anaphora is defined by an accessibility relation among nested DRSs. A discourse referent is accessible from a DRS D if the discourse referent is in D, in a DRS enclosing D, in a disjunct that precedes D in an *or*-DRS, or in the antecedent of an *ifthen*-DRS with D as consequent. The resolution algorithm always picks the closest accessible referent that agrees in gender and number with the anaphor.

Here is a more complex example from the ACE version of the underground notice. The sentence

If a passenger alerts a driver of a train then the driver stops the train in a station.

is translated into the DRS

```
[]
IF
     [A,B,C,D]
     passenger(A)
     driver(B)
     train(C)
     of(B,C)
     event(D,alert(A,B))
THEN
     [E,F]
     station(E)
     event(F,stop(B,C))
     location(F,in(E))
```

The outermost DRS has an empty list of discourse referents and an *ifthen*-DRS as condition. Both the *if*-part and the *then*-part are DRSs. The discourse referents (`[A,B,C,D]`) occurring in the *if*-part of the DRS are universally quantified, while the discourse referents (`[E,F]`) in the *then*-part of the DRS are existentially quantified. The prepositional phrase in a station leads to the condition `station(E)` and to the predefined condition `location(F,in(E))` that indicates the location of the event `F`. The Attempto system resolves the driver and the train as anaphoric references to a driver and to a train, respectively.

A DRS can be automatically translated into the standard form of first-order predicate logic and into clausal form. ACE sentences without disjunctive consequences lead to Prolog programs.

3.2 Constraining Ambiguity

Ambiguity is the most prevalent problem when natural language is processed by a computer. Though ambiguity occurs on all levels of natural language, here we will only discuss a case of syntactical ambiguity. Take the three sentences

The driver stops the train with the smashed head-light.

The driver stops the train with great effort.

The driver stops the train with the defect brake.

Note that all three sentences have the same grammatical structure, and that all three sentences are syntactically ambiguous since the prepositional phrase with... can be attached to the verb stops or to the noun train. A human reader will perceive the first and the second sentence as unambiguous since each sentence has only one plausible interpretation according to common sense, while the third sentence allows two plausible interpretations since with the defect brake can refer to stops or to train. For this sentence additional contextual knowledge is necessary to select the intended interpretation. Altogether, we find that humans can disambiguate the three sentences with the help of contextual, i.e. non-syntactical, knowledge.

Standard approaches to handle ambiguity by a computer rely on *Generate and Test* or on *Underspecified Representations.*

Generate and Test. The traditional account is first to generate all possible interpretations, and then to eliminate those which are not plausible. The elimination of implausible interpretations can be done by presenting all interpretations to the user who selects the intended one [e.g. Macias & Pulman 95], or by formalising relevant contextual knowledge and automatically selecting the most fitting interpretation on the basis of this knowledge [e.g. Hindle & Rooth 93]. *Generate and Test* suffers from several shortcomings: it is inefficient and can lead to the combinatorial explosion of interpretations; manually selecting one of many interpretations is a burden on the user; formalising relevant contextual knowledge for the automatic disambiguation is difficult.

Underspecified Representations. A sentence gets just one semantic representation based on its syntactic form leaving certain aspects of the meaning unspecified [Alshawi 92; Reyle 93]. Fully specified interpretations are obtained by filling in material from formalised contextual knowledge. This approach has a drawback that we encountered already: it is difficult to formalise rules that lead to more specific interpretations because complex contextual effects — world knowledge, linguistic context, lexical semantics of words etc. — play a crucial role in the human disambiguation process.

In approaches based on *Generate and Test* or on *Underspecified Representations* disambiguation depends in one way or other on context. Thus the same syntactic construct could get different interpretations in different contexts which are perhaps only vaguely defined [Hirst 97]. This may be desirable for the processing of unrestricted natural language but is highly problematic for specifications. Writing specifications is already very hard. If on top of this the specification language allowed context-dependent interpretations then writing, and particularly reading, specifications would indeed be very difficult, if not entirely impossible.

To avoid this problem, ACE resolves ambiguity by a completely syntactical approach without any recourse to the context.

More concretely, ACE employs three simple means to resolve ambiguity:

– some ambiguous constructs are not part of the language; unambiguous alternatives are available in their place,
– all remaining ambiguous constructs are interpreted deterministically on the basis of a small number of interpretation rules that use syntactic information only; the interpretations are reflected in a paraphrase,
– users can either accept the assigned interpretation, or they must rephrase the input to obtain another one.

For the third example sentence

The driver stops the train with the defect brake.

the Attempto system would generate the paraphrase

The driver {stops the train with the defect brake}.

that reflects ACE's interpretation rule that a prepositional phrase always modifies the verb. This interpretation is probably not the one intended by the user. To obtain the other interpretation the user can reformulate the sentence employing the interpretation rule that a relative sentence always modifies the immediately preceding noun phrase, e.g.

The driver stops the train that has a defect brake.

yielding the paraphrase

The driver stops {the train that has a defect brake}.

Altogether, ACE has just over a dozen interpretation rules to handle ambiguity.

4 Semantics of ACE Specifications

In this chapter we introduce the model-theoretic interpretation of ACE, and explain ACE's model of time, events and states.

4.1 Model-Theoretic Interpretation

A discourse representation structure is a syntactic variant of full first-order predicate logic and thus allows for the usual model-theoretic interpretation. According to [Kamp & Reyle 93] any interpretation of a DRS is also an interpretation of the text from which the DRS was derived. Concretely, we have the following definition:

> Let D be the DRS derived from the set S of ACE sentences with the vocabulary V, and M be a model of V. Then S is true in M iff D is true in M.

Thus we can associate meaning to ACE specifications in a way that is completely analogous to the model-theoretic interpretation of logic formulas. We can give each ACE sentence a truth value, i.e. we call a sentence true if we can interpret it as describing an actual state of affairs in the specified domain, or we label the sentence false if we cannot establish such an interpretation.

We can interpret simple sentences as descriptions of distinct events or states. The sentence

A driver stops a train.

is true if the word driver can be interpreted as a relation *driver* that holds for an object A of the specified domain, the word train as a relation *train* that holds for an object B of the specified domain, and the word stop as a relation *stopping event* that holds between A and B. Otherwise, the sentence is false.

Once we have assigned meaning to simple sentences we can give meaning to composite sentences. Again, this is completely analogous to classical model theory. A conjunction of sentences is true if all conjoined sentences are true. A disjunction of sentences is true if at least one of the sentences of the disjunction is true. A sentence with a negation is true if the sentence without the negation is false. An *ifthen*-sentence is only false if the *if*-part is true and the *then*-part is false; otherwise, the *ifthen*-sentence is true. A sentence with a universal quantifier is true if the sentence without the universal quantifier is true for all objects denoted by the quantified phrase.

4.2 Events and States

Attempto Controlled English has a model of time, events and states that closely resembles that one of the event calculus [Sadri & Kowalski 95].

Each verb in a sentence denotes an event or a state. Events occur instantaneously, while states — i.e. relations that hold or do not hold — last over a time period until they are explicitly terminated.

Each occurrence of a verb is implicitly associated with a time. If the verb denotes an event this is the time point at which the event occurs; if the verb denotes a state this is the time point from which on the state holds.

Per default the textual order of verbs establishes the relative temporal order of their associated events or states. E.g. in

A passenger alerts a driver. The driver stops a train. The train is in a station.

the event alert is temporally followed by the event stop which is temporally followed by the onset of the state be in a station. Our default assumption "textual order = temporal order" is supported by psychological and physiological evidence [Münte et al. 98].

Users can override the default temporal order by explicitly specifying times through prepositional phrases like at 9 o'clock, and by adding prepositional phrases like at the same time, or in any temporal order. A future version of ACE will allow users to combine sentences with the help of temporal prepositions like before, after and while.

5 Deductions from Discourse Representation Structures

We describe how deduction from discourse representation structures can be used to answer queries, to perform hypothetical and abductive reasoning, and to execute a specification. Furthermore, we briefly indicate how users can define domain theories and ontologies.

5.1 Theorem Prover for 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* can be derived from a discourse representation structure *DRS1*

$$DRS1 \vdash DRS2$$

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

$$L \vdash (R1 \vee R2) \quad \text{by} \quad L \vdash R1 \text{ and } L \vdash R2$$

or

$$L \vdash (R1 \Rightarrow R2) \quad \text{by} \quad (L \cup R1) \vdash R2$$

This theorem prover will form the kernel of a general deduction system for DRSs. The deduction system will answer queries, perform hypothetical reasoning ('What happens if ... ?'), do abductive reasoning ('Under which conditions does ... occur?'), and execute specifications. All interactions with the deduction system will be in ACE.

5.2 Query Answering

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

- *yes/no*-queries asking for the existence or non-existence of the state of affairs defined by the ACE specification,
- *wh*-queries (who, what, when, where, how, etc.) asking for specific details of the state of affairs described by the ACE specification.

Here is an example of a *wh*-query. Once we have translated the ACE sentence

A passenger enters a train.

into the DRS *S*

```
[A,B,C]
passenger(A)
train(B)
event(C,enter(A,B))
```

we can ask

Who enters a train?

The query sentence is translated into the DRS Q

```
 []
 WHQ
     [A,B,C]
     who(A)
     train(B)
     event(C,enter(A,B))
```

and answered by deduction

$$S \vdash Q$$

Query words — like who — are replaced during the proof and answers are returned to the user in ACE, i.e.

[A passenger] enters a train.

5.3 Hypothetical Reasoning

When a logic specification partially describes a state of affairs there can be various possibilities to extend the specification. 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 happens if ...?'.

'What happens if ...?' questions mean that we test a particular hypothesis H by examining its implied consequence C in the context of a given 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 consequence C of the union of S and H.

Here is a simple example shown on the level of ACE. If we extend the specification

If a passenger alerts a driver then the driver stops a train.

by the sentence

A passenger alerts a driver.

then we can deduce the ACE sentence

A driver stops a train.

as the logical consequence of the specification and its extension.

5.4 Abductive Reasoning

Once we have devised a logic specification we may want to investigate under which conditions a certain state of affairs occurs. If the conditions are already described by the logic specification we have the situation of a simple query. However, if the specification does not yet contain the pre-conditions of the state of affairs we are interested in then the question 'Under which conditions does ... occur?' can lead to abductive extensions of the specification.

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

$$(S \cup A) \vdash C$$

Again a simple example. Given the specification

If a passenger alerts a driver then the driver stops a train.

If a train arrives at a station then a driver stops the train.

we want to know under which conditions the state of affairs occurs that is described by the sentence

A driver stops a train.

Abduction will give us first the ACE sentence

A passenger alerts a driver.

and then the ACE sentence

A train arrives at a station.

as two possible conditions.

5.5 Executing an ACE Specification

Model-oriented logic specifications build a behavioural model of the program to be developed [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, it can be conducted in a way that the logical and the temporal structure of the specification are traced, and that 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 realised by the execution environment.

Executing the ACE specification

A passenger alerts the driver. The driver stops the train. The train is in the station.

leads to the execution trace:

```
event:  A alerts B
    A:  passenger
    B:  driver

event:  B stops D
    D:  train

state:  D is in F
    F:  station
```

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. This not only reduces the semantic distance between the concepts of the application domain and the specification but also increases the efficiency of the validation process.

5.6 Domain Knowledge

The Attempto system is not associated with any specific application domain, nor with any particular software engineering method. By itself it does not contain any knowledge or ontology of application domains, of software engineering methods, or of the world in general. Thus users must explicitly define domain knowledge. Currently, this is possible with the help of ACE sentences like

Waterloo is a station.

Every train has a driver.

Even constraints can be expressed. If a specification states in one place that a train is out of order, and in another place that at the same time the same train is available, the contradiction can be detected if we explicitly define that out of order and available exclude each other, e.g.

No train is out of order and available at the same time.

In the future, ACE will provide meta-statements like

Define a train as a vehicle.

which will allow users to define domain theories and ontologies more concisely. With other meta-statements users will be able to specify constraints, safety properties, and exceptions.

6 Conclusions

We have developed Attempto Controlled English (ACE) as a specification language that combines the familiarity of natural language with the rigour of formal specification languages. Furthermore, we have implemented the Attempto specification system that allows domain specialists with little or no knowledge of formal specification methods to compose, to query and to execute formal specifications using only the concepts and the terms of their respective application domain.

The Attempto system translates ACE specifications into discourse representation structures (DRSs). Being a variant of first-order predicate logic, DRSs can readily be translated into a broad range of other representations. While the Attempto system comprises an automatic translation of DRSs into first-order predicate logic, clausal logic and Prolog, other DRSs were manually translated into the input language of the Finfimo theorem prover [Bry et al. 98]. This means that ACE is not only a specification language but also a convenient means to express theorems, integrity constraints and rules.

Currently, we are extending ACE with plurality and with complementary notations for graphical user interfaces and algorithms. Furthermore, we are investigating ways how to structure large ACE specifications.

Acknowledgements

We would like to thank Pierre Flener, Alberto Pettorossi, Julian Richardson, Dave Robertson and other participants of LOPSTR'98 for fruitful discussions and many constructive remarks. Also, we would like to express our gratitude to the anonymous reviewers for their helpful comments.

References

[Alshawi 92] H. Alshawi, The Core Language Engine, MIT Press, 1992

[Balzer 85] R. M. Balzer, A 15 Year Perspective on Automatic Programming, IEEE Transactions Software Engineering, vol. 11, no. 11, pp. 1257-1268, 1985

[Bowen & Hinchey 95a] J. P. Bowen, M. G. Hinchey, Seven More Myths of Formal Methods, IEEE Software, July 1995, pp. 34-41, 1995

[Bowen & Hinchey 95b] J. P. Bowen, M. G. Hinchey, Ten Commandments of Formal Methods, IEEE Computer, vol. 28, no. 4, pp. 56-63, 1995

[Bry et al. 98] F. Bry, N. Eisinger, H. Schütz, S. Torge, SIC: Satisfiability Checking for Integrity Constraints, Research Report PMS-FB-1998-3, Institut für Informatik, Universität München, 1998

[Businger 94] A. Businger, Expert Systems for the Configuration of Elevators at Schindler AG, Talk at Department of Computer Science, University of Zurich, July 1994

[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, vol. 40, no. 3, pp. 275-296, March 1998

[CLAW 98] Second International Workshop on Controlled Language Applications CLAW'98, Carnegie Mellon University, 21-22 May 1998

[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, 1990

[Fuchs 92] N. E. Fuchs, Specifications Are (Preferably) Executable, Software Engineering Journal, vol. 7, no. 5 (September 1992), pp. 323-334, 1992; reprinted in: J. P. Bowen, M. G. Hinchey, High-Integrity System Specification and Design, Springer-Verlag London Ltd., 1999 (to appear)

[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, vol. 7, no. 5, pp. 11-19, 1990

[Hindle & Rooth 93] D. Hindle, M. Rooth, Structural Ambiguity and Lexical Relations, Computational Linguistics, vol. 19, no. 1, pp. 103-120, 1993

[Hirst 97] G. Hirst, Context as a Spurious Concept, AAAI Fall Symposium on Context in Knowledge Representation and Natural Language, Cambridge, Mass., 8 November 1997

[IEEE 91] IEEE Standard Glossary of Software Engineering Terminology, Corrected Edition, February 1991 (IEEE Std 610.12-1990)

[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

[Kowalski 90] R. Kowalski, English as a Logic Programming Language, New Generation Computing, no. 8, pp. 91-93, 1990

[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

[Münte et al. 98] T. F. Münte, K. Schiltz, M. Kutas, When temporal terms belie conceptual order, Nature, no. 395, p. 71, 1998

[Parsons 94] T. Parsons, Events in the Semantics of English: A Study in Subatomic Semantics, Current Studies in Linguistics, MIT Press, 1994

[Reyle 93] U. Reyle, Dealing with Ambiguities by Underspecification: Construction, Representation and Deduction, Journal of Semantics, 10, pp. 123-178, 1993

[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

[Sadri & Kowalski 95] F. Sadri, R. Kowalski, Variants of the Event Calculus, in: L. Sterling (ed.), Proc. ICLP'95, 12th International Conference on Logic Programming, MIT Press, pp. 67-82, 1995

[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, vol. 23, no. 9, pp. 8-24, 1990

Appendix

Original Version of the London Underground Notice

Press the alarm signal button to alert the driver.
The driver will stop immediately if any part of the train is in a station.
If not, the train will continue to the next station where help can be more easily given.
There is a £50 penalty for improper use.

ACE Version with Sentence by Sentence Translation into DRSs

If a passenger presses the alarm signal button then the passenger alerts the driver.

```
[]
IF
   [A,B,C]
   passenger(A)
   alarm_signal_button(B)
   event(C,press(A,B))
THEN
   [D,E]
   driver(D)
   event(E,alert(A,D))
```

If a passenger alerts the driver of a train and a part of the train is in a station then the driver stops the train immediately.

```
[]
IF
   [A,B,C,D,E,F,G]
   passenger(A)
   driver(B)
   train(C)
   of(B,C)
   event(D,alert(A,B))
   part(E)
   of(E,C)
   station(F)
   state(G,be(E))
   location(G,in(F))
THEN
   [H]
   event(H,stop(B,C))
   manner(H,immediately)
```

If a passenger alerts the driver of a train and no part of the train is in a station then the driver stops the train at the next station.

```
[]
IF
   [A,B,C]
   passenger(A)
   driver(B)
```

```
      train(C)
      of(B,C)
      event(D,alert(A,B))
      IF
         [E]
         part(E)
         of(E,C)
      THEN
         []
         NOT
            [F,G]
            station(F)
            state(G,be(E))
            location(G,in(F))
   THEN
      [H,I]
      next(H)
      station(H)
      event(I,stop(B,C))
      location(I,at(H))
```

If the driver stops the train in a station then help is available.

```
   []
   IF
      [A,B,C,D]
      driver(A)
      train(B)
      event(C,stop(A,B))
      station(D)
      location(C,in(D))
   THEN
      [E,F]
      help(E)
      state(F,available(E))
```

If a passenger misuses the alarm signal button then the passenger pays a £50 penalty.

```
   []
   IF
      [A,B,C]
      passenger(A)
      alarm_signal_button(B)
      event(C,misuse(A,B))
   THEN
      [D,E]
      £_50_penalty(D)
      event(E,pay(A,D))
```