TECHNICAL REPORT 94.17, DEPARTMENT OF COMPUTER SCIENCE, UNIVERSITY OF ZURICH, NOVEMBER 1994

Specifying Logic Programs in Controlled Natural Language

Norbert E. Fuchs, Hubert F. Hofmann, Rolf Schwitter
Department of Computer Science, University of Zurich

Abstract

Writing specifications for computer programs is not easy since one has to take into account the disparate conceptual worlds of the application domain and of software development. To bridge this conceptual gap we propose controlled natural language as a declarative and application-specific specification language. Controlled natural language is a subset of natural language that can be accurately and efficiently processed by a computer, but is expressive enough to allow natural usage by non-specialists. Specifications in controlled natural language are automatically translated into Prolog clauses, hence become formal and executable. The translation uses a definite clause grammar enhanced by feature structures. Inter-text references of the specification, e.g. anaphora, are resolved by discourse representation theory. The generated Prolog clauses are added to a knowledge base, and furthermore provide the input of a concept lexicon. We have implemented a prototypical specification system that successfully processes the greater part of the specification of a simple automated teller machine.

Table of Contents

1 Declarative Specifications

Program development means that informal and formal knowledge of an application domain is ultimately formalised as a program. To cope with the large conceptual gap between the world of the application specialist and the world of the software developer this formalisation process is usually divided into several intermediate steps associated with different representations of the relevant knowledge. In the context of this report we are mainly interested in two of these representations: requirements and specifications.

By requirements we understand a representation of the problem to be solved by a program. Requirements may come from different sources and express disparate viewpoints. They are often implicit, and may have to be elicited in a knowledge acquisition process. Consequently requirements tend to be informal, vague, contradictory, incomplete, and ambiguous.

From the requirements we derive specifications as a first statement of a solution to the problem at hand, of the services that the intended program will provide to its users. Specifications as an agreement between the parties involved should be explicit, concrete, consistent, complete, and unambiguous. Furthermore, we demand that specifications be formal.

The derivation of formal specifications from informal requirements is difficult, and known to be crucial for the subsequent software development process. The specification process itself cannot be formalised, neither are there formal methods to validate the specifications with respect to the requirements [Hoare 87]. Nevertheless, the process can be made easier by the choice of a specification language that allows us to express the concepts of the application domain concisely and directly, and to convince ourselves of the adequacy of the specification without undue difficulty. Furthermore, we wish to support the specification process by computer tools. Which specification languages fulfil these prerequisites?

Since we want to develop logic programs, specifically Prolog programs, it is only natural that we consider Prolog itself as a first candidate. Though Prolog has been recommended as a suitable specification language [Kowalski 85, Sterling 94] and has often been used as such, application-specific specification languages seem to be a better choice since they allow us to express the concepts of the application domain directly, and still can be mapped to Prolog [Sterling 92]. By making "true statements about the intended domain of discourse" [Kramer & Mylopoulos 92] and "expressing basic concepts directly, without encoding, taking the objects of the language [of the domain of discourse] as abstract entities" [Börger & Rosenzweig 94], application-specific specification languages are – in the original sense of the word – declarative, and have all the practical advantages of declarative programming [Lloyd 94]. Specifically, they are understood by application specialists.

In a previous phase of our project we have shown that graphical and textual views of logic programs can be considered as application-specific specification languages [Fuchs & Fromherz 94]. Each view has an associated editor, and between a program and its views there is an automatic bi-directional mapping. Both these features lead to the following important consequences.

- With the help of the view editors we can compose programs in application-specific concepts.

- The bi-directional mapping of a view to a program in a logic language assigns a formal semantics to the view. Thus, though views give the impression of being informal, they are in fact formal and have the same semantics as their associated program.
- The executability of the logic program and the semantics-preserving mapping between a program and its views enable us to simulate the execution of the program on the level of the views. Thus validation and prototyping in concepts close to the application domain become possible.
- Providing semantically equivalent representations, we can reduce the gap between the different conceptual worlds of the application domain specialist and the software developer.
- The dual-faced informal/formal appearance of the views provides an effective solution for the critical transition from informal to formal representations.

Altogether, the characteristics of the views induce us to call them specifications of the program. Furthermore, since the views are semantically equivalent to the (executable) program, they can even be considered as executable specifications.

Natural language (NL) has a long tradition as a specification language though it is well-known that the advantages of using NL, e.g. its familiarity, are normally outweighed by its disadvantages, e.g. its vagueness and ambiguity. Nevertheless, NL being the standard means of communication between the persons involved in software development, it is only tempting to use NL as specification language in a way that keeps most of its advantages and eliminates most of the disadvantages.

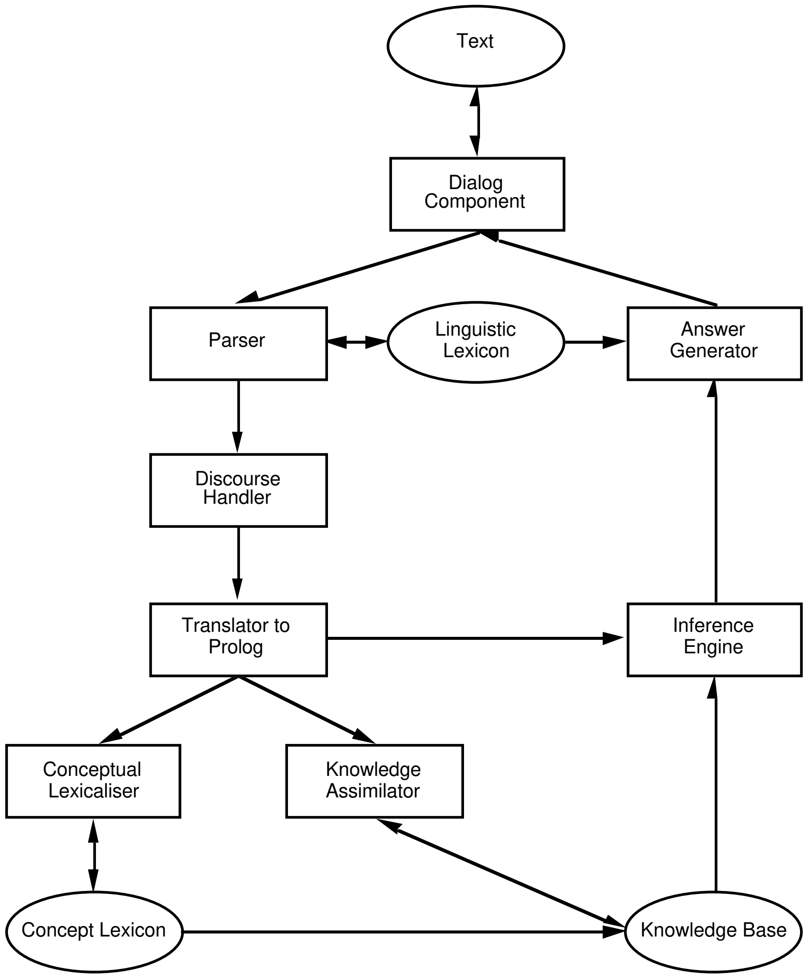
In this report we describe an approach using controlled natural language as a view of a logic program. Users – application specialists and software developers – compose specifications for logic programs in controlled natural language that is automatically translated into Prolog. As pointed out above, this translation makes seemingly informal specifications in controlled natural language formal, and gives them the combined advantages of informality and formality.

We have been developing a system for the specification of logic programs in controlled natural language. In our approach, we assume a strict division of work between a user and the system. Users are always in charge, they are the sole source of domain knowledge, and take all design decisions. The system does not initially contain any domain knowledge besides the vocabulary of the application domain, and plays only the rule of a diligent clerk, e.g. checking specifications for consistency.

Seen from the standpoint of a user our specification system offers the following functionality. The user enters interactively specification text in controlled natural language (cf. section 3) which is parsed (cf. section 4), analysed for discourse references (cf. section 5), and translated into Prolog clauses (cf. section 5). The Prolog clauses are added to a knowledge base, moreover they provide the input for a concept lexicon (cf. section 6). The user can query the knowledge base. Answers are returned to the user in restricted natural language.

2 Overview of the Specification System

In this section we will briefly describe the components of the specification system, and indicate their implementation status. The subsequent sections contain detailed descriptions of the implemented components and their underlying mechanisms.



The *dialog component* is the single interface for the dialog between the user and the specification system. The user enters specification text in controlled natural language which the dialog component forwards to the parser in tokenised form. Parsing errors and ambiguities to be resolved by the user are reported back by the dialog component. The user can also query the knowledge base in controlled natural language. Answers to queries are formulated by the answer generator and forwarded to the dialog component. Finally, the user can use the dialog component to call tools like the editor of the linguistic lexicon. (Status: only text input is possible)

The *parser* uses a predefined definite clause grammar with feature structures and a predefined linguistic lexicon to check sentences for syntactical correctness, and to generate syntax trees and sets of nested discourse representation structures. The linguistic lexicon can be modified by an editor callable from the dialog component. This editor will be called automatically when the parser finds an undefined word. (Status: parser functional; editor of linguistic lexicon still missing)

The *discourse handler* analyses and resolves inter-text references and updates the discourse representation structures generated by the parser. (Status: fully functional; linguistic coverage needs to be extended)

The *translator* translates discourse representation structures into Prolog clauses. These Prolog clauses are either passed to the conceptual lexicaliser and the knowledge assimilator, or – in case of queries – to the inference engine. (Status: translation fully functional; linguistic coverage needs to be extended)

The *conceptual lexicaliser* uses the generated Prolog clauses to build a concept lexicon that contains the application domain knowledge collected so far. This knowledge is used by the inference engine to answer queries of the user, and by the parser to resolve ambiguities. An editor allows the user to inspect and modify the contents of the concept lexicon. (Status: conceptual knowledge is added to the concept lexicon without taking into account the already accumulated knowledge; the knowledge is currently not being used by the other components; editor is missing)

The *knowledge assimilator* adds new knowledge to the knowledge base in a way that avoids inconsistency and redundancy (cf. section 7.3). (Status: not yet implemented)

The *inference engine* answers user queries with the help of the knowledge in the knowledge base and the concept lexicon. Since both the knowledge base and the queries will be expressed in Prolog, the initial version of the inference engine will simply apply Prolog's inference strategy. (Status: not yet implemented, but inferences are possible via the Prolog programming environment)

The *answer generator* takes the answers of the inference engine and reformulates them in restricted natural language. By accessing the linguistic lexicon, the conceptual lexicon and the knowledge base the answer generator uses the terminology of the application domain (cf. section 7.4). (Status: not yet implemented)

3 Controlled Natural Language

A software specification is a statement of the services a software system is expected to provide to its users, and should be written in a concise way that is understandable by potential users of the system, by management and by software suppliers [Sommerville 92]. Strangely enough, this goal is hard to achieve if specifications are expressed in full natural language. Natural language terminology tends to be ambiguous, imprecise and unclear. Also, there is considerable room for errors and misunderstandings since people may have different views of the role of the software system. Furthermore, requirements vary and new requirements arise so that the specification is subject to frequent change. All these factors can lead to incomplete and inconsistent specifications that are difficult to validate against the requirements.

People have advocated the use of formal specification languages to eliminate some of the problems associated with natural language. Because of the need of comprehensibility, we cannot replace documents written in natural language by formal specifications in all cases. Many clients would not understand such a formal document and would hardly accept it as a contract for the software system.

Though it seems that we are stuck between the over-flexibility of natural language and the potential incomprehensibility of formal languages, there is a way out. To improve the quality of specifications without loosing their readability, it is important to establish a context where natural language is used in a controlled way. Controlled natural language enforces writing standards that limit the grammar and vocabulary, and leads to texts containing more predictable, less ambiguous language.

Controlled natural language can also help to find an agreement about the correct interpretation of a specification. When readers and writers are guided, for instance, to use the same word for the same concept in a consistent way then misunderstandings can be reduced. This is of utmost importance because a software specification will be read, interpreted, criticised, and rewritten, again and again until a result is produced that is satisfactory to all participants.

Controlled languages are neither unnatural, nor new, as the following examples from various fields show. In aerospace industry a notation was developed that relies on using natural language in a controlled way for the preparation of aircraft maintenance documentation [AECMA 85]. Epstein used syntactically restricted natural language as data base query language [Epstein 85]. Another well-known example for controlled language is legislation. This case is especially relevant for our approach since it was shown that the language of legislation has many similarities with the language of logic programming, and that statutes can easily be translated into Prolog clauses [Kowalski 90, Kowalski 92]. Finally, LPA's Prolog-based flex tool kit represents rules and frames for expert systems in the Knowledge Specification Language KSL – an English-like notation enhanced by mathematical and control expressions [Vasey 89].

Thus we propose to restrict the use of natural language in specifications to a controlled subset with a well-defined syntax and semantics. On the one hand this subset should be expressive enough to allow natural usage by non-specialists, and on the other hand the language should be accurately and efficiently processable by a computer. This means that we have to find the right trade-off between expressiveness and processability [Pulman 94].

In our approach controlled natural language specifications are translated into semantically equivalent Prolog clauses. This dual representation of specifications narrows the gap between full natural language and formal specification languages and gives us most of the benefits of both. The translation of the specification into a formal language can help to uncover omissions and inconsistencies. This point is important because human language, even when it is used unambiguously, has the tendency to leave assumptions and conclusions implicit, whereas a computer language forces them to be explicit.

Taking into account expressiveness, computer-processability, and the eventual translation into Prolog clauses, we suggest that the basic model of controlled natural language should cover the following constructions:

- simple declarative sentences of the form subject – predicate – object
- relative clauses, both subject and object modifying
- comparative constructions like *bigger than*, *smaller than* and *equal to*
- compound sentences like and-lists, or-lists, and-then-lists
- sentence patterns like *if ... then*
- negation like *does not, is not* and *has not*

This language overlaps with the *computer-processable natural language* proposed by Pulman and his collaborators [Macias & Pulman 92, Pulman 94].

Habitability – i.e. the ability to construct sentences in controlled natural language, and to avoid constructions that fall outside the bounds of the language – seems to be achievable, particularly when the system gives feedback to its users [Epstein 85, Capindale & Crawford 89]. However, we are convinced that employing controlled natural language for specifications will be only successful when users are trained and willing to strive for clear writing. Here we present some short guidelines – patterned after standard prescriptions for good style [e.g. Williams 85] – that can help to convey the model of controlled natural language to its users:

- use active rather than passive tense

- use grammatically correct and simple constructs
- be precise and define the terms that you use
- break up long sentences with several different facts
- avoid ellipsis and idioms
- avoid adverbs, subjunctive and modality
- be relevant, not verbose
- keep paragraphs short and modular
- distinguish functional and non-functional requirements, system goals and design information

A specification in controlled natural language is written in outline format, i.e. it has a hierarchical structure of paragraphs. Each paragraph consists of specification text adorned with comments. Comments can be used to express non-functional specifications, design instructions etc. The specification text will be processed by the system – i.e. ultimately translated into Prolog clauses – while comments remain unchanged. Textual references, e.g. anaphora, are resolved inside a paragraph, or in superordinate paragraphs.

The quality of the software specification is as important as program quality. As with programs, the specification should be designed so that changeability is achieved by minimising external references and making paragraphs as modular as possible.

Also, modularity of paragraphs is important for understanding the specification text and for reuse of parts of a specifications in other specification contexts. Eventually, a repository of predefined paragraphs will be provided which can be used like a card-index.

4 Unification-Based Grammar Formalisms (UBGs)

Unification-based grammar formalisms have attracted considerable attention as a common method in computational linguistic during the last few years. Historically, these formalisms are the result of several independently initiated strains of research that have converged on the idea of unification. On the one hand linguists have developed their theories following the unification-based approach to grammar [Kaplan & Bresnan 82, Gazdar et al. 85, Pollard & Sag 94]. On the other hand the logic programming community created very general formalisms that were intended as tools for implementing grammars of different style [Pereira & Warren 80, Shieber et al. 83, Dörre 91, Covington 94a].

Grammar formalisms are metalanguages whose intended use is to describe a set of well-formed sentences in an object language. The choice of this metalanguage for natural language processing is critical and should fulfil three important criteria: linguistic felicity, expressiveness, and computational effectiveness [Shieber 86].

First, linguists need notations that allow them to encode their linguistic descriptions concisely and flexibly, and to express the relevant generalisations over rules and lexical entries. For that purpose, unification-based grammars (UBGs) are advantageous since they describe abstract relations between sentences and informational structures in a purely declarative manner.

Second, UBGs use context-free grammar rules in which nonterminal symbols are augmented by sets of features. A careful addition of features increases the power of the grammar and results in a class of languages often described as indexed grammars [Gazdar & Mellish 89].

Third, we want machines to be able to understand and employ the formalism in realistic amounts of time. In our approach computational effectiveness is achieved by translating feature structures into Prolog terms that unify in the desired way.

4.1 Definite-Clause Grammars (DCGs)

To parse a natural language sentence is to determine, whether the sentence is generated by a particular grammar and what kind of structure the grammar assigns to the sentence. Thus a parser has two inputs – a grammar and a sentence to be parsed – and one or more outputs representing the syntactic and semantic structures of the parsed sentence.

Prolog provides a special syntactic notation for grammars – the so-called Definite-Clause Grammars format – that we use to implement the parser of our system.

A definite-clause grammar (DCG) is not itself a theory of grammar but rather a special syntactic notation in which linguistic theories of grammars can be expressed. The Prolog DCG-notation allows context-free phrase-structure rules to be stated directly in Prolog. For instance, we can write productions quite naturally as

```
sentence    --> noun_phrase, verb_phrase.
noun_phrase --> determiner, noun.
noun        --> [customer].
```

DCGs can be run virtually directly as Prolog clauses, so that the Prolog proof procedure gives us a backbone for a top-down, depth-first, left-to-right parsing mechanism. For this purpose, Prolog interpreters equipped with a DCG translator will automatically convert the DCG-rules into Prolog clauses by adding two extra arguments to every symbol:

```
sentence(P1,P)     :- noun_phrase(P1,P2), verb_phrase(P2,P).
noun_phrase(P1,P)  :- determiner(P1,P2), noun(P2,P).
noun(P1,P)         :- 'C'(P1,customer,P).
```

We can state the first rule declaratively in English as: there is a sentence in the list `P1-P`, if there is a noun phrase in the list `P1-P2` and a verb phrase in the list `P2-P`. For a sentence to be well formed, no words should remain after the verb phrase has been processed, i.e. `P` should be the empty list (`[]`). The second rule has a reading similar to the first one. Finally, the third rule has been translated to a call of the special Prolog built-in primitive `'$C'`. The rule can be read as: there is a noun in the list `P1-P`, if the noun in `P1-P` is `customer`.

Normally, we are not satisfied with a grammar that simply recognises whether a given input string is a sentence of the language generated by the grammar. We also want to get the syntactic structure, or a semantic representation, of the sentence. To this purpose we augment the DCG notation in two different ways:

- we introduce additional arguments for nonterminal symbols
- we add Prolog goals in braces on the right-hand side of DCG-rules

Through the use of additional arguments DCGs allow us to describe "quasi context-sensitive" phenomena quite easily. Otherwise, representing occurrences of agreement or gap threading would be cumbersome, or even difficult. Prolog goals on the right-hand side of DCG-rules allow us to perform computations during the syntactic analysis of an input sentence.

A parser produces some structure of the input sentence that it has recognised. That means, a natural language sentence is not just a string of words, but the words are grouped into constituents that have syntactic and semantic properties in common. A quite natural way to represent syntactic trees is to supply every nonterminal symbol with an additional argument that encodes the information for the pertinent subtree. Thus, these additional arguments can be seen as partially specified trees in which variables correspond to not yet specified subtrees. Enhanced by arguments for a syntax tree, our DCG-rules would look like:

```
sentence(s(NP,VP))       --> noun_phrase(NP), verb_phrase(VP).
noun_phrase(np(DET,N))   --> determiner(DET), noun(N).
noun(cn(CN))             --> [CN], { common_noun(CN) }.
common_noun(customer).
```

Now it is easy to build the syntax tree recursively during parsing since Prolog's unification will compose subtrees step by step into the full syntax tree.

As shown DCGs use term structures but this can lead to two grave drawbacks. First, DCGs identify values by their position, so that a grammar writer must remember which argument position corresponds to which function. Second, since arity is significant in term unification, the grammar writer has to specify a value for each feature, at least by marking it as an anonymous variable.

Without changing the computational power of the existing DCG formalism, we can overcome these drawbacks by adding a notation for feature structure unification.

4.2 Feature Structures and Unification

Usually, unification-based grammar formalisms use a system based on features and values. A feature structure is an information-bearing object that describes a (possibly linguistic) object by specifying values for various of its features. Such a feature structure is denoted by a feature-value matrix. For example, the matrix

$$\begin{bmatrix} \texttt{a:b} \\ \texttt{c:d} \end{bmatrix}$$

contains the value `b` for the feature name `a` and the value `d` for the feature name `c`. A value can be either an atomic symbol or another feature structure. This leads to a recursive definition since one feature structure can be embedded inside another one. Consider the following matrix

$$\begin{bmatrix} \texttt{a:b} & \\ \texttt{c:} & \begin{bmatrix} \texttt{d:e} \\ \texttt{f:g} \end{bmatrix} \\ \texttt{h:Y} & \end{bmatrix}$$

Note that the value of the feature name `c` is itself specified by a feature structure. We will refer to such a feature name in our linguistic context as being category valued. The variable `Y` stands for a feature structure that contains currently no information.

It is helpful to have the notation of a path into an embedded feature structure to pick out a particular value. A path is just a finite sequence of features. For instance, in the structure above, the value corresponding to the path `c:d` is `e`, while the value corresponding to the path `a` is `b`.

The only operation on feature structures is unification. This is a monotonic and order-independent operation. Unifying two feature structures *A* and *B* means combining their informational content to obtain a structure *C* that includes all the information of both structures – but no additional information. For example, the feature structures

$$\begin{bmatrix}\texttt{a:b}\\ \texttt{c:X}\end{bmatrix} \quad\text{and}\quad \begin{bmatrix}\texttt{c:} & \begin{bmatrix}\texttt{d:e}\\ \texttt{f:g}\end{bmatrix}\\ \texttt{h:Y} & \end{bmatrix} \quad\text{unify to give}\quad \begin{bmatrix}\texttt{a:b} & \\ \texttt{c:} & \begin{bmatrix}\texttt{d:e}\\ \texttt{f:g}\end{bmatrix}\\ \texttt{h:Y} & \end{bmatrix}.$$

Note that unification can be impossible, if two feature structures contain conflicting information. In this case we say that the unification fails. The two feature structures

$$\begin{bmatrix}\texttt{a:b}\\ \texttt{c:X}\end{bmatrix} \quad\text{and}\quad \begin{bmatrix}\texttt{a:i} & \\ \texttt{c:} & \begin{bmatrix}\texttt{d:e}\\ \texttt{f:g}\end{bmatrix}\\ \texttt{h:Y} & \end{bmatrix} \quad\text{do not unify}$$

because the feature name `a` cannot have the two atomic values `b` and `i` at the same time. From the viewpoint of theoretical linguistic it is convenient to group feature structures to account for agreement, to mark case, to build syntactic trees and semantic representations, and to undo syntactic movements.

Here is a summary of feature structure unification where unifying the feature structures *A* and *B* results in the feature structure *C*.

- Any feature that occurs in *A* but not in *B*, or in *B* but not in *A*, also occurs in *C* with the same value.
- Any feature that occurs in both *A* and *B*, also occurs with unified values in *C*.
- The values in the feature structures are unified as follows:

  Two atomic symbols unify if they are equal, else the unification fails.

  A variable unifies with any object by making the variable equal to that object. Two variables unify by becoming the same variable.

  Feature structures are unified by applying the unification process recursively.

Unification of feature structures is very closely related to term unification in Prolog but there are three important differences:

- Feature structures use no functors other than the operator relating feature name and value, i.e. in our notation ':'.
- Feature structures are terms with unrestricted and order-independent arity.
- Feature structures identify values by feature names instead of positions in a term.

4.3 Graph Unification Logic Programming (GULP)

GULP is a syntactic extension of Prolog that supports the implementation of unification-based grammars by adding a notation for linearised feature structures [Covington 94a]. Thus, the feature matrix

$$\begin{bmatrix} \mathtt{a{:}b} & \\ \mathtt{c{:}} & \begin{bmatrix} \mathtt{d{:}e} \\ \mathtt{f{:}g} \end{bmatrix} \\ \mathtt{h{:}Y} & \end{bmatrix}$$

can be written in GULP notation as

```
a:b .. c:( d:e .. f:g ) .. h:Y
```

or

```
a:b .. c:d:e .. c:f:g .. h:Y  .
```

GULP adds to Prolog two operators and a number of system predicates. The first operator ':' binds a feature name to its value which can be a category. The second operator '..' joins one feature-value pair to the next.

The GULP translator accepts a Prolog program, scans it for linearised feature structures and converts them – by means of automatically built translation schemata – into an internal representation called value list. A value list is a Prolog term with a unique functor and a fixed number of arguments, where each position parameter corresponds to a keyword parameter. In our case we obtain the translation

```
g__([ g_(b), g_(g__([g_(e), g_(g)|_1])), g_(_2)|_3 ]) .
```

A value list is always open, i.e. its tail is a variable – e.g. `_1` in the example – that can be instantiated to a value containing new feature information which itself ends with an uninstantiated tail. Thus, value lists allow Prolog to simulate graph unification.

The current program GULP 3.1 has two limitations – it cannot handle negative, or disjunctive features [Covington 94a]. This means, we are not allowed to write:

```
a:b .. c: not ( d:e .. f:g ) .. h:Y
```

or

```
a:b .. c: ( d:e ) or ( f:g ) .. h:Y  .
```

4.4 DCG and GULP

GULP feature structures can be combined with the DCG formalism to yield a powerful lingua franca for natural language processing. Technically, they are coupled by introducing GULP feature structures as arguments into nodes of DCGs. To make the topmost DCG-rules account for case and agreement in number and person, we write:

```
sentence --> noun_phrase(case:nom .. agr:Number_Person),
             verb_phrase(agr:Number_Person).
```

Instead of using feature structures in argument positions, it is also possible to replace them by variables and to unify each variable with the appropriate feature structure. In this equational style unifications of feature structure are made explicit by writing them as Prolog goals in the body of the DCG-rules – before or after the category symbols. This is somewhat less efficient but diminishes the cognitive burden of interpreting structures. We can write either

```
sentence --> { NPSyn = case:nom,
               NPSyn = agr:Number_Person,
               VPSyn = agr:Number_Person },
             noun_phrase(NPSyn), verb_phrase(VPSyn).
```

or

```
sentence --> noun_phrase(NPSyn), verb_phrase(VPSyn),
             { NPSyn = case:nom,
               NPSyn = agr:Number_Person,
               VPSyn = agr:Number_Person }.
```

Some rules can loop when written in one way, but not in the other. The order of instantiation must be kept in mind and this again depends on the parsing strategy. Consider the following two left-recursive DCG-rules under top-down parsing

```
sentence(S1) --> sentence(S2), { S1 = x:a, S2 = x:b }.
sentence(S1) --> { S1 = x:a, S2 = x:b }, sentence(S2).
```

In the first case top-down parsing will lead to an infinite loop, however not in the second case.

5 Discourse Representation Theory (DRT)

5.1 Overview of DRT

The examples up to here could suggest that unification-based grammar formalisms are sentence-based, but this is not the case. Correct understanding of a specification requires not only processing sentences and their constituents, but also taking into account the way sentences are interrelated to express complex propositional structures. To unravel these interrelations we have to consider the context of a sentence while parsing it. One way to do so is to employ the help of Discourse Representation Theory (DRT), and to extend our top-down parser to extract the semantic structure of a sentence in the context of the preceding sentences.

DRT is a method for representing a multisentential natural language discourse in a single logical unit called a discourse representation structure (DRS) [Kamp 81, Kamp & Reyle 93]. DRT differs from the standard formal semantic account in giving up the restriction to the treatment of isolated sentences. It has been recognised that aspects such as pronominal reference, tense and propositional attitudes cannot be successfully handled without taking the preceding discourse into consideration.

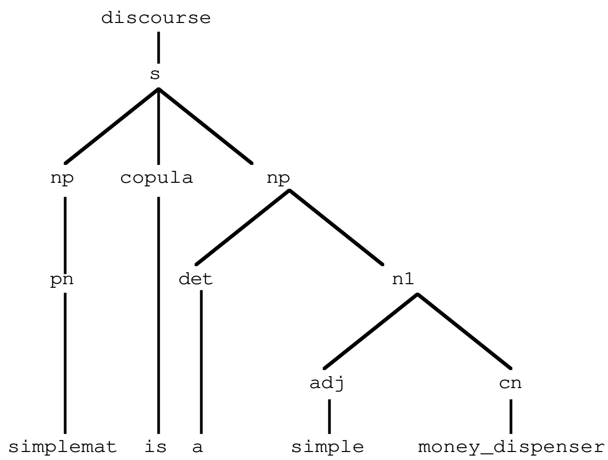
In general, a DRS K is defined as an ordered pair $<U,Con>$ where U is a set of discourse referents (discourse variables) and Con is a set of conditions. The syntax of K is one-sorted, i.e. there is only one type of discourse referents available in the universe of discourse U. The conditions Con are either atomic (of the form $P(u_1, ..., u_n)$ or $u_1 = u_2$) or complex (negation, implication, or disjunction). These conditions can be regarded as satisfaction conditions for a static model, or as goals for a dynamic theory. A DRS is obtained through the application of a set of syntactic-driven DRS construction rules *R*. These rules do not look just at the sentence under construction, but also at the DRS that has been built so far. Under this point of view, we can define the linguistic meaning of a sentence *S* as the function *M* from a given DRS *K* to *K´* induced by *R*.

5.2 Simple DRSs

The following discourse illustrates how a simple DRS is constructed.

```
SimpleMat is a simple money-dispenser.
It has a user interface.
```

Starting from the empty DRS K_0, the discourse representation structure is constructed sentence by sentence. While the first sentence is parsed top-down, the composition of the DRS K_1 goes ahead with the structural configuration of the sentence. Thus, the DRS construction rules will be triggered by the syntactic information. The relevant syntax tree of the first sentence is



Informally spoken, the following DRS construction rules are applied to this syntax tree during parsing.

- Introduce two different discourse referents into the universe. One for the entity named `simplemat` and the other for the entity `money_dispenser`.

  ```
  U = { X1, X2 }
  ```

- Introduce the conditions which the discourse referents must satisfy.

  `X1` must satisfy the condition of being named `simplemat`.

  `X2` must satisfy the two conditions of having the properties `money_dispenser` and `simple`.

  `X1` and `X2` must satisfy the condition to be equal (expressed by the copula `be` in the natural language sentence).

  ```
  Con = {   named(X1,simplemat),
            money_dispenser(X2), simple(X2),
            X2 = X1 }
  ```

The new DRS K_1 can be written in a more usual diagrammatic form as

```
X1 X2
----------------------
named(X1,simplemat)
money_dispenser(X2)
simple(X2)
X2 = X1
```

Now, let us try to incorporate the second sentence into the established DRS K_1 by extending it to K_2. To do this, we have to find – among other things – a suitable representation of the relation which holds between the personal pronoun *it* and its antecedent. In a written discourse a personal pronoun is mostly used anaphorically and not pragmatically, i.e. anaphors stand for discourse referents and not for words or phrases. They refer to some referent created in a previous step of the DRS construction. DRT makes the assumption that an antecedent can be found for every pronoun. An anaphoric pronoun and its resolving strategy can be introduced into the DRS through the following construction rules:

- Introduce a discourse referent for the anaphoric pronoun.
- Locate the referent of the anaphoric antecedent.
- Introduce the condition that the discourse referent of the pronoun equals the referent of the antecedent.

This definition leads to the extended new DRS K_2, where the logical flow of linguistic meaning from the first sentence to the second is maintained:

```
X1 X2 X3 X4
----------------------
named(X1,simplemat)
money_dispenser(X2)
simple(X2)
X2 = X1
user_interface(X4)
have(X3,X4)
X3 = X1
```

Here, the discourse referent `X3` for the pronoun *it* is understood to refer to the same entity as the discourse referent `X1` of his antecedent. The choice between the suitable referents is determined by constraints of agreement in gender and number. In addition, a new discourse referent (`X4`) and two atomic conditions (`user_interface(X4),have(X3,X4)`) have been introduced to the DRS K_2.

Let us consider the truth conditions of DRS K_2. The DRS is true if **we** can find real individuals **a** and **b** in the universe of discourse such that

- **a** is the bearer of the name SimpleMat
- **a** is a money-dispenser
- **b** is a user interface
- **a** has **b**

In other words, DRS K_2 is true provided there exist two individuals **a** and **b** for each of the discourse referents (`X1,X2,X3,X4`) in such a way that the conditions which K_2 contains for these discourse referents are satisfied by the corresponding individuals.

5.3 Complex DRSs

DRSs that represent conditional, universal or negative sentences are complex, they contain sub-DRSs.

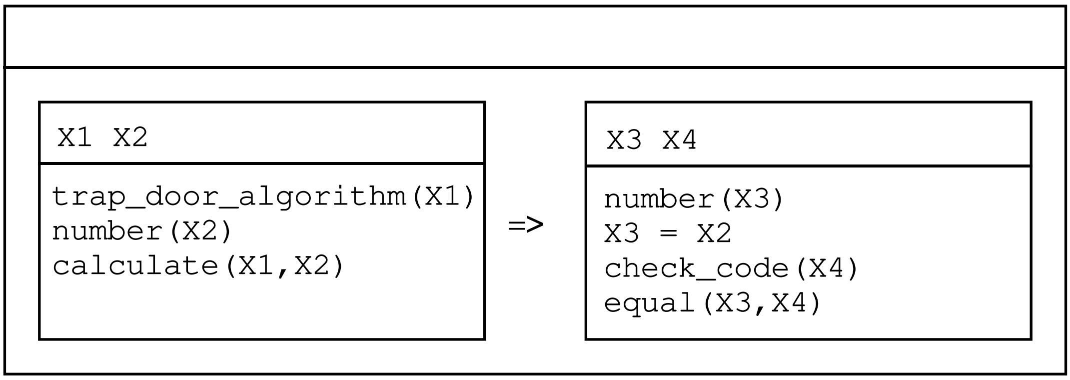
5.3.1 Conditional sentences

In linguistic terminology a subordinator is a member of a closed class of words defined by their role to induce clause subordination. An example is *if*. Sentences, in which a subordinate *if*-clause combines with a main *then*-clause are usually referred to as conditional sentences. The supposed *if*-clause is called the antecedent and the hypothetically asserted *then*-clause the consequent of the conditional. Intuitively, the consequent provides a situational description which extends that given by the antecedent.

For instance, the sentence

```
If the trap-door-algorithm calculates a number then the number equals the
check-code.
```

is represented in DRT as:



In general, a conditional sentence of the form *if A then B* contributes to a DRS K_0 a condition of the form K_1 => K_2, where K_1 is a sub-DRS corresponding to *A* and K_2 is the sub-DRS resulting from extending K_1 through the incorporation of *B*. In terms of truth conditions, the above conditional K_1 => K_2 is satisfied if and only if every individual for `X1` and `X2` that makes the sub-DRS K_1 true makes the sub-DRS K_2 true also. This definition contrasts with classical logic where the implication is also true in the situation when the antecedent is false.

Note that the above DRS assumes a different use for the two definite noun phrases: the anaphoric use and the unique reference use. The definite noun phrase `the number` is used anaphorically in the *then*-clause. Here, in DRS K_2, an equation of the form `X3` = `X2` is generated, where `X2` is the discourse referent of the antecedent object noun phrase. A unique reference use of the definite noun phrase `the trap-door-algorithm` is proposed in the *if*-clause because no antecedent can be found in the superordinate DRS K_0. In this case a potential agent will introduce a discourse referent with the appropriate conditions.

DRT claims that an anaphor can only refer to a discourse referent in the current DRS or in a DRS superordinate to it. A DRS K_1 is superordinate to a DRS K_2 if DRS K_1 contains DRS K_2, or is the antecedent of a conditional which has DRS K_2 as the consequent. This restriction makes correct predictions about the accessibility of antecedents to anaphors.

5.3.2 Universal statements

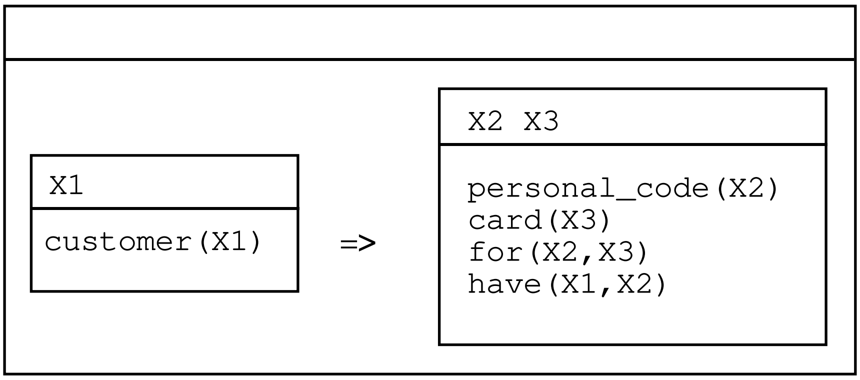
Universally quantified sentences are treated as conditional sentences. The sentence

```
Every customer has a personal-code for the card.
```

can be paraphrased as

```
If X1 is a customer,
then X1 has a personal-code for the card.
```

and that corresponds to the DRS:



This example shows that DRSs differ significantly from formulas of predicate calculus, and resemble Horn clauses. All conditions in the antecedent are implicitly universally quantified and each condition in the consequent has an implicit existential quantifier contingent on the antecedent. The sub-DRS K_1 – on the left of the arrow – is called the restrictor of the quantifier, the one on the right – K_2 – its scope. In formalisms like predicate logic the semantic contributions of the words *if ... then* and *every* would have to be simulated by appropriate combinations of the universal quantor and the implication connector. DRT seems to offer a much more natural representation for the systematic correlation between syntactic form and linguistic meaning of conditional sentences. This is in respect of the contextual role that DRSs were designed to play, namely as context for what is to be processed next, and not only as representations of what has been processed already.

5.3.3 Negative sentences

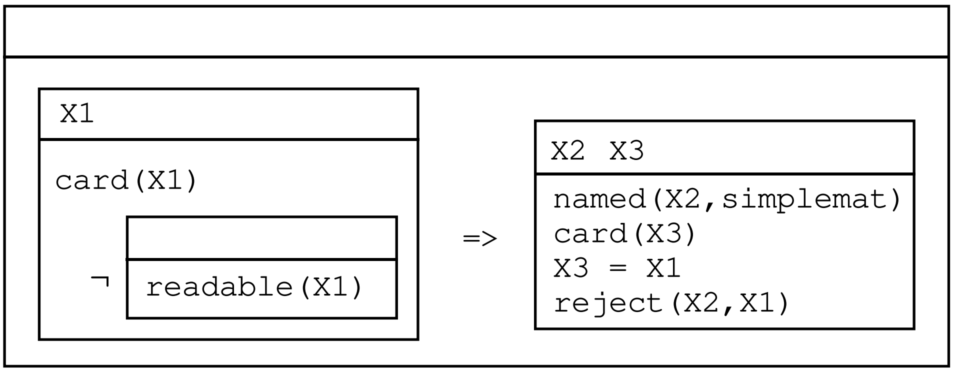
Negated sentences are represented by DRSs that contain sub-DRSs preceded by a negation symbol.
Consider the sentence

```
If the card is not readable then SimpleMat rejects the card.
```

which can be paraphrased as

```
If it is the case that there exists a card that is not readable,
then SimpleMat rejects this card.
```

Obviously, this reading corresponds to the following DRS:



Sub-DRSs can be used to represent other aspects of natural language. Kamp and Reyle propose methods to deal with disjunction, conjunction, plural, tense and aspect [Kamp & Reyle 93].

5.4 Ways to investigate a DRS

It is important to realise that a DRS can be investigated in several different ways.

First, it can be given a model-theoretic semantics in the classical predicate logic sense, where truth is defined in terms of embedding the DRS in a model. Intuitively, a DRS is true if each discourse referent in *U* can be mapped to an imaginary or real entity in the world model such that all of the DRS conditions *Con* are satisfied.

Second, a DRS can be manipulated deductively to infer further information using rules which operate only upon the structural content of the logical expressions.

And third, a DRS can be investigated from a more psychological point of view as a contribution of building up a mental model of a language user.

The second and the third ways lead to the concept of knowledge assimilation [Kowalski 93]. In this proof theoretic account a DRS is processed by resource-constrained deduction and tested whether it can be added to a continuously changing theory. The terms truth and falsity of DRSs in model theory are replaced by the proof of consistency and inconsistency in the process of knowledge assimilation. In other words, the correspondence between an incoming DRS and an agent's theory – based on its experience about the world – is tested by deduction.

A consistent DRS can be analysed further whether it is already implied by the theory, implies part of the theory, or is logically independent from it.

An inconsistent DRS identifies a part of the theory which participates in the proof of inconsistency and which is a candidate for revision.

5.5 Implementation

5.5.1 Parsing

Most of the sentence in our specification can be parsed top-down by the DCG-rules listed below. The underlying intuition for these rules is that each sub-constituent contains a certain word which is centrally important for the syntactic properties of the constituent as a whole; that word is called the lexical head of the constituent. For example, in the present context the head of a noun phrase is the noun and the head of a verb phrase is the verb. The categories that are sisters to the lexical head in the syntactic structures are its complements. Complements are those constituents that a lexical head subcategorises, e.g. the object of a verb. At a higher level we distinguish specifiers and modifiers. Specifiers are things like determiners in a noun phrase and modifiers corresponds to relative clauses or adjectives.

For the sake of clarity all feature structures in nonterminal symbols that rule out ungrammatical sentences, or build DRSs, are neglected in the DCG-rules below. Some terminal symbols are omitted and written as `[…]`.

```
discourse   --> sentence, ['.'], discourse.
discourse   --> [].

sentence    --> noun_phrase, verb_phrase.
sentence    --> noun_phrase, copula, adjective.
sentence    --> noun_phrase, copula, noun_phrase.
sentence    --> noun_phrase, copula, comparative_phrase.
sentence    --> [if], sentence, [then], sentence.
sentence    --> look_ahead, sentence, conjunction, sentence.

noun_phrase --> noun.
noun_phrase --> determiner, noun2.
noun_phrase --> [].
noun_phrase --> […].

noun2       --> noun1.
noun2       --> noun1, prepositional_phrase.
noun2       --> noun1, relative_clause.

noun1       --> noun.
noun1       --> adjective, noun1.

comparative_phrase   --> comparative, noun_phrase.
comparative          --> adjective, [than].
comparative          --> adjective, [or], adjective, [than].

prepositional_phrase --> preposition, noun_phrase.

relative_clause      --> […], sentence.

verb_phrase --> verb, noun_phrase .
verb_phrase --> negation, verb, noun_phrase.
copula      --> […], negation.

adjective   --> […].
copula      --> […].
conjunction --> [].
determiner  --> […].
noun        --> […].
negation    --> […].
preposition --> […].
verb        --> […].

look_ahead(P1,P) :-
  remove(and,P1,P).
```

The predicate `look_ahead/2` deserves closer inspection. It modifies the input string during parsing to avoid loops. In the DCG above, the predicate finds the conjunction `and` in the input list `P1` and removes it in advance. In our implementation we will deal with left-recursion in a non-destructive way by giving additional arguments to each phrasal node

```
sentence(Stack1) -->
  look_ahead(Stack1,Stack2),
  sentence(Stack2),
  conjunction(Stack2,Stack3),
  sentence(Stack3).
```

Furthermore, we define the new predicate `look_ahead/4`

```
look_ahead(Stack1,Stack2,P1,_) :-
  member(and,P1),
  Stack1 = [dummy],
  Stack2 = [and].
```

When a conjunction is a member of the input list `P1`, the variable `Stack1` is instantiated with a `dummy` and the encountered conjunction is pushed onto the `Stack2`. The conjunction can be removed from this stack leaving behind the empty `Stack3` when the DCG rule

```
conjunction([and],[]) --> [and].
```

was successful during parsing. This may not be a theoretically satisfying way to process conjoined sentences, but it works.

5.5.2 Feature structures

Most of the work of the parser is done by feature structure unification implemented using GULP. Therefore, each linguistic object must be described through its feature structure. Such information-bearing objects are called signs. A sign is a partial information structure which mutually constrains possible collocations of graphematical form, syntactic structure, semantic content, discourse factors and phrase-structural information. Signs fall into two disjoint classes – those which have internal constituent structure (phrasal signs), and those which do not (lexical signs or words). In our approach, phrasal signs are realised as phrase-structure rules in DCG notation and lexical signs are elements of the linguistic lexicon.

Feature structures are not only a powerful tool to rule out ungrammatical sentences but are also well-suited to build DRSs. The DCG-rules have the task to compose DRSs in combining predicate-argument structures of lexical signs with partly instantiated feature structures of phrasal signs.

Feature structures are of the general form:

```
gra:    ...

syn:  ( head:   ...
        subcat: ... )

sem:  ( index:  ...
              ( arg1: ...
                arg2: ... )
        rel:    ... )
```

```
drs:  ( in:  ...
        out: ...
            res:    ( in:  ...
                      out: ... )
            scope: ( in:  ...
                      out: ... ) )
```

Not all of the features are instantiated for each sign. The role of these features is as follows:

- `gra`

  The value for this feature name is the information about the graphematical form of a lexical sign.

- `syn`

  The syntactic information part for a sign is divided into two sub-parts: First there are the `head` features, which specify syntactic properties (as case, agreement and position) that a lexical sign shares with its projections. Second, the `subcat` features gives information about the subcategorisation, or valence, of a sign. Its value is a specification of the number and kind of the signs which characteristically combine with the sign in question to saturate it.

- `sem`

  These features are defined for lexical signs only. The value for `index` is a discourse referent and is created for nouns during parsing. The other signs that have indices (adjectives, noun phrases or pronoun) obtain them by unification. The values for `arg1` and `arg2` are discourse referents for the subject and the direct object of the verb. And finally, the value for the feature name `rel` is the property expressed by the lexical sign.

- `drs`

  DRS features are defined for nonterminal symbols. The feature name `drs:in` stands for the DRS as it exists before processing the current phrasal sign. The state of the DRS after processing the current phrasal sign is expressed by the value for `drs:out`. And finally, `res` and `scope` are used to determine the logical structure of a sentence.

The lexicon is the place where the graphematical, syntactic and semantic properties of a word are specified. The following lexical entry for the verb form `rejects` is mostly self-explanatory. New are the two head feature names `maj` and `vform`. The major value `v` corresponds to the familiar notion of part of speech, namely verbal. And the verb form's value `fin` specifies the verb as finite.

```
lex_tv(
  gra:   rejects,
  syn: ( head:         ( maj:v ..
                         vform:fin ) ..
         subcat:subj: ( head:maj:n ..
                         head:case:nom ..
                         head:agr:person:third ..
                         head:agr:number:singular  ) ..
         subcat:dobj: ( head:maj:n ..
                         head:case:acc ) ),
  sem: ( index:        ( arg1:X ..
                         arg2:Y  )  ..
         rel:           [reject(X,Y)] ) ).
```

5.5.3 Building DRSs

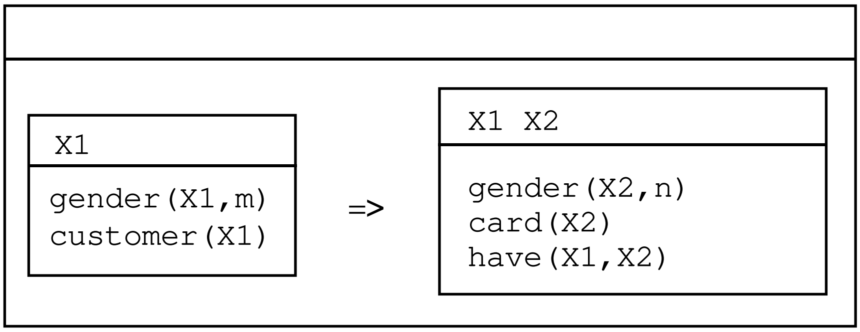
A DRS is represented by a Prolog term of the form `drs(U,Con)`. `U` is a list of discourse referents represented by unique Prolog variables and `Con` is a list of terms containing these variables. For example, the DRS for the sentence

```
Every customer has a card.
```

becomes in Prolog

```
drs(
  [],
  [ ifthen(
      drs([X1],   [gender(X1,m), customer(X1)]),
      drs([X2,X1],[gender(X2,n), card(X2), have(X1,X2)]) )
  ] )
```

Before we discuss how this Prolog term is composed during parsing, we note in passing that we can map the above term directly to a diagram if we interpret the predicate `drs/2` as a graphic frame and the predicate `ifthen/2` as =>.



Despite their complexity, we can build such DRSs by unification of arguments in DCG-rules only. The key question is how to represent determiners. Semantically, a determiner affects more of the structure than its minor syntactic role would suggest. On the semantic level a determiner is an ingredient for making a quantifier. A determiner has two arguments: a restrictor and its scope. The restrictor consists of the remaining conditions within the noun phrase (restrictor = noun phrase – determiner). The scope is composed of the conditions expressed by the verb phrase. Thus, a determiner is a relation that takes a restrictor and a scope and turns them into a DRS. The most general form of a determiner's feature structure is:

```
drs:in:  ...
drs:res:in:  ...
drs:res:out: ...
drs:scope:in:  ...
drs:scope:out: ...
drs:out: ...
```

The DCG-rule for the determiner takes information coming in from previous DRSs (`drs:in`) and passes it to the restrictor (`drs:res:in`). After that, the rules dealing with the restrictor add more semantic information to the feature structure (`drs:res:out`) and pass all the information to the scope (`drs:scope:in`). Finally, it remains the task of the rules for the scope to add its information to the feature structure (`drs:scope:out`) and this forms the semantic information (`drs:out`) of the sentence.

Determiners can manipulate a DRS in complex ways as the example for the determiner `every` shows:

```
determiner(Det) --> [every],
  { Det = drs:in:DRSIn,
    Det = drs:res:in:[drs([],[])|DRSIn],
    Det = drs:res:out:Res,
    Det = drs:scope:in:[drs([],[])|Res],
    Det = drs:scope:out:[Scope,Res,drs(U,Con)|Super],
    Det = drs:out:[drs(U,[ifthen(Res,Scope)|Con])|Super] }.
```

In our current implementation it is assumed that the subject noun phrase has scope over the object noun phrase, but according to the judgement of English speakers such sentences are ambiguous. In real life, almost any quantifier can be raised to have scope over the entire sentence. Especially the determiner `each` almost always raises; `some`, `all` and `every` can raise but need not to raise; `any` tends not to raise, and numerals generally do not raise [Covington 94b].

5.5.4 From DRSs to Prolog

Translating DRSs into Prolog clauses poses a problem – free variables in Prolog facts and rules have implicit universal quantifiers. It is not possible to translate the DRS for

```
SimpleMat is a simple money-dispenser.
```

namely

```
X1 X2
---------------------
named(X1,simplemat)
money_dispenser(X2)
simple(X2)
X2 = X1
```

into Prolog as:

```
named(X1,simplemat).
money_dispenser(X2).
simple(X2).
X2 = X1.
```

The first fact would mean "Anything is named SimpleMat". We would not even be able to say that the money-dispenser is the same thing as the object named SimpleMat, because variables with the same name in different clauses are distinct. What we need is a discourse marker for each existentially quantified entity. For that reason, a constant (integer) has to be randomly chosen to represent the individual `X1`. Then, the DRS conditions, with the constants instantiated for the discourse referents, would be asserted to the knowledge base:

```
named(1,simplemat).
money_dispenser(1).
simple(1).
```

Two additional problems arise when we translate conditions which use sub-DRSs. First, Prolog rules cannot have two predicates in its consequent, i.e. rules of the form

```
a,b :- c,d.
```

are not permitted. To deal with this problem, Covington and his collaborators introduce a special operator (`::-`) as intermediate representation for rules with more than one consequent [Covington et al. 88]. Now we can write

```
a,b ::- c,d.
```

Since this rule cannot be asserted directly into the knowledge base it is split up into several Prolog rules by distributing the consequents:

```
a :- c,d.
b :- c,d.
```

Second, if the consequent of a conditional sentence introduces new variables, these variables have implicit existential quantifiers which depend on the antecedent. Since Prolog cannot represent this dependence directly, we have to simulate it by a form of skolemisation which goes into extended Prolog as

```
card([2,X1]), have(X1,[2,X1]) ::- customer(X1).
```

and is represented as the two Prolog rules:

```
card([2,X1])        :- customer(X1).
have(X1,[2,X1]) :- customer(X1).
```

The Prolog term `[2,X1]` can be interpreted as a value that is a function of the value of `X1`.

6 Concept Lexicon

6.1 Introduction

Natural language as a standard means of communication is an essential part of requirements engineering. However, uncontrolled NL often expresses the specification in an incomplete and ambiguous way. While software development proceeds, we face a gradual formalisation of our initial ideas expressed in NL. At this more formal level NL statements denote, for example, entities, events and processes. Finally, at the implementation level, NL statements are being used in data-dictionaries and application programs.

It would be of great help, and the quality of the resulting software system would be increased considerably, if all these statements are used in a consistent manner. For that purpose a conceptual lexicon is needed containing these statements in the form of lexical items thus facilitating the understanding of what statements in one language (e.g. NL) stand for in another language (e.g. Prolog).

In requirements engineering, a lexicon can be thought of as the collection of concepts used in a particular application domain. A concept refers to the particular information used to represent a category (or exemplar) on a particular occasion [Rosch & Loyd 78]. More specifically, it will be assumed that the concept used to represent a category on a particular occasion contains (1) information that provides relevant expectations about the category in that context, and (2) information that provides relevant expectations when interacting with the category in most contexts [Barsalou 82, Medin & Smith 84, Lakoff 87]. When dealing with a concept lexicon we are especially interested in the features of a concept, its instances in the application domain, and its links to other concepts.

Recently there has been a great deal of interest in the structure of concept lexicons. One of the major problems encountered has been the optimal organisation of lexical knowledge necessary for robust natural language processing systems.

In this chapter, we are concerned with the semantic and conceptual structure of lexical items within a lexicon that supports the process of software development. We will report on that part of the NLP project in which the structure, content and use of such a concept lexicon has been investigated.

6.2 The Lexicon in Time and Context

From a traditional point of view, we can distinguish between two approaches to the study of word meaning: primitive-based approaches and relation-based approaches. We will outline both approaches and argue for the necessity to leave the beaten track.

6.2.1 The Classical View

What is the meaning of a word? The classical answer to this question is generally ascribed to Aristotle. The classical theory holds that a concept can be stated in terms of individually necessary and jointly sufficient features for membership of a thing in a category. For example, an "ATM" is denoted by the noun phrase `the automatic teller machine`. Following the classical view an ATM is defined by an enumeration of its features, e.g. screen, key-pad and card-slot. These features are often referred to as the "extension" of an NL statement.

In linguistics, this view has emerged in the componential analysis of Katz and Fodor [Katz & Fodor 63]. Leech and Bierwisch continue this line of research [Bierwisch & Lang 87]. This, indeed, is the typical form of the classical view in linguistics: the meaning of a word is essentially decomposable into a set of features. The features that represent the meaning of a word are just those which distinguish it from others in the relevant domain.

Moreover, most of these approaches assume that word meaning can be exhaustively defined in terms of a fixed set of primitive elements (e.g. [Schank 75, Wilks 75]). Schank, for example, argues that a small number of concepts corresponding to "primitive acts" can be used to construct meaning representations for most descriptions of events. These primitive concepts are simple actions of the kind "move a body part" (MOVE), "build a thought" (MBUILD), "transfer a physical object" (PTRANS), and "transfer mental information" (MTRANS). Inferences are made through such primitives [Miller & Johnson-Laird 76]. For example, to answer questions like: What actor performs which action, what actor has which objects, or what directions does a particular actor have?

In contrast to this primitive-based view, a relation-based view of word meaning claims that there is no need for decomposition into primitives if words (and their concepts) are associated through a network of explicitly defined links (e.g. [Quillian 68, Brachman 79, Lyons 81]). Sometimes referred to as meaning postulate, these links establish any inference between lexems as an explicit part of a conceptual network. The issue, then, is the question of whether words are meaningful solely in relation to other words in the same language: a word means what it means because of what it does not mean. This view has been held by linguists such as Lyons, and comes from the Saussurian tradition [Saussure 66]. Lyons, for example, puts it like this [Lyons 81, p. 124]: "The sense of a lexical item is [...] identical with, the set of relations which hold between the item in question and other items in the same lexical system."

However, according to the organisation of lexicons, primitive-based and relation-based approaches follow a single strategy. The traditional organisation of lexicons in natural language processing (NLP) systems assumes that word meaning can be exhaustively defined by an enumerable set of extensions per word. Computational lexicons, to date, generally tend to follow this organisation. As a result, whenever natural language interpretation tasks face the problem of lexical ambiguity, a particular approach to disambiguation is warranted. The system attempts to select the most appropriate "definition" available under the lexical entry for any given word. The selection process is driven by matching meaning representations (i.e. extensions) against contextual factors (e.g. different application domains where the same expression is in use). One disadvantage of such a design follows from the need to specify, ahead of time, the contexts in which a word might appear; failure to do so results in incomplete coverage. Furthermore, lexicons are currently of a distinctly static nature, the division into separate word senses not only precludes permeability; it also fails to account for the creative use of words in novel contexts.

6.2.2 Leaving the Beaten Track

We argue below that our model for representing lexical knowledge is superior to lexical entry formats currently in use both in terms of expressiveness and the kind of supported interpretative operations. Current models in lexical representation select among a static, pre-determined set of word sense. Once a certain structure for lexical items is decided to be useful for the particular application domain, it is often assumed that a full-fledged dictionary exists for the software's lifetime. Thus, the dictionary plays a passive role. We, to the contrary, believe a conceptual lexicon is open-ended in nature. Our model for lexical representation allows for extendibility. It differs from most approaches, because lexical information is taken to be incomplete. Here the lexicon plays an active role, because it is the conceptual model that needs to be built by using the natural language specification.

Mere understanding of the syntax or even the specific semantics of a controlled language is not the most crucial factor in bridging the communication gap that often occurs in requirements engineering. The stated or unstated assumptions reflecting the shared knowledge of people familiar with the social, business, and technical contexts in the application domain are of far greater significance. Thus, their considerations in lexical development is essential.

Adopting a rich and expressive lexicon of domain concepts has a number of benefits. Such a lexicon helps, for example, to take into account different viewpoints, which are an important issue in requirements engineering and in software development in general [Hofmann 93]. The immediate understanding of a lexical item is often restricted to simple items. For most items, however, their understanding requires the exploration of their conceptual dependencies (e.g. usage context) in order to grasp their meaning. The lexical information accessed is influenced by the demands of the communicative situation, which include the goals of the individuals or groups performing a software project. For example, there is evidence that people remember a passage differently according to the goal(s) they want to achieve. This evidence supports the notion that the motivation for processing a sentence can affect the way it is processed [Anderson et al. 76, Barsalou 82].

From the point of view of a NLP system, a rich and extensive conceptual lexicon can offer improvements in its robustness of coverage, especially in ambiguity resolution during parsing. Such benefits stem from the fact that our model offers a scheme for explicitly encoding lexical knowledge at several levels of generalisation. As we discuss below, factoring these along different dimensions makes it possible for NLP systems to carry out semantically intensive tasks.

6.3 A Model for Lexical Semantics

In this section, we will outline a model for lexical semantics. We will show what we think is the basic structure of a lexical item. We will present an approach to lexicalisation, where lexical items are structured forms (or templates). This will provide a generative framework for the composition of lexical meaning.

The model we have in mind acts to constrain possible word meanings, by restricting the forms that lexical decomposition can take. Our model for lexical semantics currently offers a template that characterises the semantics of nominals [cf. Grimshaw 90]. A lexical item is specified by four roles of meaning:

- *Purpose*: The purpose denotes the function of a lexical entry in terms of the goals for a specific software project. An Automatic Teller Machine (ATM) has, for example, the purpose to give money to an authorised user.
- *Structure*: The structure shows how a lexical entry is globally related to other entries in the same conceptual lexicon. For example, an ATM is one of the banks interfaces to its customers and it is connected to a bank consortium.
- *Form*: Those attributes of a lexical entry that describe the relation with its constituting parts. For example, an ATM is composed of a screen, key-pad, door, card-slot, etc.
- *Behaviour*: The necessary behaviour of a lexical entry is to fulfil its purpose. That are, those factors which are involved in its origin and its “bringing about”. For example, an ATM's behaviour is giving a receipt, providing the correct amount of money, or keeping a card due to security restrictions.

This model of lexical semantics is based on two premises. First, we suggest an evolving set of core components for constructing lexical items. Second, instead of a fixed meaning ascribed to a certain lexical item, we rely on contextual expansion to derive its meaning.

Evolving Set of Core Components. In AI literature primitives have been suggested and employed with varying success [Schank 75, Wilks 75]. What we would like to do, however, is to propose a new way of viewing primitives looking more at the generative or compositional aspects of lexical semantics, rather than the decomposition into a specific number of primitives. For that, we do not propose a fixed set of universal primitives, but build lexicons on core components that evolve during the software's development and use. These components are by no means fixed, but denote the most primitive instances of domain concepts currently in use.

Contextual Expansion of Conceptual Meaning. Rather than committing ourselves to an enumeration of a pre-determined number of different word meanings, a lexical entry encodes a range of representative aspects of conceptual meaning. That is, we focus on aspects essential to the people engaged in constructing and using the concept lexicon. As we will demonstrate for an isolated word, such meaning components define the semantic boundaries appropriate to a lexical item's use. When embedded in context, however, mutually compatible roles in the lexical decomposition of each item become more prominent, thus forcing a specific interpretation of individual lexical items within a specific context. It is important to realise that this is a generative process, which goes well beyond the simple matching of features. In fact, this approach requires, in addition to a flexible notion for expressing semantic generalisations at the conceptual level, a mechanism for composing the "meaning" of these individual items on the phrasal level. DRS contributes this mechanism in our approach (cf. section 5 for more detail).

The contribution of such a dynamic concept lexicon to an NLP system is twofold. On the one side, a dynamic concept lexicon supports inference, especially the resolution of ambiguities occurring during the parsing of a text written in controlled language.

Let's assume we have specified an ATM as something that accepts a card. Such a card, in turn, is currently described as a card issued by a particular bank. Now someone tries to insert his or her credit card. Although the credit card is issued by the bank, the case is ambiguous in the sense that we do not have specified what we mean by the term issued. Do we mean all cards giving access to the bank's service or just those cards where the issuing company is owned by the bank which provides the financial service?

To resolve this issue would be beyond a machine's capacity. The point is that while the inference can build on an emerging set of core components, man-machine interaction is necessary to update and expand these components. On the other side, a dynamic concept lexicon facilitates text generation. That is, each feature value (i.e. `purpose`, `structure`, `form`, `behaviour`) is represented in a logic form and as text in a controlled language. The linkage between textual and logic representation is used to generate a NL description in controlled language from a set of lexical items.

6.4 TANDEM – The Conceptual Lexicaliser

TANDEM, a domain-based computer tool for lexicalising concepts, incorporates both lexical data and domain knowledge. It is a tool that implements the proposed model for lexical semantics. As Webster tells us the term TANDEM means: "[...] a vehicle having close-coupled pairs of axles [or] a group of two or more [...] acting in conjunction." Thus, the analogy is twofold. First, there is the project-dependent coupling between "work-axles." From the tool's perspective, the division of labour between machine and human that the tool supports is crucial for its success. Second, the focus on compositional aspects during conceptual modelling will clarify the process and will enhance its computer-support, too.

The availability of such a computer-based tool is highly desirable because it simplifies the creation of a formal specification while increasing the reliability of the formulation process. Moreover, it improves the maintainability of the formal specification and expands the base of potential users [Balzer et al. 78]. The question of feasibility – the paramount issue – rests clearly on the ability to correctly interpret an informal specification (i.e. the problem statement). ´

As follows, we will present some preliminary results obtained by the prototype system TANDEM and describe its control structure and operation so that the reader can observe its performance level and judge for herself the generality of its lexical template and therefore its feasibility. We will do so by showing its implementation in Prolog.

6.4.1 The Control Structure of TANDEM

The top-down parser written in DCG/GULP notation provides TANDEM with a DRS of the parsed sentence and a list of its lexical conditions. The predicate `tandem/2` controls the lexicalisation of the DRS:

```
tandem(DRStructure,LexWordList):-
  conceptualize_drs(DRStructure,LexWordList,NewDRS),
  get_words(lex_cn,LexWordList,CommonNouns),
  update_lexicon(CommonNouns),
  lexicalize_concepts(CommonNouns,NewDRS,RestDRS),
  get_words(lex_pn,LexWordList,PersonalNouns),
  update_lexicon(PersonalNouns),
  lexicalize_concepts(PersonalNouns,RestDRS,_).
```

First, we conceptualise the given DRS. That is, we adjust the DRS for the purpose of conceptual modelling. The predicate `conceptualize_drs/3`, for example, resolves verb subcategories for a complement. Next, we filter the common nouns out of the conceptualised DRS (`get_words/3`). Before we can start to lexicalise the common nouns, we have to update the lexicon. The predicate `update_lexicon/1` checks for each noun, whether it is already stored in the concept lexicon. Then the predicate `lexicalize_concepts/3` lexicalises the nouns of a given DRS successively. Moreover, all the references of the common nouns to other parts of the DRS are resolved. For all proper names of the given sentence, which are not lexicalised yet, `tandem/2` follows the same procedure as for the common nouns.

In the following, we will explain our method of lexicalising concepts by outlining the implementation of TANDEM. We will explain the two major steps to lexicalise a given DRS in more detail: (1) reference resolution and (2) lexicalising concepts.

6.4.2 Reference Resolution

The DCG/GULP parser provides a reference structure for a given sentence. Let's assume we have parsed the sentence:

```
A money-dispenser activates a simple printer.
```

This would lead to the following DRS represented in a Prolog-like notation:

```
money_dispenser(1)
printer(2)
simple(2)
activate(1,2)
```

To lexicalise a concept, we have to resolve its references. In the case of the concept `money_dispenser` the reference number is `1`. Given the reference number, which denotes a discourse marker, and the DRS, the predicate `resolve_ref/4` binds all elements of the DRS structure, where the reference number has the same value.

In our example, `activate` is bound to the concept `money_dispenser`. While resolving the references of a given concept `resolve_ref/4` also constructs a list of the unresolved conditions of the initial DRS.

To resolve the references of a given concept, we can distinguish three cases: (a) instance of the concept, (b) feature of the concept, and (c) a link to the concept. The first part of the predicate `resolve_ref/4` handles the cases (a) and (b). It also provides the interface for lexicalising concepts:

```
resolve_ref(Concept,[Referent|Refs],WorkDRS,RefNr,DRS):-
  Referent =.. RefElem,
  on(RefNr,RefElem),
  (
    on(named,RefElem),                          % instance of concept
    lexicalize_drs(Concept,RefElem)
  ;
    mem(RefElem,[2],MemNr),RefNr = MemNr,       % feature of concept
    functor(Referent,_,Arity),
    update_refs(Referent,Arity,RefElem,UpdateElem,Refs,RefNr),
    lexicalize_drs(Concept,UpdateElem)
  ),
  resolve_ref(Concept,Refs,WorkDRS,RefNr,DRS).
```

Let's take a look at case (a) first. In our approach proper names give names to the instances of a concept. Proper names like SimpleMat are represented as: `named(1, simplemat)`. If the reference number of the concept, e.g. `money_dispenser(1)`, equals the proper names' reference number, than the reference can be resolved immediately. That is, SimpleMat is lexicalised as a type of `money_dispenser`.

In case (b), `resolve_ref/4` resolves a feature of the concept. In our example sentence, `activate(1,2)` would be a feature of `money_dispenser(1)`. In order to resolve `activate(1,2)`, we have to find out what the reference number (`2`) stands for in the context of the given sentence. For that, we call the predicate `update_refs/6`. This predicate replaces those arguments of a condition (i.e. `activate`) that are numbers, but do not equal the actual reference number (`1`). In our example, the updated reference would be `activate(1,printer)`:

```
resolve_ref(Con,[Referent|Refs],WorkDRS,RefNr,[Elem|DRS]):-
  get_logic_form(Con,ConForm),
  functor(Referent,F,Arity),                    %  link  (pointer)  to  a
concept
  functor(Elem,F,Arity),
  search_for_ref(Referent,RefNr,Elem,ConForm,Arity),
  resolve_ref(Con,Refs,WorkDRS,RefNr,DRS).

resolve_ref(Con,[Ref|Refs],WorkDRS,RefNr,[Ref|DRS]):-
  resolve_ref(Con,Refs,WorkDRS,RefNr,DRS).

resolve_ref(Con,[],WorkDRS,RefNr,[]).
```

For case (c), we have the situation that the reference marker, e.g. `2`, is not the first argument of the actual referent, e.g. `activate(1,2)`. Although we cannot resolve such a referent immediately, `resolve_ref/4` updates its arguments. For that, the reference marker is replaced by the concept at hand, which would be `printer(2)` for our example sentence. The resulting referent, i.e. `activate(1,printer)`, is added to the list of unresolved conditions. The same is true for all other conditions to be resolved that are not covered by cases (a)-(c).

6.4.3 Lexicalising Concepts

TANDEM uses different techniques for different linguistic constructs. Every common noun results in an entry of the concept lexicon. With the predicate `lexicalize_drs/2`, the current version of TANDEM lexicalises proper names, adjectives, and verbs according to a given concept respectively common noun:

```
% lexicalize a proper name
lexicalize_drs(Concept,DRSList):-
  remove(named,DRSList,NewList),
  append(_,[DRSItem],NewList),
  update_concept(DRSItem),
  update_instance(DRSItem,Concept,is_type_of).

% lexicalize an adjective
lexicalize_drs(Concept,[DRSItem,_]):-
  update_feature(DRSItem,Concept,form).

% lexicalize a verb
lexicalize_drs(Concept,DRSList):-
  DRSItem=..DRSList,
  update_feature(DRSItem,Concept,form).
```

To maintain the concept lexicon, TANDEM continuously updates its entries. Two predicates are mainly responsible for updating the concept lexicon: `update_instance/3` and `update_feature/3`.

```
update_instance(Item,Concept,FType):-
  retract(lex_con(
             gra: Item,
             con: ( purpose: P ..
                    structure: S ..
                    form: F ..
                    behavior: B) )),
  get_logic_form(Concept,LForm),
  resolve_feature(FType,LForm,P,S,F,B,NewP,NewS,NewF,NewB),
  add_concept(Item,NewP,NewS,NewF,NewB).
```

The predicate `update_instance/3` updates an instance (e.g. `simplemat`) of a given concept (e.g. `money_dispenser`) according to a given feature type (e.g. `structure`). In our approach, the feature names of a lexical condition are `purpose`, `structure`, `form`, and `behaviour`. To lexicalise an instance, `update_instance/3` restores the structural link to the given concept. For example, the Prolog query `update_instance(simplemat, money_dispenser,is_type_of)`, TANDEM creates the following lexical condition:

```
simplemat
  purpose:
  structure: is_type_of(X, money_dispenser(Y))
  form:
  behavior:
```

The predicate `update_feature/3` updates any value of a lexical condition according to a given feature type. When updating a feature value, we have to ensure that the previous enrichment of a lexical feature is considered. For that we cannot solely rely on linguistic information, because the logic form of a condition as provided by the linguistic lexicon is static. Unlike TANDEM's concept lexicon, the linguistic lexicon is not dynamically updated. For example, the logic form of a condition like `have(1,user_interface(Y))` would be `have(X,Y)`. While we want to generalise from the current value of the reference number (`1`) to any referent (`X`), we also want to keep the previous enrichment, i.e. `user_interface(Y)`. With the help of `keep_item_enrichment/2` the resulting feature is both generalised and also preserves the previous enrichment, e.g. `have(X,user_interface(Y))`.

```
update_feature(Item,Concept,FType):-
  retract(lex_con(gra: Concept,
                  con: (purpose: P ..
                        structure: S ..
                        form: F ..
                        behavior: B) )),
  (get_logic_form(Item,LForm)
  ;
    keep_item_enrichment(Item,LForm)
  ),
  resolve_feature(FType,LForm,P,S,F,B,NewP,NewS,NewF,NewB),
  add_concept(Concept,NewP,NewS,NewF,NewB).
```

6.5 On the Road: An Example Session with TANDEM

In this section, we illustrate our approach to lexicalising concepts. For that, we present some interesting and problematic examples of lexicalising concepts.

Let's take a look at the sentence

```
SimpleMat is a simple money-dispenser.
```

of the ATM specification. This sentence is parsed by DCG/GULP resulting in the following DRS in Prolog:

```
named(1,simplemat)
money_dispenser(1)
simple(1)
```

According to our method of lexicalisation we then select all common nouns. In this case, the only common noun selected is money-dispenser.

Reference Resolution. The concept `money_dispenser` is the starting point for the reference resolution in the sentence:

```
money_dispenser(X)
  simple(X)
```

Through the reference number we can assign the adjective `simple` to the common noun. In other words, if two conditions of the DRS have the same referent, for example, `simple(1)` and `money_dispenser(1)`, than the common noun denotes the concept and the other condition is used to specialise it.

Next TANDEM resolves the proper name, e.g. SimpleMat. Proper names are resolved with the help of common nouns, e.g. `money_dispenser`. In other words, instances like `simplemat` are resolved with the help of the denoted concepts.

```
simplemat(X)
  is_type_of(X,money_dispenser(Y))
```

Lexicalising Concepts. If not existing, TANDEM creates conceptual entries for money-dispenser and SimpleMat:

```
money_dispenser
  purpose:
  structure:
  form: simple(X)
  behaviour:
```

```
simplemat
  purpose:
  structure: is_type_of(X,money_dispenser(Y))
  form:
  behavior:
```

In a lexical entry, adjectives like `simple` are per default given the feature `form`. When `simple` is specified in more detail later on it might shift to one of the other features of the lexical condition, for example, a person can understand `simple` as the easy way to interact with a money-dispenser. That, in turn, might lead to the specification of an easy to read display, comfortable key-pad, etc. However, this is not the only interpretation possible. Another person can ascribe the following meaning: `simple` stands for a minimal version of security checks. That is, someone suggests to have money-dispensers without a door to cut costs. Whatever interpretation of the adjective `simple` is adopted, the important point is not to obstruct one of these interpretations. On the other hand, we want to make sure that once an interpretation has been chosen everybody sticks to it.

Parsing the sentence

```
The customer has a card that carries machine-readable data.
```

of the ATM specification leads to the following DRS. The DRS contains `have` and `carry`, because we work with the basic form of verbs in lexicalising DRSs:

```
customer(2)
card(1)
data(0)
machine_readable(0)
carry(1,0)
have(2,1)
```

The common nouns of the sentence are customer, card, and data.

Reference Resolution. Starting with the concept `customer` resolves the condition `have(2,1)` of the DRS:

```
customer(X)
  have(X,card(Y))
```

The concept `card` is referred by the condition `carry(1,0)`. Here we have two possible referents: `data(0)` and `machine_readable(0)`. In such a case, the (common) noun is always selected, because it denotes the more general form. This allows not only to resolve the actual specialisation, i.e. `machine_readable`, but also specialisations that are added later on in the conceptualisation of a particular domain.

```
card(X)
  carry(X,data(Y))

data(X)
  machine_readable(X)
```

Lexicalising Concepts. Similar to the preceding sentence, the lexicalisation of these concepts is straightforward leading to the following lexical conditions:

```
data
  purpose:
  structure:
  form: machine_readable(X)
  behavior:

card
  purpose:
  structure:
  form: carry(X,data(Y))
  behavior:

customer
  purpose:
  structure:
  form: have(X,card(Y))
  behavior:
```

Parsing the sentence

```
A trap-door-algorithm uses the personal-code as input.
```

of the ATM specification leads to the DRS:

```
trap_door_algorithm(1)
personal_code(2)
input(3)
as(2,3)
use(1,2)
```

The common nouns of the sentence are trap-door-algorithm, personal-code, and input.

Reference Resolution. To resolve the references of this sentences, we have to deal with a so-called verb subcategory for a complement, e.g. `uses ... as input.` The verb `uses` does not call for a preposition. That is, it is already saturated. Consequently, the "content" of the as-construct (`input`) is linked to the object of the sentence, i.e. `personal_code`.

In a next step, the relationship of the referred noun (`personal_code`) to other parts of the DRS has to be determined. In our sentence there is link between `trap_door_algorithm` and `personal_code`, because the trap-door-algorithm uses the personal-code.

To finally resolve the web of relationships, we have to replace the referred noun, i.e. `personal_code`, by its denoted use, i.e. `input`, and then to mark down that `input` is of type `personal-code`: `is_type_of(input,personal_code).`

```
input(X)
  is_type_of(X,personal_code(Y))

personal_code(X)

trap_door_algorithm(X)
  use(X,input(Y))
```

Lexicalising Concepts. Again, the lexicalisation of these concepts is straightforward leading to the following lexical conditions:

```
trap_door_algorithm
  purpose:
  structure:
  form: use(X,input(Y))
  behavior:

input
  purpose:
  structure: is_type_of(X,personal_code(Y))
  form:
  behavior:

personal_code
  purpose:
  structure:
  form:
  behavior:
```

6.6 Related Work in Conceptualisations

The availability of reliable tools aimed at the (semi-)automatic transfer of knowledge from texts to structured knowledge bases constitutes an important component of powerful software development environments. Various experiments to construct such tools have been carried out in recent years, especially in restricted domains characterised by a typical sublanguage (e.g. [Brodie et al. 84, Mylopoulos 86]). Research in the field is furthermore encouraged by the increasing availability of large quantities of machine-readable texts and lexica. Nevertheless, the results are far from being satisfactory.

Following the natural language approach, computer tools for lexicalisation as described above either acquire conceptual information by asking the user or by inferential derivation based on known semantic relationships. Derivational techniques determine the meaning of a text fragment by combining its components, from the fragment's context, or by a mixture of both.

Extracting concepts from given texts has been a very early research topic in logic programming [Kowalski 79]. Research efforts in this vein lead, for example, to methods for incremental text analysis implemented in systems like KRITON [Diederich et al. 86] and KNACK [Klinker et al. 86]. Moreover, several text understanding systems have incorporated conceptual knowledge, e.g. PETRARCA [Velardi et al. 89], RINA [Jacobs and Zernik 88], GENESIS [Mooney and DeJong 85].

Natural language processing has been used in building of knowledge-based systems, e.g. IRACQ [Ayuso et al. 87], SCISOR [Rau et al. 89]. Work on natural language interfaces for databases also leads to system components for knowledge acquisition, e.g. TELI [Ballard & Stumberger 88], TEAM [Grosz et al. 87], TQA [Damerau 85] and ASK [Thompson & Thompson 85].

Other systems are the ACME system [Kersten et al. 86] or SECSI from Bouzeghoub [Bouzeghoub & Metais 86]. While both have natural language text as input, ACME derives an extended Entity-Relationship model and SECSI generates semantic networks describing the application domain.

In the area of requirements analysis, for example, Leite proposes a natural language approach – conceptual model derivation from a lexicon [Leite & Franco 93]. The vocabulary of the application domain is defined in a lexicon. Then a formal conceptual model is derived from the lexicon using acquisition heuristics.

7 Conclusion and Future Research

The present prototypical implementation of our system proves that controlled natural language can be used for the non-trivial specification of an automated teller machine, and that the specification can be translated as coherent text into Prolog. Much more work needs to be done, however, to turn the prototype into a useful tool.

7.1 Increased Coverage of Controlled Natural Language

Simple declarative sentences and *if-then* sentences proved to be sufficiently expressive to specify the automated teller machine. Furthermore, these sentences could effectively and efficiently be translated into Prolog.

It should be noted, however, that the specification does not contain any temporal information. To specify time explicitly, e.g. in the form of sequences of events, requires first an extension of the controlled natural language by temporal constructs like *before*, *after* and *when*, second an already suggested event-oriented extension of DRT [Brown 94].

7.2 Complementary Specification Notations

Though natural language – even in a controlled form – is a universal specification notation, we believe that it is not in each case the optimal notation.

A case in point are graphical user interfaces which are much better specified directly with a graphical editor. The Explore system [Fromherz 93] provides graphical editors for the specification of both finite state machines and window-oriented user-interfaces. We plan to combine the Explore system with the current system to provide users with graphical and textual specification notations.

Another problem are algorithms. Natural language is not suited to express algorithms concisely. Evolving algebras, however, have been shown to be optimal for this task [Gurevich 91, Börger & Rosenzweig 94]. Evolving algebras use *if-then* rules to express the dynamic behaviour of systems. As it happens these rules resemble very much the *if-then* sentences of our controlled natural language. We plan to exploit the syntactic similarity to develop a notation in which *if-then* statements are either parsed as sentences in natural language, or interpreted as sentences of a formal specification language.

7.3 Knowledge Assimilation

Specifications in controlled natural language are gradually composed by a user and translated into Prolog clauses. These Prolog clauses are currently simply added to a knowledge-base. A more refined solution should employ knowledge assimilation [Kowalski 93].

The knowledge-base is considered as a (changing) theory of the domain in question. New information arrives confirming or refuting the current theory. Four cases have to be considered:

- The new Prolog clauses can be derived from the existing theory. In this case they are redundant, but the increased support given to the theory is recorded in a form which measures the degree of confirmation or utility of the Prolog clauses in the theory.
- Part of the theory can be derived from the new Prolog clauses together with the rest of the theory. In this case the derived part is redundant and can be replaced by the new Prolog clauses.

- The new Prolog clauses are inconsistent with the existing theory. In this case either the new clauses are rejected or a sub-theory is identified which participates in the proof of inconsistency and which is a candidate for revision.
- The new Prolog clauses cannot be derived from the theory. In this case they are supposed to be logically independent from the existing theory, and will therefore be added to it.

With the help of induction and abduction it is also possible to abstract away from the huge amount of factual information entered into the knowledge base and to derive more concise theoretical sentences [Kowalski 93].

7.4 Template-Based Text Generation

To lead a coherent dialog with the user we need not only to process input, e.g. specification text, queries, in controlled natural language but also to generate answers of the system, e.g. results of inferences, paraphrases, in a similar language. To avoid the difficulties of full natural language generation we will follow an approach used in the TabVer project [Glöckner et al. 94]. In this approach text is generated with the help of predefined templates containing fixed phrases and place holders for variable phrases. During text generation an appropriate template is selected and its variable parts replaced by the pertinent phrases. In the TabVer project the templates are domain-specific. We will eliminate the dependency of the templates on a specific domain, and use the linguistic lexicon, the concept lexicon and the knowledge base to provide the domain specific terminology.

References

[AECMA 85] AECMA Document: PSC-85-16598, AECMA/AIA Simplified English, A Guide for the Preparation of Aircraft Maintenance Documentation in the International Aerospace Maintenance Language, BDC Technical Services, Slack Lane, Derby, 1985.

[Anderson et al. 76] R.C. Anderson, J.W. Pichert, E.T. Goetz et al., Instantiation of General Terms, Journal of Verbal Learning and Verbal Behavior, vol. 15, pp. 667-679, 1976

[Ayuso et al. 87] D.M. Ayuso, V. Shaked, R.M. Weischedel, An Environment for Acquiring Semantic Information, 25 (24)th Annual Meeting of the ACL, Stanford, pp. 32-40, 1987

[Ballard & Stumberger 88] W. Ballard, D.E. Stumberger, Semantic Acquisition in TELI: A Transportable, User-Customized Natural Language Processor, 25th Annual Meeting of the ACL, New York, pp. 20-29, 1988

[Balzer et al. 78] R. Balzer, N. Goldman, D. Wile, Informality in Program Specifications, IEEE Transactions on Software Engineering, vol. 4, no. 2, pp. 94-103, 1978

[Barsalou 82] L.W. Barsalou, Context-Independent and Context-Dependent Information in Concepts, Memory and Cognition, no. 10, pp. 82-93, 1982

[Bierwisch & Lang 87] M. Bierwisch, E. Lang, Dimensional Adjectives: Grammatical Structure and Conceptual Interpretation, New York, Springer, 1987

[Börger & Rosenzweig 94] E. Börger, D. Rosenzweig, A Mathematical Definition of Full Prolog, Science of Computer Programming, 1994 (to appear)

[Bouzeghoub & Metais 86] M. Bouzeghoub, E. Metais, SECSI: An Expert System Approach for Database Design, IFIP Information Processing '86, North-Holland, pp. 251-257, 1986

[Brachman 79] R.J. Brachman, On the Epistemelogical Status of Semantic Networks, in: N. Findler (eds.), Associative Networks: Representation and Use of Knowledge by Computers, New York, Academic Press, 1979

[Brodie et al. 84] M. Brodie, J. Mylopoulos, J. Schmidt, On Conceptual Modeling: Perspectives from Artificial Intelligence, Data Bases and Programming Languages, New York, Springer, 1984

[Brown 94] D. W. Brown, A Natural Language Querying System Based on Discourse Representation Theory and Incorporating Event Semantics, Research Report AI-1994-03, Artificial Intelligence Center, University of Georgia, 1994

[Capindale & Crawford 89] R. A. Capindale, R. G. Crawford, Using a natural language interface with casual users, International Journal Man-Machine Studies, 32, pp. 341-362, 1989

[Covington et al. 88] M. A. Covington, D. Nute, N. Schmitz, D. Goodman, From English to Prolog via Discourse Representation Theory, Research Report 01-0024, Artificial Intelligence Programs, University of Georgia, 1988

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

[Covington 94 b] M. A. Covington. Natural Language Processing for Prolog Programmers, Prentice Hall, New Jersey, 1994

[Damerau 85] F. Damerau, Problems and Some Solutions in Customization of Natural Language Database Front Ends, ACM Transactions on Office Information Systems, vol. 3, no. 2, pp. 165-184, 1985

[Diederich et al. 86] J. Diederich, M. May, I. Ruhmann, KRITON – A Knowledge Acquisition Tool for Expert Systems, AAAI Workshop on Knowledge Acquisition for Knowledge Acquisition for Knowledge-Based System, Banff, 1986

[Dörre 91] J. Dörre. The Language of STUF, in: O. Herzog, C.-R. Rollinger (eds.): Text Understanding in LILOG: Integrating Computational Linguistics and Artificial Intelligence, Springer-Verlag, Berlin, Heidelberg, pp. 39-50, 1991

[Epstein 85] S. S. Epstein, Transportable Natural Language Processing Through Simplicity - the PRE System, ACM Transactions on Office Automation Systems, 3(2), pp. 107-120, 1985

[Fromherz 93] M. P. J. Fromherz, A Methodology for Executable Specifications – Combining Logic Programming, Object-Orientation and Multiple Views, University of Zurich, Computer Science Dept., Ph.D. thesis, 1993

[Fuchs & Fromherz 94] N. E. Fuchs, M. P. J. Fromherz, Transformational Development of Logic Programs from Executable Specifications – Schema-Based Visual and Textual Composition of Logic Programs, in C. Beckstein, U. Geske (eds.), Entwicklung, Test und Wartung deklarativer KI-Programme, GMD Studien Nr. 238, Gesellschaft für Informatik und Datenverarbeitung, 1994

[Gazdar et al. 85] G. Gazdar, E. Klein, G. Pullum, I. A. Sag, Generalized Phrase Structure Grammar, Blackwell, Oxford, 1985

[Gazdar & Mellish 89] G. Gazdar, Chris Mellish, Natural Language Processing in Prolog, An Introduction to Computational Linguistics, Wokingham, Addison-Wesley, 1989

[Glöckner et al. 94] J. Glöckner, A. Grieszl, M. Müller, M. Ronthaler, TabVer: A Case Study in Table Verbalization, in: B. Nebel, L. Dreschler-Fischer (eds.), Proceedings of KI-94: Advances in Artificial Intelligence, Saarbrücken, Springer-Verlag, pp. 94-105, 1994

[Grimshaw 90] J. Grimshaw, Argument Structure, Cambridge, MIT Press, 1990

[Grosz et al. 87] B.J. Grosz, D.E. Appelt, P.M. Appelt et al., TEAM: An Experiment in the Design of Transportable Natural-Language Interfaces, Artificial Intelligence, vol. 33, pp. 172-243, 1987

[Gurevich 91] Y. Gurevich, Evolving Algebras: An Attempt to Discover Semantics, Bull. EATCS, 43, February 1991, pp. 264 - 284, 1991

[Hoare 87] C. A. R. Hoare, An overview of some formal methods for program design, in: C. A. R. Hoare, C. B. Jones, Essays in Computing Science, Prentice Hall, pp. 371-387, 1987

[Hofmann 93] H.F. Hofmann, Requirements Engineering: A Survey of Methods and Tools, TR 93.05, Department of Computer Science, University of Zurich, 1993

[Jacobs & Zernik 88] P. Jacobs, U. Zernik, Acquiring Lexical Knowledge from Text: A Case Study, 7th National Conference on Artificial Intelligence, St. Paul, Minnesota, pp. 739-744, 1988

[Kamp 81] H. Kamp. A theory of truth and semantic representation, in: J. A. G. Groenendijk, T. M. V. Janssen, M. B. J. Stokhof (eds.): Formal Methods in the Study of Language, Mathematical Center Tract 135, Amsterdam, pp. 277-322, 1981

[Kamp & Reyle 93] H. Kamp, U. Reyle, From Discourse to Logic, Introduction to Modeltheoretic Semantics of Natural Language, Formal Logic and Discourse Representation Theory, Kluwer Academic Publishers, Dordrecht, 1993

[Kaplan & Bresnan 82] R. M. Kaplan, J. Bresnan. Lexical-Functional Grammar: A Formal System for Grammatical Representation, in: J. Bresnan (ed.): The Mental Representation of Grammatical Relations, MIT Press, Cambridge, Mass., 1982

[Katz & Fodor 63] J.J. Katz, J. Fodor, The Structure of a Semantic Theory, Language, vol. 39, pp. 170-210, 1963

[Kersten et al. 86] M.L. Kersten, H. Weigand, F. Dignum et al., A Conceptual Modelling Expert System, 5th International Conference on Entity-Relationship Approach, Dijon, pp. 275-288, 1986

[Klinker et al. 86] G. Klinker, S. Bentolila, M. Genetet et al., KNACK – Report Driven Knowledge Acquisition, AAAI Workshop on Knowledge Acquisition for Knowledge Acquisition for Knowledge-Based Systems, Banff, 1986

[Kowalski 79] R. Kowalski, Logic for Problem Solving, Amsterdam, North-Holland, 1979

[Kowalski 85] R. A. Kowalski, The relation between logic programming and logic specification, in: C. A. R. Hoare, J. C. Shepherdson (eds.), Mathematical Logic and Programming Languages, Prentice-Hall International, 1985

[Kowalski 90] R. A. Kowalski, English as a Logic Programming Language, in: New Generation Computing, 8, pp. 91-93, 1990

[Kowalski 92] R. A. Kowalski, Legislation as Logic Programs, in: G. Comyn, N. E. Fuchs, M. J. Ratcliffe (eds.), Logic Programming in Action, Lecture Notes in Artificial Intelligence 636, pp. 203-230, 1992

[Kowalski 93] R.A. Kowalski, Logic without Model Theory, Technical Report, Imperial College, 1993

[Kramer & Mylopoulos 92] B. Kramer, J. Mylopoulos, Knowledge Representation, in: S. C. Shapiro (ed.), Encyclopedia of Artificial Intelligence, Wiley, 1992

[Lakoff 87] G. Lakoff, Woman, Fire, and Dangerous Things, Chicago, University of Chicago Press, 1987

[Leite & Franco 93] J.C. Leite, A.P. Franco, A Strategy for Conceptual Model Acquisition, in: S. Fickas and A. Finkelstein (eds.), IEEE International Symposium on Requirements Engineering, San Diego, IEEE Computer Society Press, pp. 243-246, 1993

[Lloyd 94] J. Lloyd, Practical Advantages of Declarative Programming, Invited Lecture, GULP-PRODE '94, Peñiscola (Spain), September 1994

[Lyons 81] J. Lyons, Language, Meaning and Context, London, Fontana, 1981

[Macias & Pulman 92] B. Macias, S. Pulman, Natural Language Processing for Requirements Specifications, in: F. Redmill, T. Anderson (eds.), Safety-Critical Systems, Current Issues, Techniques and Standards, Chapman & Hall, pp. 67-89, 1993

[Medin & Smith 84] D.L. Medin, E.E. Smith, Concepts and Concept Formation, Annual Psychological Review, vol. 35, pp. 113-138, 1984

[Miller & Johnson-Laird 76] G.A. Miller, P.N. Johnson-Laird, Language and Perception, Cambridge, MIT Press, 1976

[Mooney & DeJong 85] R. Mooney, G. DeJong, Learning Schemata for Natural Language Processing, 9th IJCAI, pp. 681-687, 1985

[Mylopoulos 86] J. Mylopoulos, The Role of Knowledge Representation in the Development of Specifications, in: H.-J. Kugler (eds.), IFIP Information Processing '86, Amsterdam, Elsevier, 1986

[Pereira & Warren 80] F. C. N. Pereira, D. H. D. Warren, Definite Clause Grammars for Language Analysis – a Survey of the Formalism and a Comparison with Augmented Transition Networks, in: Artificial Intelligence, 13, pp. 231-278, 1980

[Pollard & Sag 94] C. Pollard, I. A. Sag. Head-Driven Phrase Structure Grammar, Center for the Study of Language and Information, Stanford, Chicago Press, Chicago, London, 1994

[Pulman 94] S. G. Pulman. Natural Language Processing and Requirements Specification, Presentation at the Prolog Forum, Department of Computer Science, University of Zurich, February 1994

[Quillian 68] M.R. Quillian, Semantic Memory, in: M. Minsky (eds.), Semantic Information Processing, Cambridge, MIT Press, 1968

[Rau et al. 89] L. Rau, P. Jacobs, U. Zernik, Information Extraction and Text Summarization Using Linguistic Knowledge Acquisition, Information Processing and Management, vol. 25, no. 4, pp. 419-428, 1989

[Rosch & Loyd 78] E. Rosch, B.B. Loyd, Cognition and Categorization, Hillsdale, Erlbaum, 1978

[Saussure 66] F. Saussure, Course in General Linguistics, New York, McGraw Hill, 1966

[Schank 75] R.C. Schank, Conceptual Information Processing, Amsterdam, North-Holland, 1975

[Shieber et al. 83] S. M. Shieber, H. Uszkoreit, F. C. N. Pereira, J. J. Robinson, M. Tyson, The Formalism and Implementation of PATR-II, in: Research on Interactive Acquisition and Use of Knowledge, Artificial Intelligence Center, SRI International, Menlo Park, Cal., 1983

[Shieber 86] S. M. Shieber, An Introduction to Unification-Based Approaches to Grammar, CSLI Lecture Notes 4, Center for the Study of Language and Information, Stanford University, Cal., 1986

[Sommerville 92] I. Sommerville, Software Engineering, Fourth Edition, Addison-Wesley, Wokingham, 1992

[Sterling 92] L. Sterling, A Role for Prolog in Software Engineering, Computer Science Colloquium, Department of Computer Science, University of Zurich, 1992

[Sterling 94] L. Sterling, Prolog for Software Engineering, Tutorial at the Second International Conference on the Practical Applications of Prolog PAP'94, London, April 1994

[Thompson & Thompson 85] B. Thompson, F. Thompson, ASK Is Transportable in Half a Dozen Ways, ACM Transactions on Office Information Systems, vol. 3, no. 2, pp. 185-203, 1985

[Vasey 89] P. Vasey, flex Expert System Toolkit, Version 1.2, Logic Programming Associates, London, 1989

[Velardi et al. 89] P. Velardi, M.T. Pazienza, S. Magrini, Acquisition of Semantic Patterns from a Natural Language Corpus of Texts, SIGART Newsletter, vol. 108, pp. 115-123, 1989

[Wilks 75] Y. Wilks, A Preferential Pattern Seeking Semantics for Natural Language Inference, Artificial Intelligence, no. 6, pp. 53-74, 1975

[Williams 85] J. Williams, Style – Ten Lessons in Clarity and Grace, Scott, Foremann Co., 1985

Appendix A: SimpleMat (Plain English Version)

This appendix contains the plain English version of the specification of a simple automated teller machine called *SimpleMat*. The specification describes the user interface, and some of the mechanisms behind the user interface.

Cards

The customer has a card that carries the following machine-readable data: the card number, the check code, and the maximum amount of money that can be withdrawn.

Personal Code

Customers have personal codes for their cards. The personal code is a fixed length (n) number that is used by a trap door algorithm to calculate a number which should be identical to the check code on the card.

Access

The customer introduces his card, which is read by the SimpleMat. The card number is checked for validity and whether it is currently active. If the card cannot be read, has an invalid card number, or is not currently active it is rejected with the message 'Invalid Card'. Otherwise the access door of the SimpleMat is opened and the message 'Enter Your Personal Code' is displayed. The customer enters the personal code. The code is checked for validity by a trap door algorithm that should result in the check code stored on the card. If the personal code entered is not correct the SimpleMat displays the message 'Incorrect Code. Transaction Terminated. Card Retained' and the door of the SimpleMat is closed. The SimpleMat retains the card.

Transactions

If the correct code has been entered the SimpleMat displays the following message

- 'Withdraw Money Without Receipt - Key A'
- 'Withdraw Money With Receipt - Key B'
- 'Terminate Transaction - Terminate'

Selection B will not be displayed if there is no paper to print the receipt.

By hitting the appropriate key the customer can execute one of the following transactions.

- Key A: Withdraw money without receipt.

  The customer pushes key A and the message 'Enter Amount To Be Withdrawn' is displayed. If the amount entered exceeds the maximum amount the SimpleMat displays 'Enter Smaller Amount' and the customer can reenter the amount s/he wants to withdraw. If the amount entered does not exceed the maximum amount the SimpleMat displays 'Amount is Being Prepared. Retract Card' and expels the card. Finally, the amount of money is prepared and presented to the customer. Then the SimpleMat closes the door.
- Key B: Withdraw money with a receipt.

  As in point 1, only that after expelling the card and before the money is presented a receipt is printed and expelled which contains the amount of money withdrawn, the card number, the SimpleMat number and address, and the time and date. The SimpleMat closes the door.
- Terminate: Terminate transaction.

If the customer pushes the key 'Terminate' the message 'Transaction Terminated. Retract Card' is displayed, and the card is expelled. The SimpleMat closes the door.

Appendix B: SimpleMat (Controlled English Version)

This appendix contains the controlled English version of the specification of a simple automated teller machine called *SimpleMat*. The specification describes the user interface, and some of the mechanisms behind the user interface.

% General

SimpleMat is a simple money-dispenser.
It has a user-interface.

% Cards

The customer has a card that carries machine-readable data.
The card has a card-number.
The card has a check-code.
The card has a maximum limit.
The customer withdraws an amount of money that does not exceed the maximum limit.

% Personal Code

Every customer has a personal-code for the card.
The personal-code is a number of fixed length.

A trap-door-algorithm uses the personal-code as input.
If the trap-door-algorithm calculates a number
then the number equals the check-code.

% Access

The customer introduces the card.
SimpleMat reads the card.

If the card is not readable
then SimpleMat rejects the card
and SimpleMat gives the message 'Invalid Card'.

SimpleMat checks the validity of the card-number.
If the card-number is not valid
then SimpleMat rejects the card
and SimpleMat gives the message 'Invalid Card'.

SimpleMat checks the activity of the card-number.
If the card-number is not active
then SimpleMat rejects the card
and SimpleMat gives the message 'Invalid Card'.

If (none of the above conditions hold) SimpleMat opens the access-door.
SimpleMat displays the message 'Enter Your Personal Code'.

The customer enters the personal-code.

SimpleMat checks the correctness of the personal-code.
If the trap-door-algorithm results in the check-code
then the personal-code is correct.

If the personal-code is not correct

then SimpleMat displays the message
'Incorrect Code. Transaction Terminated. Card Retained'
and SimpleMat closes the door
and SimpleMat retains the card.

% Transactions

If the customer enters the correct code
then SimpleMat displays the message
'Withdraw Money Without Receipt - Key A'
and 'Terminate Transaction - Terminate'.

If the printer has paper for the receipt
then SimpleMat displays the message
'Withdraw Money With Receipt - Key B'
and activates receipt-printing.

If the customer pushes the key 'A'
then SimpleMat displays the message
'Enter Amount To Be Withdrawn'.

If the entered amount is higher than the maximum amount
then SimpleMat displays 'Enter Smaller Amount'
and the customer can reenter another amount.

If the entered amount is smaller or equal
than the maximum amount
then SimpleMat displays
'Amount is Being Prepared. Retract Card'
and SimpleMat expels the card
and SimpleMat prepares the amount of money
and SimpleMat presents the amount of money to the customer
and SimpleMat closes the door.

If the customer pushes the key 'B'
then SimpleMat displays the message
'Enter Amount To Be Withdrawn'.

A receipt contains the entered amount of money.
A receipt contains the card-number.
A receipt contains the SimpleMat number.
A receipt contains the SimpleMat address.
A receipt contains the time.
A receipt contains the date.

If the entered amount is higher than the maximum amount
then SimpleMat displays 'Enter Smaller Amount'
and the customer enters another amount.

If the entered amount is smaller or equal
than the maximum amount
then SimpleMat displays
'Amount is Being Prepared. Retract Card'
and SimpleMat expels the card
and SimpleMat prepares the receipt

and SimpleMat prepares the amount of money
and SimpleMat presents the amount of money to the customer
and SimpleMat closes the door.

If the customer pushes the key 'Terminate'
then SimpleMat displays the message
'Transaction Terminated. Retract Card'
and SimpleMat expels the card
and SimpleMat closes the door.