CM-Builder: An Automated NL-based CASE Tool

H.M. Harmain
Dept. of Computer Science
University of Sebha, Libya
hmharmain@dcs.shef.ac.uk

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

Abstract

This paper describes a natural language-based CASE tool called CM-Builder which aims at supporting 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 build an integrated discourse model of the processed text, represented in a Semantic Network. This Semantic Network is then used to automatically construct 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 [4]. 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 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 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 automated CASE tool which analyses software requirements written in English and produces 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 our system.

The rest of this paper is structured as follows: section 2 provides background information and describes related work; section 3 describes in detail our fully implemented NL-based CASE tool; section 4 shows the results of using the system to analyse a case study; section 5 discusses the evaluation methodology; section 6 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 Artificial Intelligence (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 [18].

The second approach 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 existing work.

Abbott [1] proposed 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 [3]. Booch described 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" [1].

Saeki et. al. [28] described 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 were automatically extracted from the informal requirements but the system could 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 [10] described a natural language interpreter for the construction of NIAM (Nijssen's Information Analysis Method [1]) 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 [24] 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 [26, 25] described 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 [13], ASPIN, an Automatic Specifications Interpreter [9], and the Requirements Analysis Support System described in [2] are examples of systems that analyse subset of a natural language (English). Macias and Pulman [23] 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 require an unacceptable 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 (see, e.g. [19]).

Drawing on this work, 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 automat-

[1] Also know as Natural-language Information Analysis Method

ically processes a requirements documents and 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.

3. The CM-Builder

The Class Model Builder (CM-Builder) is a modular NL-based CASE tool which performs domain independent OO analysis. It takes a single software requirements document as an input, linguistically analyses this document to build an integrated discourse model and extracts the main object classes and the static relationships among objects of these classes. The system produces three kinds of outputs: a list of candidate classes; a list of candidate relationships; and a conceptual model represented in a standard interchange format, the CASE Data Interchange Format (CDIF) [11], which can be directly input to a graphical CASE tool such as Select Enterprise Modeler for further refinement by a human analyst.

Figure 1 illustrates the main steps of the CM-Builder approach. These steps can be summarised as follows.

1. Obtain a set of functional requirements or a 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.

3.1. The Input Document

CM-Builder takes a plain text file containing the software requirements. We impose no restrictions on the form of the requirements document provided that it is written in English. It can be in the form of a general problem statement describing the software problem or a list of more detailed functional requirements.

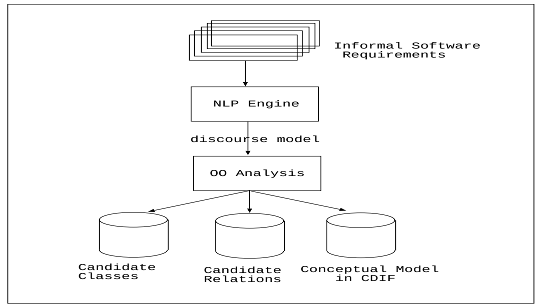

Figure 1. The CM-Builder Approach

3.2. The Core NLP System

The core of CM-Builder is a set of NLP modules organised in a pipelined architecture. These modules were originally built for the LaSIE (Large Scale Information Extraction) system at Sheffield [15, 21]. 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. The core system processes a text in three main processing stages:

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, to be used by the parser. The processing steps are carried out in the following order:

   (a) Tokenization: The tokenizer splits a plain text file into tokens. This includes, e.g., separating words and punctuation, identifying numbers, and so on.

   (b) Sentence Splitting: The sentence splitter identifies sentence boundaries

   (c) 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 [6].

   (d) 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.

2. *Parsing and Semantic Interpretation* We have used a bottom-up chart parser implemented in Prolog, an enhanced version of the parser described in [16]. 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 [27]). 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: library(e2), determiner(e2,a), number(e2,sing), issue(e1), time(e1,present), aspect(e1,simple), voice(e1,active), lsubj(e1,e2), loan(e4), item(e3), qual(e3,e4), number(e3,plural), lobj(e1,e3).

   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).

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 [14].

   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 2. 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.

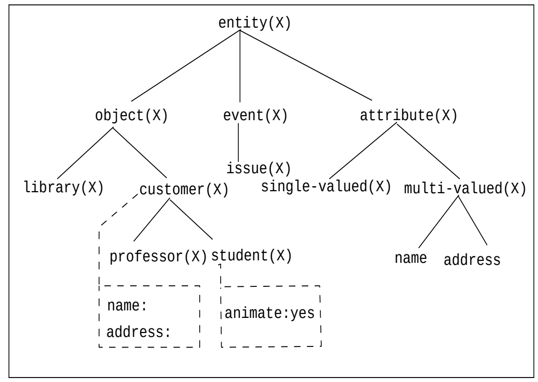

Figure 2. A simple world model

The background world model may be either very general, in which case it provides weak support for text understanding, but is very portable across domains, or it may be specialised for a particular domain, in which case it provides strong support for text understanding, but only in the domain for which it has been specialised. Because we want CM-Builder to be a general purpose tool we have used only a very general world model; however, domain specific models can be constructed if there is a need to analyse many requirements documents in the same domain.

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:

(a) 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.

(b) 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.

(c) 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 [21].

(d) 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 takes the meaning representation of the next sentence. All sentences go through the same stages.

3.3. The OOA Module

This module generates lists of candidate classes and candidate relationships from the discourse model produced by the NLP component. A domain-independent algorithm is then used for extracting the class model elements (i.e. classes, attributes, and relationships). 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 [12], 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 [5]:
    - **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.

The result of the above steps is an initial class model. This model consists of classes, their attributes, relationships between classes, and multiplicities on relationships.

3.4. The CDIF Output File

A CDIF file is produced as a final output of the system. This 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[2] modeler V5.0 to test our system.

4. A Case Study

Here we discuss a case study from the domain of library information systems. This case study was originally presented by Callan [7], pages 169-174. The problem statement for this case study is as follows:

[2]SELECT Enterprise is a trademark for SELECT Software Tools.

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 4.1 presents Callan's analysis results, and section 4.2 presents the results produced automatically by the CM-Builder and their comparison with Callan's analysis.

4.1. Callan's Class Model

Figure 3 shows a class diagram of the library system produced by Callan [7]. 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 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**.

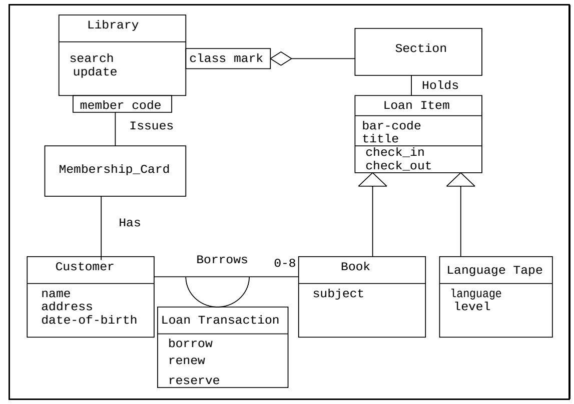

Figure 3. A Class Model of the Library System from [7], p.171

4.2. CM-Builder Analysis of the Library System

From the above problem statement CM-Builder produces 72 candidate classes and attributes, and 18 candidate relationships. The phrase frequency analysis process reduces the number of candidates to 34. 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 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. those that do not participate in relations) with low frequency (less than 2) are deleted.

The final model produced by the CM-Builder consists of 13 classes and 6 associations as shown in figure 4. Six out of the 13 classes are 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 an 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 [22, 30].

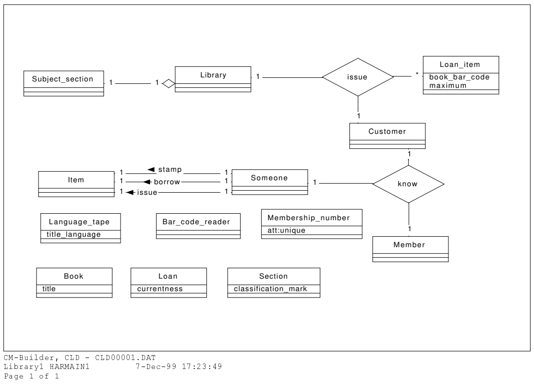

Figure 4. A Class Model of the Library System produced by CM-Builder

5. Evaluation

Following Hirschman and Thompson [20] we can distinguish three broadly different sorts of software evaluation. *Adequacy evaluation* is the determination of system fitness for some particular task. *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 or the correctness of its implementation. Specifying a methodology for performance evaluation requires specifying three things [20]:

- **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 the CM-Builder. We view this methodology 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.

5.1. Evaluation Methodology

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.

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 [29] and now are being widely used in evaluating Information Extraction systems [17]. 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.

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.

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 [8] 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 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 4). 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.

5.2. 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 1 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%.

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

Table 1. Evaluation results

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 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.

6. 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 software requirements and builds an integrated discourse model of the requirements. This model is then used for the identification of object classes, their attributes, and the static relationships among them. The final 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. An important aspect of CM-Builder is that it quickly and cheaply produces first-cut class models which can reduce the cost and time of software development. We have also defined an evaluation methodology and have used this methodology to evaluate the class identification capability of CM-Builder. The results achieved have shown the benefits of our approach for Object-Oriented Analysis.

There is much scope for further work.

- 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, will provide a valuable source of knowledge for improving our approach.
- 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 independ-

ent), though the identification of this kind of knowledge is a 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 associations.

Addressing these points will lead to improvement in the NL-based class 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.

References

[1] R. J. Abbott. Program design by informal english descriptions. *Comm. of the ACM*, 26(11):882–894, Nov 1983.

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

[3] G. Booch. Object-oriented development. *Trans. on Software Eng.*, SE-12(2):211–221, 1986.

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

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

[6] E. Brill. Some advances in transformation-based part of speech tagging. In *Proceedings of the Twelfth National Conference on AI (AAAI-94)*, Seattle, Washington, 1994.

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

[8] N. Chinchor. Four scorers and seven years ago: The scoring method for MUC-6. In *Proc. of the Sixth Message Understanding Conference (MUC-6)*, Morgan Kaufmann, 1995.

[9] W. Cyre. A requirements sublangauge for automatic analysis. *Int'l J. of Intelligent systems*, 10(1):665–689, 1995.

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

[11] J. Ernst, editor. *CDIF -Integrated Meta-Model- Object-Oriented Analysis and Design Core Subject Area*. Electronic Industries Association, 1996.

[12] C. Fellbaum, ed. *WORDNET: An Electronic Lexical Database*. The MIT Press, Cambridge, MA, 1998.

[13] R. S. Fuchs, Norbert E. Attempto controlled english. In *CLAW'96, The First Int'l Workshop on Controlled Language Applications*, Katholieke Universiteit, Leuven, March 1996.

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

[15] R. Gaizauskas, T. Wakao, K. Humphreys, H. Cunningham, and Y. Wilks. Description of the LaSIE system as used for MUC-6. In *Proceedings of the Sixth Message Understanding Conference (MUC-6)*. Morgan Kaufmann, 1995.

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

[17] R. Grishman and B. Sundheim. Message understanding conference-6: A brief history. In *Proc. of the 16th Int'l Conf. on Computational Linguistics*, Copenhagen, 1996.

[18] H. M. Harmain. *Building Object-Oriented Conceptual Models using Natural Language Processing Techniques*. PhD Thesis, Dept. of Computer Science, Univ. of Sheffield, 2000.

[19] L. Hirschman. The evolution of evaluation: Lessons from the message understanding conferences. *Computer Speech and Language*, 12(4):281–305, 1998.

[20] L. Hirschman and H. S. Thompson. 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, editors, *Survey of the State of the Art in Human Language Technology*. 1995.

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

[22] C. Larman, editor. *Applying UML and Patterns: An interoduction to Object-Oriented Analysis and Design*. Prentice-Hall PTR, 1998.

[23] B. Macias and S. Pulman. A method for controlling the production of specifications in natural language. *The Computer Journal*, 38(4), 1995.

[24] F. Meziane. *From English to Formal Specifications*. PhD thesis, Department of Mathematics and computer Science, University of Salford. UK, 1994.

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

[26] L. Mich and R. Garigliano. A linguistic approach to the development of object-oriented systems using the NL sytem LOLITA. In *Object-Oriented Methodologies and Systems Int'l Symposium, (ISOOMS'94), LNCS 858*, 371–386, 1994.

[27] F. Pereira and S. Shieber. *Prolog and Natural-Language Analysis*. Number 10 in CLSI Lecture Notes. Stanford University, Stanford, CA, 1987.

[28] M. Saeki, H. Horai, K. Toyama, N. Uematsu, and H. Enomoto. Specification framework based on natural language. In *Proc. of the 4th Int'l Workshop on Software Specification and Design*, 87–94. IEEE, 1987.

[29] C. van Rijsbergen. *Information Retrieval*. Butterworths, London, 1979.

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