CM-Builder: A Natural Language-based CASE Tool

H.M. Harmain (hmharmain@hotmail.com)
Dept. of Computer Science
University of Sebha, Libya

R. Gaizauskas (r.gaizauskas@dcs.shef.ac.uk)
Dept. of Computer Science
University of Sheffield, UK

Abstract. Graphical CASE (Computer Aided Software Engineering) tools can provide considerable help in documenting the output of the Analysis and Design stages of Object-Oriented software development and can assist in detecting incompleteness and inconsistency in an analysis. However, these tools do not contribute to the initial, difficult stage of the analysis process, that of identifying the object classes, attributes and relationships used to model the problem domain. This paper describes an NL-Based CASE tool called CM-Builder which aims at supporting this aspect of the Analysis stage of software development in an Object-Oriented framework. CM-Builder uses robust Natural Language Processing techniques to analyse software requirements texts written in English and construct, either automatically or interactively with an analyst, an initial UML Class Model representing the object classes mentioned in the text and the relationships among them. The initial model can be directly input to a graphical CASE tool for further refinement by a human analyst. CM-Builder has been quantitatively evaluated in blind trials against a collection of unseen software requirements texts and we present the results of this evaluation, together with the evaluation methodology. The results are very encouraging and demonstrate that tools such as CM-Builder have the potential to play an important role in the software development process.

1. Introduction

Object-Oriented Technology (OOT) has become a popular approach for building software systems. This technology has recently been extended from the programming phase of the software development to cover the earlier phases of *Analysis* and *Design*. Many object-oriented methods have been proposed for the analysis and design phases (e.g. Booch, OMT, Objectory, OOSE). In these methods, the Object-Oriented Analysis process is considered one of the most critical and difficult tasks (Booch, 1994). It is critical because subsequent stages rely on it, and it is difficult because most of the input to this process is given in natural languages (such as English) which are inherently ambiguous

Graphical CASE (Computer Aided Software Engineering) tools can provide considerable help in documenting and analysing the output of Analysis and Design, and can assist in detecting inconsistency and incompleteness in an analysis. However, they do not contribute to the initial, difficult stage of the analysis process, that of identifying the object classes, attributes and relationships which will figure in modelling the problem domain. To contribute to this part of the analysis process a CASE tool has to be able to deal with human languages which software engineers use as media to understand the problems they are addressing. The Artificial Intelligence (AI) subfield of Natural Language Processing (NLP) suggests promising approaches that may help software engineers in the analysis phase of software development.

While other researchers have proposed NLP-based approaches to analysing software requirements documents (see section 2), in this paper we make two original contributions. First, we describe a robust, domain independent CASE tool which analyses software requirements written in English and produces, either in interaction with a human analyst or fully automatically, first-cut class models represented in the Unified Modeling Language (UML). These models can be directly input to graphical CASE tools by which human analysts can further refine and extend them. Second, we specify an evaluation methodology for quantitatively evaluating NL-based CASE tools, and use this methodology to evaluate the automatic output of our system.

The rest of this paper is structured as follows: section 2 provides background information and describes related work; section 3 gives a general overview of our approach; section 4 describes in detail our fully implemented NL-based CASE tool, both the interactive version and the automated version; section 5 shows the results of using the automated system to analyse a case study; section 6 discusses the evaluation methodology and presents results of evaluating the automated system; section 7 presents conclusions and discusses future work.

2. Background and Related Work

The past two decades have seen growing interest in AI-based CASE technology. Our review of this area shows that there have been two main approaches to providing automated tools that support the early stages (mainly *analysis* and *design*) of software development.

The first approach is based on applying general AI techniques, such as schema-based reasoning and search strategies, to create intelligent tools. This approach claims that *analysis* and *design* are very knowledge intensive activities, and should be supported by AI-based tools, mostly Rule-Based systems. Examples of such systems can be found in Harmain (2000).

The second approach to providing automated tools that support the early stages of software development is based on the analysis of natural languages (human languages). This approach realises that most software requirements data available to software engineers are expressed in natural languages. The system described in this paper (see section 3) also follows this approach, but before we describe our system we provide a brief survey of what other people have already done.

Abbott (1983) proposes a linguistic based method for analysing software requirements, expressed in English, to derive basic data types and operations. This approach was further developed by Booch (1986). Booch describes an Object-Oriented Design method where nouns in the problem description suggest objects and classes of objects, and verbs suggest operations. However, both Abbott and Booch recognise the importance of semantic and real-world knowledge in the analysis process:

> although the steps we follow in formalising the strategy may appear mechanical, it is not an automatic procedure... it requires a great deal of real world knowledge and an intuitive understanding of the problem (Abbott, 1983).

Saeki et al. (1987) describe a process of incrementally constructing software modules from object-oriented specifications obtained from informal natural language requirements. Their system analyses the informal requirements sentence at a time. Nouns and verbs are automatically extracted from the informal requirements but the system can not determine which words are important for the construction of the formal specification. Hence an important role is played by the human analyst who reviews and refines the system results manually after each sentence is processed.

Dunn and Orlowska (1990) describe a natural language interpreter for the construction of NIAM (Nijssen's, or Natural-language, Information Analysis Method) conceptual schemas. The construction of conceptual schemas involves allocating surface objects to entity types (semantic classes) and the identification of elementary fact types. The system accepts declarative sentences only and uses grammar rules and a dictionary for type allocation and the identification of elementary fact types.

Meziane (1994) implemented a system for the identification of VDM data types and simple operations from natural language software requirements. The system first generates an Entity-Relationship Model (ERM) from the input text and then generates VDM data types from the ERM.

Mich and Garigliano (1994) and Mich (1996) describe an NL-based prototype system, NL-OOPS, that is aimed at the generation of object-oriented analysis models from natural language specifications. This system demonstrated how a large scale NLP system called LOLITA can be used to support the OO analysis stage.

Some researchers, also advocating NL-based systems, have tried to use a controlled subset of a natural language to write software specifications and build tools that can analyse these specifications to produce useful results. Controlled natural languages are developed to limit the vocabulary, syntax and semantics of the input language. Attempto (Fuchs et al., 1998), ASPIN, an Automatic Specifications Interpreter (Cyre, 1995), and the Requirements Analysis Support System described in (Belkhouche and Kozma, 1993) are examples of systems that analyse subset of a natural language (English). Macias and Pulman (1995) also discuss some possible applications of NLP techniques, using the CORE Language Engine, to support the activity of writing unambiguous formal specifications in English.

This research work has provided valuable insights into how NLP can be used to support the analysis and design stages of software development. However, each of these approaches has weaknesses which mean that as yet NL-based CASE tools have not emerged into common use for OO analysis and design. Abbott and Booch's work describes a methodology, but they have not produced a working system which implements their ideas. Saeki et al. and Meziane both produced working systems, but these systems arguably require an unacceptably high level of user interaction (accepting or rejecting noun phrases to be represented in the final model on a sentence by sentence basis as the requirements document is processed). Dunn and Orlowska's work is not directly relevant to OO analysis and design, as they focus on another analysis method. The controlled language approaches have been implemented and appear to work, but they have the drawback that authors of software requirements documents must learn and use a specialised language, albeit a subset of natural language. Mich and Garigliano's approach, which is closest to our own, is reliant on the coverage of a very large scale knowledge base and the impact of (inevitable) gaps in this knowledge base on the ability of the system to generate usable class models is unclear.

It is also worth noting that none of these systems, so far as we are aware, has been evaluated on a set of previously unseen software requirements documents from a range of domains. This surely ought to become a mandatory methodological component of any research work in this area, as it has in other areas of language processing technology, such as the DARPA-sponsored Message Understanding Conferences

(Hirschman, 1998) or the NIST-sponsored Text Retrieval Conferences (Voorhees and Harman, 1999) (for discussion of a broader set of areas where evaluation methodologies are being developed for speech and language technology, see Gaizauskas (1997)).

3. The CM-Builder Approach: Overview

Drawing on the work described in the preceding section, our perspective is this. First, no automated NL-based CASE tool that aims to *replace* the analyst is likely to be successful at present, given the current imperfect state of language processing technology. What is needed is a tool that can *assist* the analyst by making proposals that he or she can refine in an effective manner. Our proposal is that this should be done by a tool which proposes an initial model in a commonly accepted format (UML class diagrams) that a human analyst can then further refine using existing CASE tools. This tool should be robust and domain independent, to allow for widespread use, but be capable of taking advantage of advances in language processing research, so as to improve its performance. Finally, the tool should be evaluated using a well-defined methodology against blind test data, so that potential users can assess the level at which the technology performs, and so that alternate approaches and proposed enhancements can be objectively compared.

The Class Model Builder (CM-Builder) is a modular NL-based CASE tool. The CM-Builder approach consists of four stages as illustrated in figure 3. These stages can be summarized as follows.

1. Obtain a set of functional requirements or problem description in natural language.

2. Use an NLP system to syntactically and semantically analyse the informal requirements text and keep all the intermediate analysis results for further analysis.

3. Use the results produced by the NLP system to extract object classes, their attributes, and the relationships among them.

4. Produce a first-cut static structure model of the system from the extracted objects in a standard format and use a graphical CASE tool to manually refine the initial model.

Depending on the level of analyst interaction versus automation which is desired or acceptable and the sophistication and accuracy of

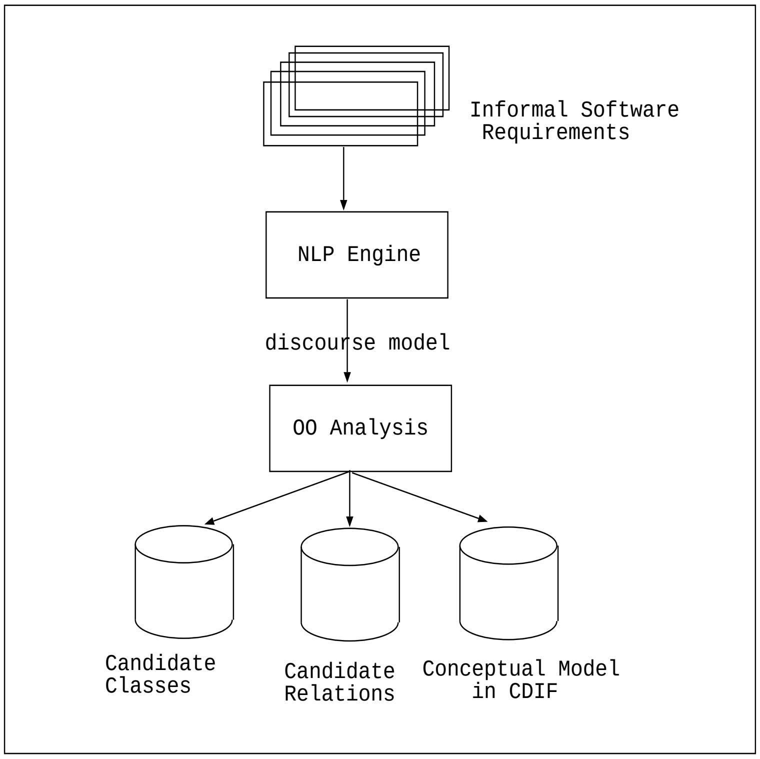


Figure 1. The CM-Builder Approach

analysis of which the core NLP system is capable, a variety NLP-based CASE tools which conform to this broad approach can be designed. We distinguish here three possible levels of NL analysis and the sort of OO CASE tools that could be designed to utilise them, discussing as well their potential strengths and weaknesses.

3.1. Surface Analysis

At the level of least sophisticated, but most reliable, NL processing, one can envisage a system which uses shallow syntactic knowledge to produce lists of candidate classes and relationships which are then filtered by a human analyst to choose the appropriate model elements.

As has been observed by Abbott (1983) and Booch (1986), candidate classes can frequently be found in the requirements text by considering the noun phrases. Likewise, candidate relationships can be found by considering verb phrases. The simplest way of doing this is by using shallow parsing techniques to analyze the requirements text for base noun phrases (noun phrases without post-modifiers such as prepositional phrases and relative clauses) and core verb groups (main verb plus auxiliaries and possibly following particle). For example, by analysing the sentence *A library issues loan items to customers* we can extract three candidate classes (*library*, *loan items*, and *customers*) and one candidate relationship (*issues*).

Shallow or so-called "chunking" parsers can now achieve high levels of accuracy at such tasks – for example Cardie and Pierce (1998) report levels of 94% precision and recall in identifying base noun phrases in unseen test data. As well as identifying such phrases, an automated system can provide additional support by performing phrase frequency analysis across a text to identify the most frequently mentioned entities and ranking them before presenting them to the analyst.

We call this process *Surface Analysis* because it does not require any understanding, whatsoever, of the problem domain. This approach has the advantage of quickly and accurately producing lists of candidate classes and candidate relationships from which the analysis process starts. However, it must not be assumed to be unproblematic as there are different ways of expressing the same statement. For example, the two sentences *The actors planned their performance* and *The acting group had a plan about how to perform* express the same thing. Furthermore, it requires a high level of analyst interaction. Not only must spurious candidates be rejected and variants merged, but since this approach does not specify which classes are related by which relationships, the analyst must perform this association manually. So, in the example above, while *issues* is identified as a candidate relationship and *library*, *loan items*, and *customers* as candidate classes, this approach could not propose that *issues* is a relationship holding between these classes.

3.2. Deep Analysis

To not merely identify candidate classes, relationships and attributes, but to identify which classes are related by which relationships, and which attributes hold of which classes, and to do so completely and accurately, requires large amounts of syntactic, lexical and world knowledge, as well as significant inference capabilities. In other words, in the general case, such a goal presupposes full language understanding, and is therefore an "AI-complete" problem.

Applications that require language understanding capabilities have made progress where the applications have been limited to specific domains. For example, reasonably successful information extraction systems have been constructed for specific domains as diverse as corporate management succession events, product and joint venture announcements, and terrorist attacks (Cowie and Lehnert, 1996; Gaizauskas and Wilks, 1998). Domain specific text analysis needs large amounts of knowledge from the application domain to be encoded in the system. So, for example, in the library system domain we might need to know about people involved in the system (e.g. borrowers and library staff), what material can be borrowed, and who can borrow what.

Consideration of domain dependent analysis as an approach to supporting OO Analysis reveals a number of major problems:

- Building a domain model is at present a highly labour intensive manual process.
- Though knowledge reuse is an active research topic (see, e.g. Gomez-Perez and Benjamins (1999), in practice domain models have to be created more or less from scratch for each new domain.
- The final discourse model derived by the system from the text will not substantially differ in its *conceptual* content and structure from the domain model encoded in advance in the system. Where it will differ is in its knowledge of the behaviour or properties of instances reported in the text. So, an information extraction system with a domain model appropriate for management succession events will not alter its underlying conceptual model after reading a text; rather it will supplement this model with knowledge about instances reported in the text – e.g person X has left position P in company C to join company D. But in processing software requirements texts it is precisely a conceptual model for a domain that we are trying to acquire. Presupposing one leads us into circularity.

Given these problems we have decided to abandon, for the time being, any further investigation of a deep analysis approach, either domain dependent or domain independent.

3.3. Domain Independent Semantic Analysis

Given the limitations of surface analysis and the intractability of deep analysis, it is appropriate to ask whether there is any intermediate level of language analysis that may increase the functionality that can be offered to OO Analyst, without succumbing the problems mentioned in the last section. We believe that there is and that an NL system can deliver a domain independent "semantic" analysis, which is partial, but still of use in the context of the assistive technology which are proposing.

In essence this approach relies on two things. First, richer syntactic analysis than simple base noun phrase and core verb group analysis is possible *in some cases* and this analysis can with reasonable reliability suggest semantic relations. So, for example, in many English sentences logical subjects and objects of verbs can be detected on the basis of relatively general syntactic knowledge, and can be used to suggest, in the context of software requirements texts, which classes stand in what relationships to what other classes. Or, possessive relations can be used to hypothesise attributes of classes – for example *title of the book* or *book's title* suggest that `title` should be regarded as an attribute of `book`.

Second, while retaining domain independence we can recognise that certain words and concepts occur frequently across software requirements texts. In other words, there is a genre of software requirements documents, which is at least partly identifiable by its common vocabulary and concepts. For example, words or phrases pertaining to aggregation (*is made up of, is composed of, contains*) or subtyping (*is a kind of, is an X which/that Ys*) are frequently found in requirements texts. These words and concepts can be modelled more extensively, and do provide a starting point for extracting and proposing class models from texts.

Thus, our hypothesis is that a useful NL-based OO Analysis tool can be built which utilises this intermediate level of NL analysis. It still aspires only to produce an initial conceptual model which must be refined by a human analyst. But it provides a richer starting point than the surface analysis approach by proposing which classes stand in what relationships to what other classes, and which attributes hold of which classes, and by proposing multiplicities on relationships. Thus, the analyst starts with a full-fledged conceptual model rather than just a catalogue of candidate classes and relationships.

4. The CM-Builder: Detailed Description

We have designed and implemented two versions of CM-Builder, each of which is meant to support a specific level of analysis as discussed in the previous section. The first version, CM-Builder 1, is intended to support OOA by performing surface analysis as discussed in section 3.1, whereas the second version, CM-Builder 2, is intended to carry out domain independent semantic analysis as discussed in section 3.3. Both systems are implemented in the GATE environment (Gaizauskas et al., 1996; Cunningham et al., 1996). GATE, the General Architecture For Language Engineering, is a rapid application development environment for Language Engineering (LE) applications which provides tools for data visualisation, debugging and evaluation of LE modules

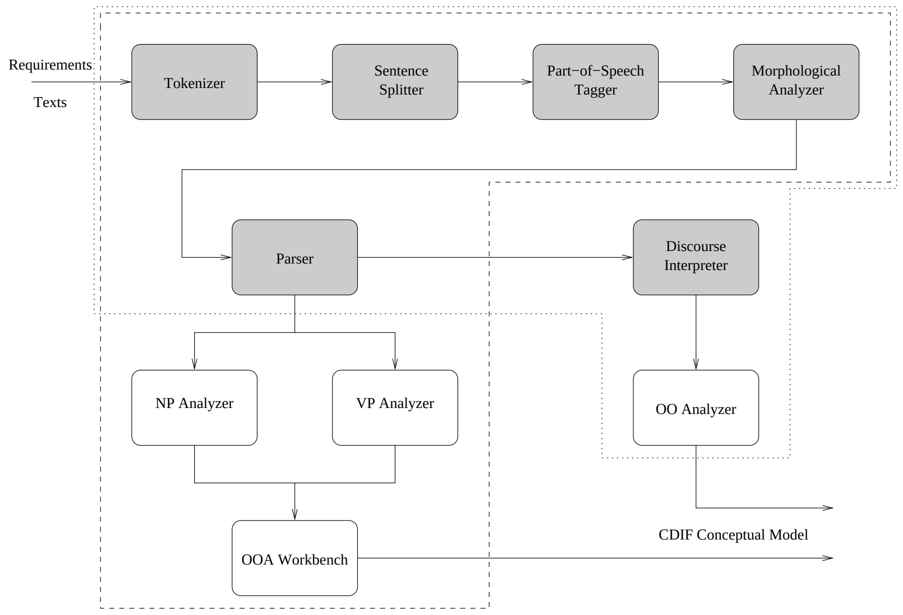


Figure 2. CM-Builder

and systems. The core NLP system used in CM-Builder was originally built for the LaSIE (Large Scale Information Extraction) system at Sheffield (Gaizauskas and Humphreys, 1997; Humphreys et al., 1998) and is embedded in GATE. This system is highly robust, and has been successfully used to process millions of words of text. While by no means a perfect analyser, it can be relied upon to produce reasonable (partial) analyses for a wide range of input texts.

Both versions of CM-Builder take as input plain text file containing a software requirements specification in English and produce as output an initial OO Conceptual Model of the problem being analysed, represented as a CDIF (CASE Date Interchange Format) file (Ernst, 1996). This output file contains the identified object classes, their attributes, and the relationships among them. The main advantage of producing the output in a standard format is that any graphical CASE tool supporting this format can be used to import the CM-Builder results. We have used the SELECT Enterprise modeler V5.0 to test our system.

Figure 2 shows the overall architecture of the CM-Builder. The core NLP systems modules are shaded; the modules which comprise CM-Builder 1 are surrounded by a dashed line; those comprising CM-Builder 2 are surrounded by a dotted line.

In the rest of this section we discuss first the core NLP system underlying CM-Builder and then the design of CM-Builder 1 and 2, in turn.

4.1. The Core NLP System

The core NLP system processes text in three main processing stages: lexical preprocessing, parsing, and discourse interpretation.

4.1.1. *Lexical Preprocessing*

The Lexical Preprocessor consists of four modules namely, a tokenizer, sentence splitter, tagger, and a morphological analyser. The input to the lexical preprocessor is a plain ASCII file containing a description of the problem at hand (informal requirements). The output is a set of charts, one per sentence, written as Prolog terms to be used directly by the parser which is written in Prolog. The processing steps are carried out in the following order:

1. Tokenization: The tokenizer takes a plain text file as input and splits it into tokens. This includes, e.g., separating words and punctuation, identifying numbers, and so on.

2. Sentence Splitting: The sentence splitter identifies sentence boundaries within the given text.

3. Part-of-Speech (POS) Tagging: The POS tagger assigns to each token in the input one of 48 POS tags . We have used a slightly modified version of the Brill tagger (Brill, 1994).

4. The Morphological Analysis: After POS tagging, all nouns and verbs are passed to the morphological analyser which returns the root and suffix of each word. For example, a plural noun like "boxes" will be analysed as "box + s", and an inflected verb form like "framing" will be analysed as "frame + ing". These roots and suffices are included in the input to the parser.

4.1.2. *Parsing*

We have used a bottom-up chart parser implemented in Prolog, an enhanced version of the parser is described in Gazdar and Mellish (1989). It uses a feature-based phrase structure grammar and relies on unification of feature structures to enforce grammatical constraints during parsing and to build semantic representations (as in Pereira and Shieber (1987)). The parser takes the output of the Lexical Preprocessor, and, using the grammar rules, builds a syntactic tree and in parallel generates a semantic representation for every sentence in the text. The semantic representation is simply a predicate-argument structure (first order logical terms). The morphological roots of the simple verbs and nouns are used as predicate names in the semantic representations. Tense and numbers features are translated directly into this notation where appropriate. All NPs and VPs introduce unique instance constants in the semantics which serve as identifiers for the objects or events referred to in the text. For example, *A library issues loan items.* will map to something like:

```
issue(e1),time(e1,present),aspect(e1,simple),
   voice(e1,active),lsubj(e1,e2),lobj(e1,e3),
library(e2),determiner(e2,a),number(e2,sing),
item(e3),loan(e4),qual(e3,e4),number(e3,plural).
```

The parser is a partial parser, which means it produces correct but not necessarily complete syntactic structures and hence semantic representations. If a full parse of a sentence is not found, the parser uses a 'best' parse algorithm to choose the best complete sub-structures (i.e. phrases of category). In CM-Builder 1, we use only the syntactic output of the parser; the semantic output is used in CM-Builder 2.

4.1.3. *Discourse interpretation*

This module reads the meaning representation of every sentence produced by the parser and adds it to a predefined 'world' model to produce a final model specific to the processed text called the discourse model.

The meaning representation is translated into a representation of instances, their ontological classes and their attributes in the XI knowledge representation language, a language which allows straightforward definition of cross-classification hierarchies and the association of arbitrary attributes with classes or instances in the hierarchy (Gaizauskas, 1995).

The definition of a cross-classification hierarchy in XI is called an *ontology.* The ontology together with an association of attributes with nodes in the ontology form a world model. It serves as a declarative knowledge base that contains the background information the system has about the world. An example of a simple world model is given in figure 4.1.3. This model shows four object classes (namely, library, customer, student, and professor), one event class, and two attributes. The customer class is shown to be a superclass of the student and professor subclasses. Also two attributes, name and address, are attached to this class.

Knowledge from the input text is added to the world model in order to extend it into a model specific to the text called a *Discourse Model.* The discourse interpreter takes the semantic representation of each sentence and applies four processing stages as follows:

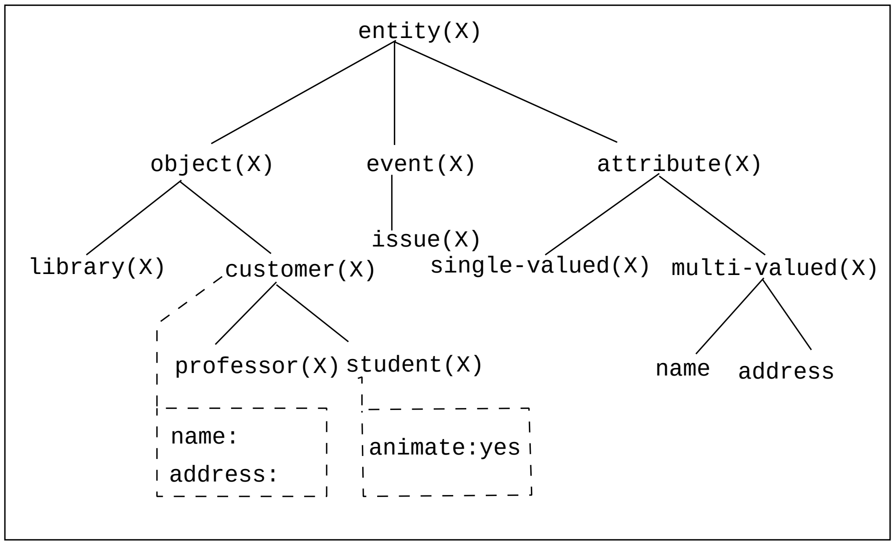


Figure 3. A simple world model

1. Adding Instances and Attributes to the World Model: In this stage the Discourse Interpreter adds *simple noun phrase* instances and their attributes under the *Object* node in the world model, and *verb* instances and their attributes under the event node.

2. Presupposition Expansion: Any presupposition rules attached to nodes to which the new instances or attributes have been added are evaluated, and as a result additional instances or attributes may be added to or removed from the discourse model. For example, an agent-less passive such as 'A book can be borrowed' might cause an anonymous object of type 'person' to be added to the discourse model as the agent of the borrow event, which may prove useful in interpreting the rest of the text.

3. Coreference Resolution: After adding the semantics of a new sentence to the world model, and expanding any presuppositions, all newly added instances are compared with previously added ones to see if any two instances can be merged into a single one, representing a coreference resolution. This is necessary to link pronominal references (such as *he, it*) or definite noun phrases (*the user*) to earlier referents in the text. A full description of this algorithm can be found in Gaizauskas and Humphreys (2000).

4. Consequence Expansion: In this stage the Discourse Interpreter checks to see if any rules may be applied as a consequence of merging instances in the previous stage – this may again lead to

additional instances or attributes being added to or removed from the discourse model.

Having finished this stage the Discourse Interpreter goes back to the first stage and take the meaning representation of the next sentence. All sentences go through the same stages.

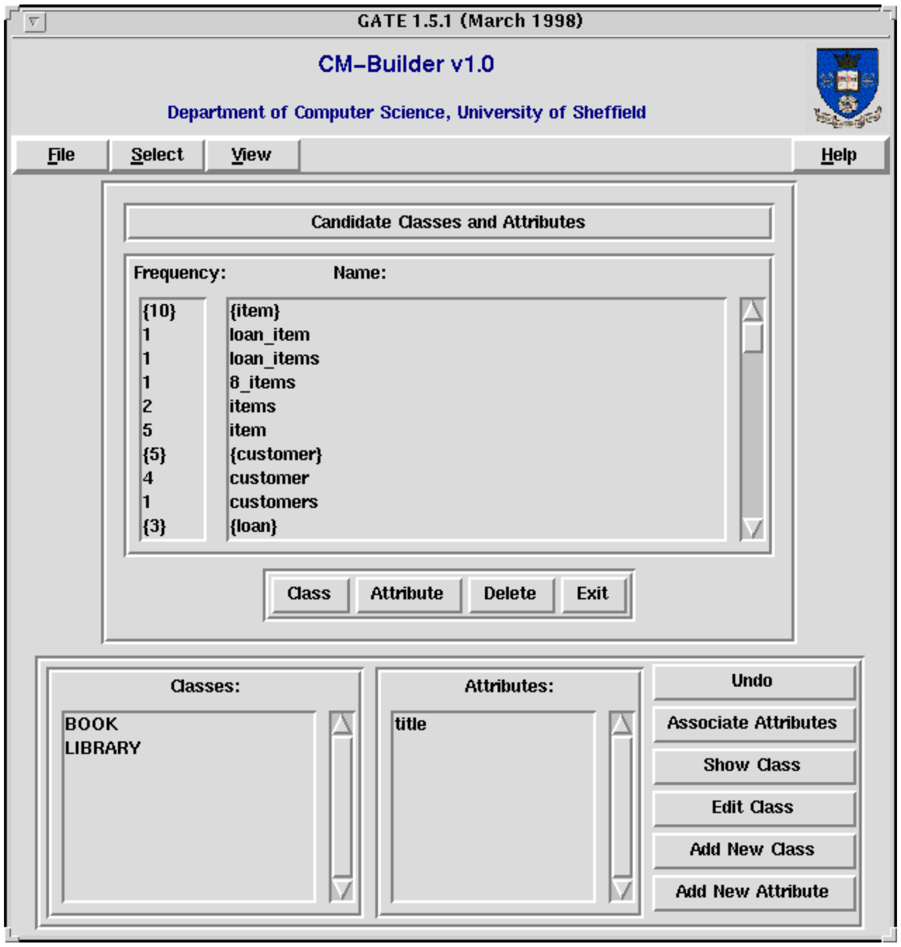
4.2. CM-Builder 1

The main goal of CM-Builder 1 is to carry out surface analysis as discussed in section 3.1, i.e., to generate lists of candidate classes, attributes, and relationships and help the user in filtering these lists. An interactive graphical interface, built in TCL/TK (Ousterhout, 1994), is provided to help the system users in filtering the candidate lists, and in associating candidate attributes, classes and relations with each other (see figures 4 and 5).

CM-Builder 1 (see figure 2) consists of eight modules in a pipelined architecture in the following order: a tokeniser, sentence splitter, part-of-speech tagger, morphological analyser, parser, verb phrase filter, noun phrase frequency analyser, and an OOA workbench. The first five modules are part of the core NLP system described in the preceding section. The verb phrase filter is used to produce a list candidate relations and the noun phrase frequency analyser produces a list of candidate classes. OOA Workbench is the interactive module through which the analyst works to build the final class model.

4.2.1. *The NP Frequency Analysis Module*

This module takes the syntactic trees produced by the parser and generates a list of candidate classes and attributes from the base noun phrases found in the text. For each base noun phrase found by the parser, the head of the noun phrase is assumed to be the rightmost noun and the root form of this word becomes a candidate class name. To each candidate class name, this module attaches its frequency in the requirements document. Synonymous words (e.g. customer, client) are not dealt with here, so they give rise to different candidate classes. This problem could be solved by using an external resource such as the WordNet lexical database (Fellbaum, 1998). Also we have not dealt with the problem of word sense disambiguation because in the short texts we have processed we have not encountered this problem, so we rely on a “one sense per discourse” assumption (Gale et al., 1992).



Frequency:	Name:
{10}	{item}
1	loan_item
1	loan_items
1	8_items
2	items
5	item
{5}	{customer}
4	customer
1	customers
{3}	{loan}

Classes:
BOOK
LIBRARY

Attributes:
title

Figure 4. Identifying Classes and Attributes in CM-Builder 1

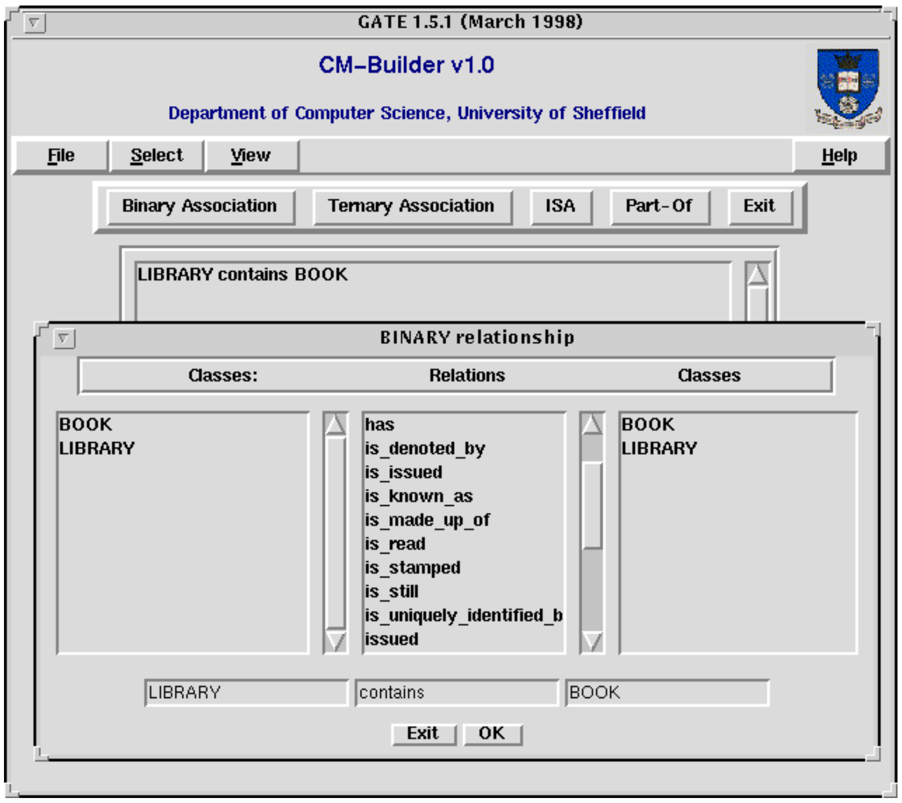
4.2.2. *The VP Filter Module*

This module takes the syntactic trees produced by the parser and extracts all the core verb phrases (i.e. the verb groups). These verb phrases are used as candidate associations.

4.2.3. *The OOA Workbench Module*

This module provides an interactive GUI for the system. With this GUI, users of the system can choose classes, their attributes, and the relationships between them from the lists of candidate classes and candidate relationships. Also, the system allows for new classes, attributes and relationships to be added and existing ones to be edited or deleted.

The process of building the model starts by selecting the classes and their attributes from the list of candidate classes and attributes. Figure



LIBRARY contains BOOK

Classes:	Relations	Classes
BOOK	has	BOOK
LIBRARY	is_denoted_by	LIBRARY
	is_issued	
	is_known_as	
	is_made_up_of	
	is_read	
	is_stamped	
	is_still	
	is_uniquely_identified_b	
	issued	
LIBRARY	contains	BOOK

Figure 5. Building Relationships in CM-Builder 1

4 shows a screen-shot of the system with a list of candidate classes displayed. Candidate classes are enclosed in curly brackets to suggest a high possibility of being a relevant class (e.g. `item` in figure) and the list is given in descending order of frequency of references in the text to the candidate class. Following the proposed class immediately, the list displays, as evidence, the actual phrases in the text in which this head noun appeared. These phrases are also given with their individual frequencies. Any candidate class can be chosen for inclusion in the conceptual model by highlighting it and clicking the button labeled **Class**. This will make the chosen class appear in the class list – bottom left in figure 4. Similarly, if any candidate is selected as an **Attribute** – it is moved to the list of attributes. Attributes then may be associated with classes by pressing the **Associate Attributes** button, and may these associations may subsequently be viewed and edited.

In a similar way choosing relationships is a point and click operation. The analyst is provided with a list of classes and a list of candidate relations from which he can select any of the four main relationships (i.e.

binary/ternary associations, generalisation, or aggregation relations). Figure 5 shows a screen-shot of the user interface for building relationships. By clicking any of the four buttons (**Binary Association, Ternary Association, ISA,** or **Part_Of**) in the main window a small window pops up with lists of classes and candidate relationships. The small window in figure 5 is for specifying binary associations. It shows two lists of classes separated by a list of candidate relations. When any item in these lists is selected with the mouse it appears immediately as a text entry below the list. The relationship name can be changed as required to improve readability. When the **OK** button is clicked the relationship is displayed in the main window and a second relationship can be chosen in the same way.

At any time during analysis new classes, attributes, and relationships can be added. After the analyst is satisfied with the model he can generate a CDIF file and then save the model.

Note that in CM-Builder 1 *all* associations between attributes and classes and between classes and other classes via relationships are specified manually by the user. The tool is helping simply by drawing attention to the most likely classes and the words most suggestive of relationships between classes.

4.3. CM-Builder 2

This version of the system performs domain independent OO analysis as described in section 3.3. It takes a single software requirements document as an input and linguistically analyses this document to build an integrated discourse model. This discourse model is passed to an OO analysis module which extracts the main object classes and the static relationships among objects of these classes. The system produces three kinds of output: a list of candidate classes; a list of candidate relationships; and a conceptual model represented in the standard CDIF format mentioned above. The general architecture of this system is shown above in Figure 2. It should be noted that in contrast to CM-Builder 1 there is no user interaction with CM-Builder 2: a first-cut conceptual model is produced entirely automatically from a NL software requirements text. Of course it is likely that this model will contain errors or omissions, so the intent is that the CDIF output file will be input to a CASE tool which will be used to further refine and extend the model.

4.3.1. *The OOA Module*

The processing steps of this module can be summarised as follows.

1. All object classes newly added to the semantic net (i.e. all nouns in the text) are taken as candidate classes.

2. All event classes newly added to semantic net (i.e all non-copular verbs in the text) are taken as candidate relationships.

3. For every candidate class find its frequency in the text (i.e. how many times it has been mentioned). The most frequent candidates highly suggest classes.

4. Attributes can be found using some simple heuristics like the possessive relationships and use of the verbs *to have, denote,* and *identify.*

5. Attributive adjectives denote attribute values of the nouns they modify. These are interesting language elements that give more information about the entities denoted by nouns. For example, in a sentence like "*a large library has many sections*", the adjective *large* indicates the existence of the attribute *size* associated with the entity *library.* WordNet, an external lexical database (Fellbaum, 1998), is used to find the appropriate attribute names from these adjectives.

6. For every candidate relationship find its complements (i.e. logical subject, logical object, and any attached prepositional phrases).

7. Any candidate relationship that has no complements is deleted from the list, since these arise from intransitive verbs (or errors in parsing). Intransitive verbs do not usefully signal relations.

8. Any candidate class that has a low frequency and does not participate in any relationship is discarded from the list. We have used a threshold of 2, which has proved useful for texts ranging between 100 and 300 words in size, but this threshold is a parameter which can be controlled by the user.

9. Some sentence patterns (e.g. '*something is made up of something*', '*something is part of something*' and '*something contains something*') denote aggregation relations.

10. Determiners are used to identify the multiplicity of roles in associations. Although this not a major goal of the research reported in this paper, our approach identifies three types of UML multiplicities (Booch et al., 1999):

- **1 for exactly one**: identified by the presence of indefinite articles, the definite article with a singular noun, and the determiner *one.*
- *** for many**: identified by the presence of any of the determiners *each*, *all*, *every*, *many*, and *some.*
- **N for specific numbers**: identified by the presence of numbers, such as one, two, seven, etc.

5. A Case Study

In this section we illustrate CM-Builder 2 using a case study from the domain of library information systems. This case study was originally presented in Callan (1994), pages 169-174. The problem statement for this case study is as follows:

A library issues loan items to customers. Each customer is known as a member and is issued a membership card that shows a unique member number. Along with the membership number, other details on a customer must be kept such as a name, address, and date of birth. The library is made up of a number of subject sections. Each section is denoted by a classification mark. A loan item is uniquely identified by a bar code. There are two types of loan items, language tapes, and books. A language tape has a title language (e.g. French), and level (e.g. beginner). A book has a title, and author(s). A customer may borrow up to a maximum of 8 items. An item can be borrowed, reserved or renewed to extend a current loan. When an item is issued the customer's membership number is scanned via a bar code reader or entered manually. If the membership is still valid and the number of items on loan less than 8, the book bar code is read, either via the bar code reader or entered manually. If the item can be issued (e.g. not reserved) the item is stamped and then issued. The library must support the facility for an item to be searched and for a daily update of records.

Section 5.1 presents Callan's analysis results, and section 5.2 presents the results produced automatically by the CM-Builder and their comparison with Callan's analysis.

5.1. Callan's Class Model

Figure 6 below shows a class diagram of the library system presented in Callan (1994). This model shows 8 classes drawn as solid rectangles. These classes are linked to each other with associations represented by lines between the class boxes. **Library** has been modeled as an aggregate of a number of **Section**s and this is represented by the diamond at the **Library** end of the association. Each section is uniquely

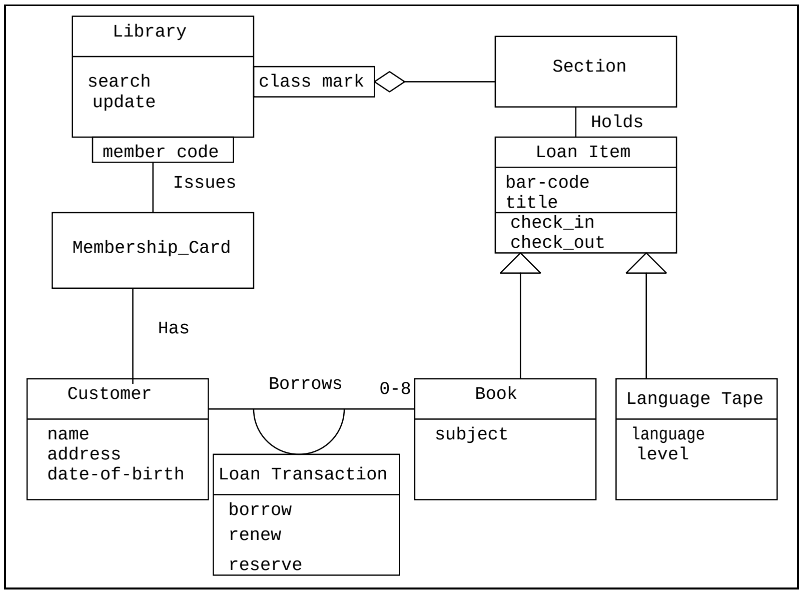
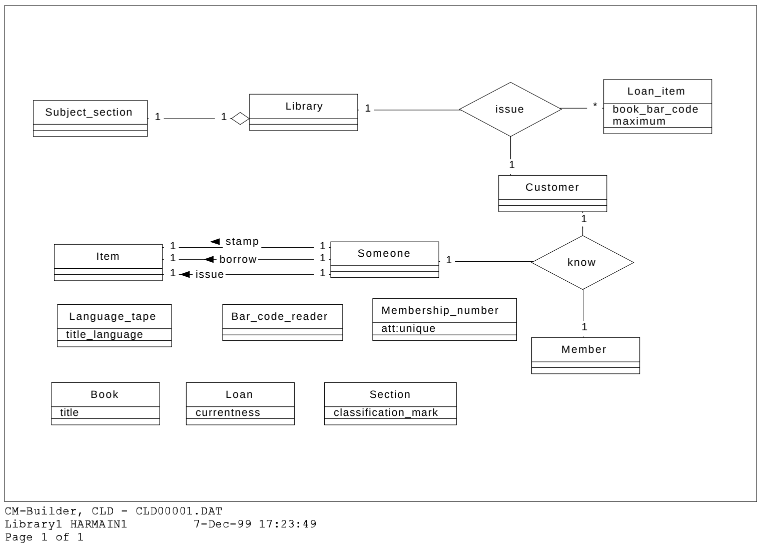


Figure 6. A Class Model of the Library System from Callan (1994), p.171

identified by a *class mark*, this is represented by a small box showing the *class mark* attribute at the **Library** end of the association. Also each section is associated with **Loan Item**s. Two operations are shown in the library class: *search* and *update*, these are shown in the third compartment of the **Library** class icon. There is an *issues* association between the **Library** and **Member Card** classes. This association is qualified with a *member code* attribute, which means every **Member Card** has a unique *member code*. The class **Customer** is associated with the class **Member Card** to show that each customer has a card. **Customer** is also associated with the class **Loan Item** via a *Borrows* association which is represented as an association class. Each **Customer** can borrow up to 8 items. This is shown by the multiplicity **0-8** at the **Loan Item** end. The class **Loan Item** has two subclasses **Language Tape** and **Book**.

5.2. CM-Builder Analysis of the Library System

From the above problem statement in section 5 CM-Builder produces 72 candidate classes and attributes, and 18 candidate relationships. The phrase frequency analysis process reduces the number of candidates to 34 candidates. The compound noun and attribute analysis process, as discussed above, further reduces the total number of candidate classes to 28. The candidate relationships are also filtered by discarding any



```
CM-Builder, CLD - CLD00001.DAT
Library1 HARMAIN1        7-Dec-99 17:23:49
Page 1 of 1
```

Figure 7. A Class Model of the Library System produced by CM-Builder

candidate that does not associate at least two classes. This process reduces the total number of candidate relationships to 8. Given this list of candidate relations, the list of candidate classes is further refined based on their frequency and their participation in relationships. Isolated candidate classes (i.e. do not participate in relations) with low frequency (less than 2) are deleted.

The final model produced by the CM-Builder consists of 11 classes and 8 associations as shown in figure 7. Six out of the 13 classes in figure 7 were exactly as shown in Callan's model. These classes are: **Book, Library, Customer, Loan_item, Section,** and **Language_tape**. One class, **Membership_number**, is incorrect because it represents an attribute. One class, **Member_card**, was discarded because it has low frequency. Six classes: **Bar_code_reader, Item, Member, Loan, Subject_section** and **Someone** are not shown in Callan's model. Three of these classes are synonymous with other classes but our system could not resolve them. These classes and their synonymous doubles are: (**Item, Loan_Item**), (**Member, Customer**), and (**Section, Subject_section**). The class named **Someone** is included by CM-Builder to indicate a missing agent. The other two classes were considered to be irrelevant by Callan.

Compared to the model produced by Callan the model our system produces is over-specified. Over-specification at the early analysis stages is considered to be an advantage because the more classes you can identify the more chance you have to select the right ones (Larman, 1998; Yourdon and Argila, 1996).

6. Evaluation

Following Hirschman and Thompson (1996) we can distinguish three broadly different sorts of software evaluation. *Adequacy evaluation* is the determination of system fitness for some particular task in context. *Diagnostic evaluation* is used by system developers to test their system during its development. *Performance evaluation* is a measurement of system performance in some area of interest.

Here we focus on performance evaluation since we are interested in evaluating a generic technology, not its fitness for a specific task in a real setting or the correctness of its implementation. Specifying a methodology for performance evaluation requires specifying three things (Hirschman and Thompson, 1996):

- **Criterion:** what we are interested in evaluating (e.g. precision, speed, error rate)
- **Measure:** what property of system performance we report to get at the chosen criterion (e.g. ratio of hits to hits and misses, seconds to process, percent incorrect)
- **Method:** how we determine the appropriate value for a given measure and a given system.

In the next section we define our evaluation methodology and then in the following section show how this methodology has been used to carry out a preliminary evaluation of CM-Builder 2. We address only the evaluation of CM-Builder 2 for various reasons. First, regardless of how conceptual models are produced – fully manually, semi-automatically, or fully automatically – quantitative evaluation of the resultant model presupposes a methodology for quantitatively assessing models, and it is this difficult, key conceptual issue that we begin to address below. In addressing it, it matters little how the models to be evaluated have been generated. Second, since CM-Builder 1 is an interactive system, it can only properly be evaluated together with a human in the loop, and this makes for difficult experimental conditions which we have not had the resources to set up. One would like to know, for example, whether an analyst working with CM-Builder 1 could produce conceptual models more accurately or faster than one working only with the requirements document. Of course, one could, and should, evaluate the purely automatic components of CM-Builder 1, i.e., the noun phrase and verb group identification; but this is now a relatively standard language engineering task, known to be achievable to certain levels, and not worth pursuing here. Of more interest is evaluating, for example, how many of CM-Builder 1's proposed candidate classes are actually appropriate classes for inclusion in the conceptual model. However, this capability is effectively evaluated out in evaluating CM-Builder 2. Finally, unlike CM-Builder 1, CM-Builder 2 does produce conceptual models automatically, and it cries out for some form of evaluation in order to assess whether the approach is worth pursing.

We view the methodology proposed below as one of the significant contributions of our work, as to the best of our knowledge no such methodology for evaluating NL-based CASE tools has been advanced to date, nor have existing systems (cf. section 2) been formally evaluated.

6.1. Evaluation Methodology

6.1.1. *Criterion*

The criterion applied to evaluate CM-Builder is how close are the models produced by the system (called *system responses*) to those produced by human analysts (called *answer keys*). However, a single gold standard model for any given software requirement does not exist, as different human analysts will usually produce different models. These models cannot be categorised as strictly *correct* or *incorrect*, but nonetheless they are usually categorised as *good* or *bad*, depending on the objects and the relationships represented in them. We have assumed that the models available in Object-Oriented text books are good models and have used them as answer keys.

6.1.2. *Measure*

We have used three metrics for evaluating our system. Two of these, known as *recall* and *precision*, were originally developed for evaluating Information Retrieval systems (van Rijsbergen, 1979) and now are being widely used in evaluating Information Extraction systems (Grishman and Sundheim, 1996). The third is a new metric we have defined.

- **Recall:** Recall reflects the completeness of the results produced by the system. The correct and relevant information returned by the system is compared with that found by human analysts. The following formula is used to calculate recall:

$$Recall = \frac{N_{correct}}{N_{key}}$$

where $N_{correct}$ refers to the number of correct responses made by the system, and N_{key} is the number of information elements in the answer key.

– **Precision:** Precision reflects the accuracy of the system (i.e. how much of the information produced by the system was correct). The following formula is used to calculate precision:

$$Precision = \frac{N_{correct}}{N_{correct} + N_{incorrect}}$$

where $N_{correct}$ and N_{key} are as above, and $N_{incorrect}$ refers to the incorrect responses made by the system.

– **Over-specification:** Over-specification measures how much extra correct information in the system response is not found in the answer key (i.e. attempts to separate the system's language processing capability from its abstraction capability). It is given by the formula:

$$\textit{Over-specification} = \frac{N_{extra}}{N_{key}}$$

where N_{extra} is the number of responses judged correct but not found in the key and N_{key} is as above.

6.1.3. *Method*

Each class model element (class, attribute, association, generalization, aggregation) in the response is compared with each element in the key and classified as follows: *correct* if it matches an element in the answer key; *incorrect* if it does not match an element in the key; *extra* if it is valid information from the text but is not in the key. An element occurring in the key but not in the response is classified as *missing.*

Determining a match is done in two stages. First the names of the matching elements must be plausibly close (e.g. *Item* can match *Loan Item* but not *Library*). Second, an element-specific process is carried out to see if the 'context' of the element gives evidence that it really is what its name suggests. So, a class with a name matching one in the key, but each of whose attributes or associations differs from those in the key, cannot be said to match the key class. Similarly attributes and associations are matched both by approximate name matching and by context – the (at least partial) correctness of the classes and other attributes with which they are associated. This means that in general the evaluation of one element type is not independent of that of other element types.

To proceduralise this matching we follow an approach similar to that used for template matching in the MUC evaluations (Chinchor, 1995) in which starts by uniquely aligning the classes proposed in the response with those in the key so as to optimise the overall class model score for the system. This is a complex, non-deterministic process whose outcome will differ depending on, e.g., whether one wants to maximise the absolute number of class model elements matched between response and key or whether one weights certain elements above others (e.g. associations above attributes) and on the level required for element contexts to match.

Because the complexity of this class model matching process demands automated support which is not yet available, we have so far limited ourselves to an evaluation in which only classes are evaluated. This is much simpler than an evaluation of all model elements because in cases where either of two response classes could be matched to one key class, or vice versa, the score will not be affected, regardless of choice. To determine attribute and association scores, however, this is no longer the case (consider e.g. **Loan_Item** and **Item** in Figure 7). Despite the simplification, class matching between key and response is a very important initial evaluation of class models, since classes are the fundamental class model element. In carrying out class matching we have required that the response and key class names approximately match and that the attributes and associations in the response class are not on balance more wrong than right.

6.2. Evaluation Results

A corpus of five case studies, each from a different domain and extracted from a text book, was used for the final evaluation of CM-Builder. None of these case studies was examined in detail prior to the final evaluation, nor was the system run on any of them before the evaluation. These case studies range between 100 and 300 words in size, with sentence length ranging between 6 and 31 words. The average sentence length was 17 words.

Table 6.2 reports the scores of the system on the five case studies. Scores for each case study are given in one row and the last row shows the overall scores of the system. The overall performance of the system was 73% recall and 66% precision, with an over-specification of 62%.

Since formal evaluation of other NL-based CASE tools has not been carried out, we cannot compare our results with them. However, we can note that other language processing technologies, such as information retrieval systems, information extraction systems, and machine translation systems, have found commercial applications with precision

Table I. Evaluation results

Case Study	N_{cor}	N_{inc}	N_{mis}	N_{ext}	REC %	PRE %	OVS %
1	4	3	6	7	40	57	70
2	6	2	2	5	75	75	63
3	5	3	0	3	100	63	60
4	8	5	1	4	89	62	44
5	4	1	1	4	80	80	80
Overall	27	14	10	23	73	66	62

and recall figure well below this level. Thus, the results of this initial performance evaluation are very encouraging and support both the approach adopted and in this paper and the potential of this technology in general.

7. Conclusions and Future Work

In this paper we have described an automated NL-based CASE tool, the CM-Builder. This tool uses Natural Language Processing techniques to analyse natural language software requirements documents and produce initial conceptual models represented in the Unified Modeling Language (UML). The overall aim is to quickly and cheaply produces first-cut conceptual models which can reduce the cost and time of software development.

CM-Builder can be used two ways. It can be used interactively, in conjunction with a human analyst, to carry out "surface analysis" of the text to propose candidate classes, attributes, and relationships which the user then explicitly relates to each other and includes in the conceptual model (CM-Builder 1). Or, it can work fully automatically, carrying out "domain independent semantic analysis" to build an integrated discourse model of the requirements text (CM-Builder 2). This discourse model is then used for the identification of object classes, their attributes, and the static relationships among them which are proposed as part of the conceptual model. In both versions of CM-Builder the final conceptual model is represented in UML and is given in a standard data interchange format, CDIF. This final model can be further refined by a human analyst using any graphical CASE tool that supports CDIF.

We have also defined a quantitative evaluation methodology and have used this methodology to evaluate the class identification capability of CM-Builder. The results achieved have, we believe, shown the benefits of our approach for Object-Oriented Analysis, though clearly more experimentation is needed. More importantly, the methodology provides a starting point for benchmarking systems that produce OO conceptual models, enabling them to be compared against human analysts, against each other and against earlier versions of themselves.

There is much scope for extension and improvement of the work presented in this paper:

- The small corpus of case studies we used has proved very useful for system development. Building and analysing a larger corpus, in particular one containing 'real world' requirements texts, would provide a valuable source of knowledge for improving our approach, and would be a useful resource for the larger community.

- While the coverage of our parser has proved adequate, it can be improved by encoding more grammar rules for the sentence types most frequently used in software requirements texts.

- The discourse interpretation module can be improved by encoding further general knowledge relevant to analysis and design (while remaining domain independent). However, the identification of this kind of knowledge is a big research topic in itself.

- Our procedure focuses on the extraction of Class Model elements (i.e. classes, attributes, and associations) which show the static aspect of the modeled system. The same thing could be done to address the dynamic aspect of the system.

- We have shown how NL-based CASE tools can be quantitatively evaluated. However, our evaluation is partial and further work is needed for full evaluation, particularly with respect to attributes and relationships.

Addressing these points will lead to improvement in the NL-based conceptual model building technology. Finally, however, to make this technology truly usable it needs to be fully integrated into a powerful graphical CASE tool with a user interface carefully designed to support software analysts and informed by studies of real analysts' design behaviour when working with tools like CM-Builder. A user-based study that observed three analysts working with the same requirements document and same graphical CASE tool, one with no additional NL tool, one with CM-Builder 1, and one with CM-Builder 2 would be highly revealing in this regard.

References

Abbott, R. J.: 1983, 'Program Design by Informal English Descriptions'. *Communications of the ACM* **26**(11), 882–894.

Belkhouche, B. and J. Kozma: 1993, 'Semantic case analysis of information requirements'. In: S. Brinkkemper and F. Harmsen (eds.): *NGCT'93, 4th Workshop on the Next Generation of CASE Tools. Memoranda Informatica 93-32.* pp. 163–182, University of Twente.

Booch, G.: 1986, 'Object-Oriented Development'. *Trans. on Software Eng.* **SE-12**(2), 211–221.

Booch, G.: 1994, *Object-Oriented Analysis And Design With Applications.* The Benjamin/Cummings Publishing Company, Inc., second edition.

Booch, G., J. Rumbaugh, and I. Jacobson: 1999, *Unified Modeling Language User Guide.* Addison-Wesley.

Brill, E.: 1994, 'Some Advances in Transformation-Based Part of Speech Tagging'. In: *Proceedings of the Twelfth National Conference on AI (AAAI-94).* Seattle, Washington.

Callan, R. E.: 1994, *Building Object-Oriented Systems: An introduction from concepts to implementation in C++.* Computational Mechanics Publications.

Cardie, C. and D. Pierce: 1998, 'Error-Driven Pruning of Treebank Grammars for Base Noun Phrase Identification'. In: *Proceedings of the COLING-ACL'98 Joint Conference (the 17th International Conference on Computational Linguistics and 36th Annual Meeting of the Association for Computational Linguistics).* Montreal, pp. 218–224.

Chinchor, N.: 1995, 'Four Scorers and Seven Years Ago: The Scoring Method for MUC-6'. In: *Proceedings of the Sixth Message Understanding Conference (MUC-6).* pp. 33–38, Morgan Kaufmann.

Cowie, J. and W. Lehnert: 1996, 'Information Extraction'. *Communications of ACM* **39**(1).

Cunningham, H., Y. Wilks, and R. Gaizauskas: 1996, 'Software Infrastructure for Language Engineering'. In: *Proceedings of the AISB Workshop on Language Engineering for Document Analysis and Recognition.* Brighton, U.K.

Cyre, W.: 1995, 'A requirements sublangauge for automatic analysis'. *International Journal of Intelligent systems* **10**(1), 665–689.

Dunn, L. and M. Orlowska: 1990, 'A natural language interpreter for construction of conceptual schemas'. In: *CAiSE'90, 2nd Nordic Conference on Advanced Information Systems Engineering, LNCS 436.* pp. 371–386, Springer-Verlag.

Ernst, J. (ed.): 1996, *CDIF -Integrated Meta-Model- Object-Oriented Analysis and Design Core Subject Area.* Electronic Industries Association.

Fellbaum, C. (ed.): 1998, *WORDNET: An Electronic Lexical Database.* Cambridge, Massachusetts, London, England: The MIT Press.

Fuchs, N., U. Schwertel, and R. Schwitter: 1998, 'Attempto Controlled English - Not Just Another Logic Specification Language'. In: P. Flener (ed.): *Logic-Based Program Synthesis and Transformation, Workshops in Computing, Eight International Workshop LOPTSR'98*, Lecture Notes in Computer Science 1559. Springer-Verlag, pp. 1–20.

Gaizauskas, R.: 1995, 'XI: A Knowledge Representation Language Based on Cross-Classification and Inheritance'. Technical Report CS-95-24, Department of Computer Science, University of Sheffield.

Gaizauskas, R.: 1997, 'Evaluation in Language and Speech Technology'. *Computer Speech and Language* **12**(4).

Gaizauskas, R., H. Cunningham, Y. Wilks, P. Rodgers, and K. Humphreys: 1996, 'GATE – an Environment to Support Research and Development in Natural Language Engineering'. In: *Proceedings of the 8th IEEE International Conference on Tools with Artificial Intelligence (ICTAI-96).* Toulouse, France.

Gaizauskas, R. and K. Humphreys: 1997, 'Using a Semantic Network for Information Extraction'. *Natural Language Engineering* **3**(2/3), 147–169.

Gaizauskas, R. and K. Humphreys: 2000, 'Quantitative Evaluation of Coreference Algorithms in an Information Extraction System'. In: S. Botley and T. McEnery (eds.): *Discourse Anaphora and Anaphor Resolution.* London: John Benjamins.

Gaizauskas, R. and Y. Wilks: 1998, 'Information Extraction: Beyond Document Retrieval'. *Journal of Documentation* **54**(1), 70–105.

Gale, W. A., K. W. Church, and D. Yarowsky: 1992, 'One Sense Per Discourse'. In: *The Proceedings of the 4th DARPA Speech and Natural Language Workshop.*

Gazdar, G. and C. Mellish: 1989, *Natural Language Processing in Prolog: An Introduction to Computational Linguistics.* Addison-Wesley.

Gomez-Perez, A. and V. R. Benjamins: 1999, 'Overview of Knowledge Sharing and Reuse Components: Ontologies and Problem-Solving Methods'. In: *Proceedings of the IJCAI-99 workshop on Ontologies and Problem-Solving Methods (KRR5).* Stockholm, pp. 1.1–1.15. Available at: http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-18/.

Grishman, R. and B. Sundheim: 1996, 'Message Understanding Conference-6: A brief history.'. In: *Proceedings of the 16th International Conference on Computational Linguistics.* Copenhagen.

Harmain, H. M.: 2000, 'Building Object-Oriented Conceptual Models using Natural Language Processing Techniques'. Ph.D. thesis, Department of Computer Science, Uni. of Sheffield.

Hirschman, L.: 1998, 'The Evolution of Evaluation: Lessons from the Message Understanding Conferences'. *Computer Speech and Language* **12**(4), 281–305.

Hirschman, L. and H. S. Thompson: 1996, 'Chapter 13 Evaluation: Overview of evaluation in Speech and Natural Language Processing'. In: R. A. Cole, J. Mariani, H. Uszkoreit, A. Zaenen, and V. Zue (eds.): *Survey of the State of the Art in Human Language Technology.* Center for Spoken Language Understanding, Oregon Graduate Institute. Available at: http://cslu.cse.ogi.edu/HLTsurvey/HLTsurvey.html.

Humphreys, K., R. Gaizauskas, , S. Azzam, C. Huyck, B. Mitchell, and Y. Wilks: 1998, 'Description of the LaSIE-II system as used for MUC-7.'. In: *Proceedings of the Seventh Message Understanding Conference (MUC-7).*

Larman, C. (ed.): 1998, *Applying UML and Patterns: An interoduction to Object-Oriented Analysis and Design.* Prentice- Hall PTR.

Macias, B. and S. Pulman: 1995, 'A Method for Controlling the Production of Specifications in Natural Language'. *The Computer Journal* **38**(4).

Meziane, F.: 1994, 'From English to Formal Specifications'. Ph.D. thesis, Department of Mathematics and computer Science, University of Salford. UK.

Mich, L.: 1996, 'NL-OOPS: from natural language to object oriented using the natural language processing system LOLITA'. *Natural language engineering* **2**(2), 161–187.

Mich, L. and R. Garigliano: 1994, 'A Linguistic Approach to the Development of Object-Oriented Systems Using the NL Sytem LOLITA'. In: *Object-Oriented Methodologies and Systems International symposium, (ISOOMS'94), LNCS 858.* pp. 371–386.

Ousterhout, J. K.: 1994, *Tcl and the Tk Toolkit.* Addison-Wesley.

Pereira, F. and S. Shieber: 1987, *Prolog and Natural-Language Analysis*, No. 10 in CLSI Lecture Notes. Stanford, CA: Stanford University.

Saeki, M., H. Horai, K. Toyama, N. Uematsu, and H. Enomoto: 1987, 'Specification Framework Based on Natural Language'. In: *Proceedings of the 4th international workshop on software specification and design*. pp. 87–94, IEEE.

van Rijsbergen, C.: 1979, *Information Retrieval*. London: Butterworths.

Voorhees, E. and D. Harman: 1999, 'Overview of the Eighth Text REtrieval Conference (TREC-8'. In: *Proceedings of the Eighth Text Retrieval Conference (TREC-8)*. NIST Special Publication 500-246. Available at: http://trec.nist.gov/pubs/trec8/t8_proceedings.html.

Yourdon, E. and C. Argila: 1996, *Case-Studies in Object-Oriented Analysis and Design*. Yourdon press.