RC24712 (W0812-106) December 18, 2008
Computer Science

IBM Research Report

An Analysis Engine for Dependable Elicitation of Natural Language Use Case Description and Its Application to Industrial Use Cases

Avik Sinha, Amit Paradkar, Palani Kumanan, Branimir Boguraev
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

An Analysis Engine for Dependable Elicitation of Natural Language Use Case Description and Its Application to Industrial Use Cases[*]

Avik Sinha[†] Amit Paradkar, Palani Kumanan, Branimir Boguraev
I.B.M. Thomas Watson Research Center
19 Skyline Dr., Hawthorne, NY. 10532
{aviksinha, paradkar, palanik, bran}@us.ibm.com

Abstract

We present 1) a novel application of a pipeline of linguistic analysis engines for understanding contents of a natural language use case specification; and 2) results of the first of a kind large scale experiment of application of linguistic techniques to industrial use cases. In spite of the recent developments in formal specification, natural language remains the predominant mode for specifying requirements. Therefore, for dependable system development and for automation of software engineering activities, a robust and scalable natural language processing technique is required that can translate natural language textual requirements into computer based models. We present one such technique that can process natural language use case descriptions. The novelty of our technique lies in our choice of architecture, which enables us to configure and extend the analysis engine to multiple domains and languages. We also report the results of applying our prototype to 80 industrial and academic use case descriptions. The results of our experiment indicate that our approach is very promising.

Keywords:[Requirements, Natural language, Validation, Feedback, Automation]. **Word count**: 8500 (approx.). **Submission Category**: PDS.

1 Introduction

We present implementation of a novel and domain independent linguistic technique, which exploits the efficiency of shallow parsers and configurability of Unstructured Information Management Architecture(UIMA) [10, 7] for online analysis of natural language (NL) use case descriptions. We also present initial results of applying the technique to 80 industrial and academic textual use cases. To the best of our knowledge, this is the first report on effectiveness of applying linguistic analysis to large and diverse set of industrial use cases.

In spite of the multiple benefits that formal methods has to offer to a requirements analyst, their adoption has been slow in software engineering processes (except possibly in the fields of hardware design and safety critical control systems). This is primarily due to the fact that the adoption overhead (learning, tool support, configurability, etc.) still overshadows the economic gain. Also, with the onset of agile methods, the emphasis is on active and continuous participation of the customer in the development process. Use of formal methods for requirement elicitation introduce high entry barrier for customer participation.

Graphical use cases along with their textual specifications are frequently used to model functional requirements of software applications [6]. As such, use cases form basis for verification & validation activities such as consistency and completeness analysis [24] and test generation [17] (along with references therein). These analyses require extraction of a formal behavioral model from use case description. The manner in which the initial use cases are described varies widely. For example, Whittle and Jayaraman [27] require that the use cases be described using notations such as sequence or activity diagrams, Ortner and Schienmann [19] propose a restricted subset of natural language, or [16, 24] propose a multi-tiered representation combining restricted subset of natural language with a formal notation such as PetriNets [16] or predicate logic [24].

Industrial use cases are primarily authored by business analysts (BA), who do not like formal notations for the following reasons:

1. As previously mentioned, formal notation prevents customer involvement.

2. Most formal notations are geared towards expressing specific properties of a system (functional, performance, safety, security, etc.) BAs, on the other hand, may have to capture a variety of such requirements for the system. Consequently, even if the BAs are not scared of choosing a set of formal notations that are fit for their task, the task of maintaining consistency across various formal models is error prone and inefficient.

These and many similar problems create impediments for adoption of formal authoring of use cases. For dependable systems, an approach is thus needed that can ensure/assess dependability of the system requirements based on free-form NL text.

[*]Article cleared for submission to DSN 09 from IBM.
[†]contact author

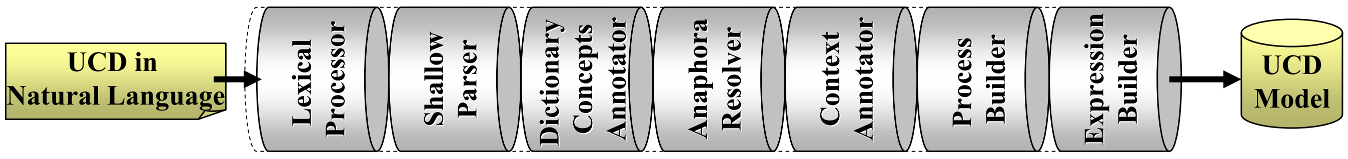


Figure 1: Overview of the analysis engine

1.1 How are we different?

As described in detail in section 2, our implementation is a pipeline of configurable analysis engines using a general purpose shallow parser (see Figure 1).

Approaches for analyzing textual requirements and use cases have been reported in the past [11, 1, 23, 9, 20, 22, 28]. These natural language processing (NLP) approaches can be classified into the following broad categories:

1. Lexical analysis of requirements to identify occurrences of certain patterns [28, 9]

2. Semantic analysis with user provided glossary of domain concepts [1]

3. Semantic analysis to identify *single* aspect of a behavior model such as conceptual Entity-Relationship (ER) models [20] or early aspects [23]

4. Semantic analysis to identify *multiple* aspects of a behavior model such as ER and activity or scenario diagrams [11, 22]

The goal of our work is to extract multiple aspects of the behavioral model from industrial use cases – closely resembling the works in [11, 22]. However, these two approaches suffer from several limitations which may prevent their application to industrial use cases. These arise largely from the relatively monolithic nature of the respective architectures, and from the fact that they rely, heavily, on extensive hand-crafted lexical semantic resources (typically, thousands of application-specific verb categorization entries). We return to these observations in detail in Section 5. In contrast our implementation is configurable and can be easily adopted to different domains and languages.

Note in Figure 1 that our implementation uses a shallow parser while both [11, 22] use a form of semantic parser. Along with the grammatical roles of phrases, the deep parsers yield configurational information (i.e., how the phrases are related to each other). This information can be used to better understand NL text. However, as has been reported in [15], deep parsing is extremely vulnerable to noise (unwanted information) in text. In contrast shallow parsers and, consequently, our implementation are much more robust in handling noise. We have observed (see section 4)that use case text frequently contain noise in forms of parenthesis, acronyms, labels, etc. This leads us to believe that a shallow parser based linguistic engine, like ours, is better suited for parsing use case text.

Further, we use a Finite State Transducer (FST) based implementation of shallow parser [5], which is faster than several other implementations. This enables our linguistic engine to support online analysis. By online analysis we mean analysis of the text as and when they are created by the author. None of the existing techniques including the ones described in [11, 22] have online analysis capabilities. This capability is important as we realize that even with the most sophisticated NLP technique, the translation process preserves most problems from the source NL description – omissions, inconsistencies, incompatible granularities, poor structurings, unmeasurable properties, etc. In online analysis, authors can be informed about the errors as and when they are committed. Thus, in a way, the authors can be induced to create error free use cases.

Further, [11, 22] do not report results of applying their approaches to industrial use cases. In fact, very few published results of applying linguistic analysis in industrial requirements context are available. Wilson *et al.* [28] report their findings on lexical scans of requirements developed in NASA. Gervasi and Nuseibeh [12] report results of applying the validation technique of [1] to part of a NASA requirement specification. Törner *et al.* [26] applied the criteria defined in [22] using a manual review process to report defects in 43 automotive use cases. However, none of these works discusses the effectiveness of the underlying linguistic analysis technique itself.

The specific contributions of this paper can, therefore, be identified as:

1. A novel application of a robust linguistic technique for deriving a computer model of use case descriptions based on outputs of a general purpose shallow parser.

2. Results of applying a state of the art linguistic technique to industrial use cases.

2 The Linguistic Analysis Engine

The linguistic analysis engine (LE) consists of multiple configurable components (see Figure 1) integrated using the Unstructured Information Management Architecture (UIMA) [10, 7]. UIMA is an open, industrial-strength, scalable and extensible platform for building analytic applications or search solutions that process text or other unstructured information. UIMA provides a simple but rich representation for unstructured information, which can be used by different analysis compoeenents to share and extend their analysis results. Additionally, UIMA enables developers to compose and configure such components and also to combine them with existing/third-party components.

Configurability and composeability aspects of the UIMA pipeline lends our implementation a unique advantage over traditional problem specific architectures as described in [11, 22]. Use case descriptions can contain NL text from a variety of application domains, ranging from aerospace to communication systems to ATMs. The current LE is developed with domain independence as focus, thus yielding almost uniformly accurate text processing across various domains. However, certain domain specific activities may require greater accuracy. The compose-ability aspect is a great help in retargeting the domain independent LE to a domain specific text by allowing easy a replacement for an existing component. For instance, retargeting the lexical analysis to a health-care-domain use case can be easily accomplished by swapping in a medical text-aware lexical processor.

Configurability is a great boon when we need to adapt the existing components to changing input language, changing structure, etc. For instance, by configuring the Lexical Processor to refer to a German rule set and plugging in a shallow parser for German we can extend the capabilities of the existing LE to handle inputs in German.

Below we will refer more than once to a *training corpus*, which was used to develop the components and subsequently, to create linguistic knowledge bases. The *training corpus* is a selection of 300 different use case text from an initial pool of use cases collected from practitioners in the industry and examples in the published literatures [22, 6]. The primary objective while selecting for the corpus was to vary the application domains of the use case text. We also ensured that UCD sentences were approximately in accordance to the recommended best practices [6]. The use cases were compiled from the application domains of ACCOUNTING, AEROSPACE, CONTROL SYSTEM, DISTRIBUTED SYSTEM, NETWORK MANAGEMENT, OPERATING SYSTEM and SERVICE MANAGEMENT.

In the following sub-sections we describe the primary components of our LE. Each sub-section describes the *contributions* of the respective component to the processing; briefly the *implementation*; and the *rationale* behind our design choices.

2.1 Lexical Processor

The lexical processor provides the following three services (collectively referred to as *Lexical Services*):

Tokenization: Broadly speaking, breaking text into words and/or punctuation marks and end-of-sentence detection;

Lemmatization: Determining the base form of a word;

Morpho-syntactic analysis: Associating lemma forms with contextually appropriate part-of-speech (POS) information.

For applications in open-ended domains (like ours), it is essential that lexical services are provided by a component which is not limited to a single application/domain. We thus use lexical analysis technology which embodies lexical knowledge not only of unconstrained English but also for a hundred other languages (including German, French, Spanish, Italian, Russian and Chinese). This greatly facilitates adaptation of the system to different domains. This however, exposes us to POS ambiguities of language at the lexical level. We use an implementation of the *Robust Risk Minimization* [29] for POS disambiguation. RRM is a high-performance linear classification technique, which has been shown useful for a number of text analysis tasks including, in particular, part-of-speech tagging [2]– which is essential for correct shallow parsing.

Consider an example snippet, *"The customer enters the withdrawal amount. If the balance is more than the amount, the ATM returns cash in $s to the customer and changes his account balance"*. The lexical processor will tokenize, lemmatize and disambiguate POSs, after which the a stream of tokens will be created as follows:{*DT:the NN:customer VBZ:enter DT:the NN:withdrawal NN:amount PUNC:. CS:if DT:the NN:balance VBZ:is JJR:more CS:than DT:the NN:amount PUNC:, DT:the NNP:ATM VBZ:return NN:cash IN:in _:$s TO:to DT:the NN:customer CC:and VBZ:change PRP$:his NN:account NN:balance*}. Where the POS acronyms mean the following: *CC-Coordinating Conjunction, CS-Subordinating Conjunction, DT-determiner, IN- preposition, JJR-comparative adjective, NN-noun, NNP-proper noun, PRP$-possessive pronoun, VBZ-verb 3rd person sing. present.*

2.2 Shallow Parser

We use a general purpose shallow parsing component that acts as a syntactic analysis system for the identification of phrasal, configurational, and grammatical information in free text documents.

The shallow parser component (described further in [4, 5]) is fully realized as a cascade of a dozen finite-state transducers (FSTs), which works on tokens output from the lexical processor. Broadly speaking, an FST identifies phrases, clauses and grammatical functions of phrases by recognizing patterns of POS of tokens and already identified phrases and clauses in the text. At the lowest level of the cascade are simple noun and verb group grammars which mark noun phrases (NP) and verb groups (VG) based on patterns defined on Tokens and their POSs. Thus the example text will be marked up with simple phrase boundaries as: *"[NP:The_customer] [VG:enters] [NP:the_withdrawal_amount] PUNC:.*

CS:if [NP:the_balance] [VG:is] JJR:more CS:than [NP:the_amount] PUNC:, [NP:System] [VG:returns] [NP:cash] IN:in _:$s TO:to [NP:the_customer] CC:and [VG:changes] [NP:his_account_balance].

The FSTs in the later stage of the cascade seeks to build complex phrases, identify clause boundaries (TC) based on patterns of already identified tokens and phrases. For instance, a simple pattern that says a preposition followed by a noun phrase is a prepositional phrase (PP). Thus, the phrase "*to the customer*" in the example will be identified as a prepositional phrase. In fact, post identification of clauses and complex phrases, the example stream of tokens will be marked up as: *"[NP:The_customer] [VG:enters] [NP:the_withdrawal_amount] PUNC:. [TC:if_[NP:the_balance]_[VG:is]_JJR:more] [TC:than_[NP:the_amount]] PUNC:, [NP:System] [VG:returns] [NP:cash] _:$$ [PP:to_[NP:the_customer]] CC:and [VG:changes] [NP:his_account_balance].*

The final set of FSTs construct predicate-argument clusters, and mark grammatical functions such as subjects and objects . For the example text, the cascade will mark the phrase *"The customer"* of the first sentence, and the phrases: *"the balance"* and *"system"* of the second sentence in subject roles. Further, the phrases: *the withdrawal amount, cash* and *his account* will be marked as objects.

The choice of a shallow parser over deep parser is driven by our need to handle noise in the input text. Notice the treatment of the extra grammatical phrase "*in $s*" in the example text. While the shallow parser ignores the phrase, it does not falter in identifying the remaining syntactic elements. Our analysis of the *training corpus* show frequent existence of such noise in the input text – mainly in the form of acronyms, parenthesized comments and programming constructs (e.g., `account.balance, CONTINUE`, etc.

As discussed in [5], the cascade of FSTs is adapted to be pluggable in a UIMA pipeline. The shallow parser is generic–in a sense, that it is based on rules for unconstrained English. Thus, as long as, a UIMA enabled lexical processor marks up the *lexical information* the shallow parser can handle any input text. Using a shallow parser provides us the additional benefit of configurability. Compared to deep parsers, shallow parsers encapsulate a more compact rule base and are easier to develop and maintain. There exists a number of shallow parsers that are available off the shelf, including those for German, Spanish, French and other Latin based languages. Therefore, extending the capabilities of the current linguistic engines to such languages becomes much easier than what it would have been if deep parsers were used.

Also, the choice of an FST based shallow parser implementation over others is driven by our needs for fast processing. Finite-state methods have been recognized for their efficiency [13, 25] and have been shown to be useful for tasks like lexical lookup, morphological analysis, part-of-speech determination, and phrase identification. However, there has been considerably less focus on developing such methods for complex phrase and clause build-up, or for the identification of grammatical function. We build upon prior work in pushing shallow analysis further into the realm of semantic and discourse processing.

2.3 Dictionary Concepts Annotator

This is the component that introduces semantic information into the analysis. The Dictionary Concepts Annotator uses an extensible and externalizeable knowledge base[1]-*domain dictionary*.

The domain dictionary is a compilation of commonly occurring verbs and their association to a set of pre-defined semantic classes. A semantic class is a kind of action found in a UCD, viz.,INPUT, OUTPUT, READ, WRITE, GIVE, GET, CREATE, QUERY, UPDATE, DELETE, DELEGATE, START, STOP AND UNCLASSIFIED. A verb entry can be associated to multiple semantic classes with differing degrees of confidence. Consider the verb group "*changes*" in our example UCD text. While in this particular context, the verb implies an UPDATE action, in a separate instance it could imply an OUTPUT action – consider e.g., "*System changes the display on the screen.*". Therefore, in the domain dictionary, the verb "change" will be linked to both UPDATE and OUTPUT. Further, if we know that it the verb "change" appears more frequently in the context of an UPDATE action than in the other context, we record this information in the *domain dictionary* by assigning higher confidence to the association of "change" to the semantic class UPDATE than to the association of the verb at to the semantic class OUTPUT. The domain dictionary is populated with an initial set of entries and their respective confidences by scanning the *training corpus* and relative frequency of contexts of the verbs.

Using the domain dictionary, the dictionary concepts annotator assigns a classification confidences to the verbs. Thus, for our example text the verb "changes" will be assigned following semantic classes: UPDATE- 78%, OUTPUT-15% and INPUT-6%. Also, "enter" will be assigned the semantic classes of {WRITE 81%, INPUT 12%, OUTPUT 7%} and "return" will be assigned {OUTPUT 79%, GIVE 21%}.Actual classification of the action is inferred by the component *Process Builder* later in the pipeline.

Note that by choosing an extensible and externalizable domain dictionary we support configurability of the linguistic engine to different languages and domains. Since the subsequent analysis (see sections 2.5 and 2.6) is based on semantic classes, modifying the domain dictionary by adding individual entries does not require modifying any other component. Notice also that unlike in [11], the syntactic analysis (by the shallow parser) in our implementa-

[1]Currently an XML blob; if needed, it could be represented as a Relational DB or an OWL ontology.

tion is independent of semantic analysis (and vice versa). Thus, the effect of a missing entry in the *domain dictionary* is restricted only to an *unclassified action*.

2.4 Anaphora Resolver

The Anaphora Resolver component provides a unique advantage to our linguistic engine by identifying usage of pronouns and replacing them with the representative noun phrase. This enables our linguistic engine to process pronouns, in contrast to many others including the ones described in [11, 22] are unable to do so.

The Anaphora Resolver component uses a specialized version of the algorithm described in [18]. The algorithm described in [18] is specially adapted to shallow parser output, to minimize dependencies of typical anaphora resolution algorithms on configurational (syntactic) information derived from a deep parser. The original algorithm is specialized by additional rules whereby a pronoun in the position of an actor is replaceable only by noun phrases that also appear in an actor role. The average accuracy of the original algorithm as reported in [18] is around 75%. With the specialization, the accuracy is increased to 95.2% for use cases that are adherent to best practices (see section 4.3). Using the anaphora resolver, the LE relates the pronoun "*his*" in the example snippet to the entity "customer".

2.5 Context Annotator

A context of a use case is the world around the use case. It is the set of entities and relations between them which get affected by execution of the use case. The context annotator component identifies instances of actors, system, business items and other use cases in an use case text.

The context annotator groups a noun entity with its variant and treats them as a single candidate. For instance, a candidate "CD" will represent noun entities: CD, CDs, compact disc and compact discs. The context annotator evaluates each candidate for its likelihood to be an ACTOR, a SYSTEM, a USE CASE, a BUSINESS ITEM or a PARAMETER.

The context annotator collects the candidates by locating the head noun entity among phrases that play the grammatical roles of subject or object in UCD sentences. For each identified candidate, the context annotator identifies its classification using a scoring mechanism. To each identified candidate, a score is assigned based on classification of the main verb and the role of the candidate in the sentence of consideration. The final verdict is reached by summing the scores for each candidate across all the sentences and picking the verdict for which the entity has the maximum score. Consider again the example snippet, *"The customer enters the withdrawal amount. If the balance is more than the amount, the ATM returns cash in $s to the customer and changes his account balance"*. In the first sentence, the noun entity "*customer*" appears in an initiator role of an INPUT action and additionally in the second sentence it appears as receiver of "*cash*". Summing the scores for these information the context annotator will know that "*customer*" is an ACTOR. Further, since in the second sentence, the entity "*ATM*" is "*returning*" (implying an OUTPUT or GIVE action) something ("*cash*" in this case) to an ACTOR, the context annotator reasons that *ATM* is a SYSTEM.

The scoring knowledge is externalized in form of a set of weighted rules and can be updated to reflect learning of new instances. Also, the scores are determined based on semantic classes (see section 2.3) of the actions and not on the verbs of the actions. Thus the context classification mechanism is not affected by addition of new verbs to the *domain dictionary*.

2.6 Process Builder

The process builder builds a model of the process described in a use case by identifying the sequence of use case actions. This extracted model forms the basis for subsequent analysis/automation activities. Figure 2 depicts the meta-model for the process. The metamodel was derived on the basis of the training corpora and summarizes our understanding of the information contained in typical UCDs. At the most abstract level, a use case contains a series of sentences that express one or more actions initiated by some actor/agent – e.g., a system or its user. Each action may have a set of parameters that are *defined* – assigned a value or *used*.

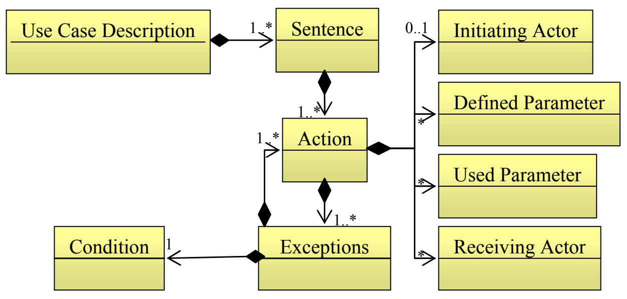


Figure 2: Use Case Description Metamodel

The process builder builds the process based on another externalize-able knowledge base called use case pattern knowledge base (UCKB). Like other knowledge bases, UCKB is compiled using the training corpora and can be updated on the fly. The patterns themselves are written using JFST[2] [4].

As a first step the process builder isolates the fragments of text that define expressions/ conditions. This is done by looking for patterns of condition in the text according to UCKB. In the domain of use cases expressions can appear as guards for conditional statements or as expressions that

[2]A detailed discussion on JFST is outside the scope of this paper, but essentially JFST patterns are like regular expression patterns over tokens, phrases and already identified patterns.

define query of objects. For instance, in our example: *"The customer enters the withdrawal amount. If the balance is more than the amount, the ATM returns cash in $s to the customer and changes his account balance"*., the sentence fragment :"*If the balance is more than amount*" is recognized as a condition and is not processed by the process builder (and instead is processed by the component *expression builder*).

In the subsequent step, the process builder scans for known patterns of use case actions and associates parameters and actors to actions. Thus for the example, it creates three action parameter groups: *customer* $\rightarrow$ *enter(withdrawal amount); ATM* $\rightarrow$ *return(cash)* $\rightarrow$ *customer;* and *ATM* $\rightarrow$*change(account balance).* The $\rightarrow$ indicate flow of action and $x \rightarrow P(y) \rightarrow z$ implies x is the initiator of action P with argument y where z is the recipient of the effect of action P.

In its next step, the process builder infers the type of action. In the context of use cases, the relevant types are INPUT, OUTPUT, CREATE, READ, UPDATE, DELETE and INCLUDE. The LE knows from the *context annotator* that "*ATM*" is a SYSTEM and "*Customer*" is an ACTOR. Further, from the dictionary concepts annotator it knows classification of the verbs as: "*enter*"-$\{$WRITE 81%, INPUT 12%, OUTPUT $7\%\}$; "*return*"- $\{$OUTPUT 79%, GIVE $21\%\}$ and "*change*" - $\{$UPDATE 78%, OUTPUT 15% INPUT $6\%\}$. The process builder infers the type of action based on the classification of verbs and the classification of the initiators and receivers. In the context of use cases, an ACTOR initiating either of WRITE, INPUT or OUTPUT actions is mapped to an action of type INPUT. Thus the action with the verb "*enter*" is unanimously voted to be an INPUT action. Similarly, the action "*return*" gets mapped to an OUTPUT action. Interestingly, in the case of the verb "*change*" the process builder infers a SYSTEM doing an UPDATE or a SYSTEM doing an INPUT or a SYSTEM doing an OUTPUT. In the context of use cases, while the first choice would mean an UPDATE the later two choices would imply an OUTPUT. The conflict is resolved by choosing UPDATE which has a higher confidence– UPDATE-78% vs OUTPUT (15+6=) 21%. Finally, used and defined parameters are identified based on the classification of use case action and the relation of the parameter to the action. The main arguments of INPUT, CREATE, UPDATE are classified as defined parameters while those of the OUTPUT, READ, DELETE actions are classified as used parameters.

Notice that an externalized and extensible UCKB lets one configure the process builder component for languages other than English. Also, except for the classification of actions and parameters, the algorithm is independent of the information in the *domain dictionary*. This enables us to handle noise more effectively– by allowing partial extraction of the model elements. Again, this is in contrast with the semantic-tag-based parsing described in [11].

2.7 Expression Builder

The expression builder parses the expression fragments to form conditional/boolean expressions and associates them to the relevant actions. As discussed in the section 2.6, the component *process builder* identifies sentence fragments that define expressions.

Similar to the process builder, the first step for the expression builder is to associate actions and parameters to form a predicate-argument structure (i.e., of the form f(X)). Thus, the sentence fragment in the example snippet :"*balance is more than the amount*" is parsed into $is\_more(balance, amount)$. The following translation is straightforward. The component uses an externalizable knowledge base of comparison operators, existential operators and logical operators to infer that "is_more" implies the operator "$>$". The association of the condition to actions is done simply by identifying the actions within the sentence containing the fragment defining the condition. The example snippet will be updated by the expression builder by associating the condition $balance > amount$ to both the actions in the second sentence. The final output model from the LE is shown in Figure 3.

The expression builder in its current form can only treat *Boolean Expressions*. It is currently limited in treating boolean operators other than the simple "AND" and "OR" operators. It treats most of the common comparison operators and can also identify some of the existential constraints.

3 The Online Analysis Environment

As a proof-of-concept we have developed a prototype implementation of an online analysis environment (OAE) for use cases. The purpose of the OAE is to provide an environment to the authors of use cases for creating consistent, unambiguous and error free use cases in NL. Among other features, the OAE provides a spell-checker backed text editor for authoring UCDs in plain English. The OAE feeds the input UCDs to the linguistic engine and the output *model* is used for correctness analysis. In particular, the OAE runs a set of validation checks on the model that can be broadly classified into the following:

Stylistic checks for English sentences e.g., voice, complexity of sentences (number of actions/actors), use of anaphora.

Completeness checks of use cases e.g., missing actors and actions, missing parameters.

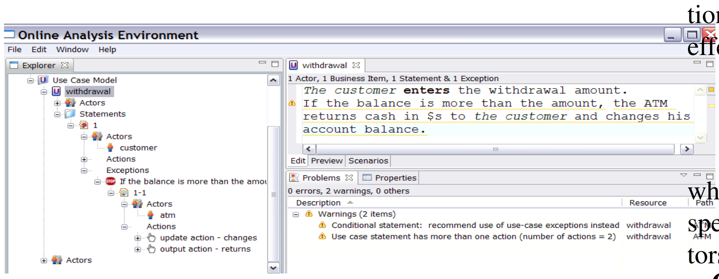
Structural checks for the model e.g., consistent use of aliases, dangling use-case references.

Flow checks for data and the control flow e.g., analysis for consistencies such as attempts to *read* variable values before they are *defined*.

Ownership checks that validate accessibility of data from the point of view of actors.

Path checks that validate accessibility of data from the point of view of use case scenarios and their sequences.

OAE provides feedback to the users based on these checks. A feedback includes a diagnostic of the problem and its severity. Severity is specified using the three categories of "info","warning" and "error". Additionally, markers are created that inform the users of the location in the UCD that cause the failing tests. Figure 3 displays a screenshot of the OAE.



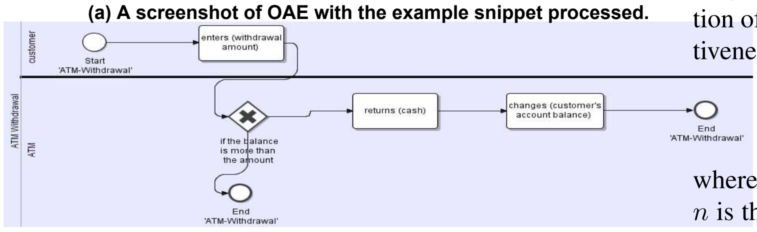
1 Actor, 1 Business Item, 1 Statement & 1 Exception

```
The customer enters the withdrawal amount.
If the balance is more than the amount, the ATM
returns cash in $s to the customer and changes his
account balance.
```

0 errors, 2 warnings, 0 others

```
Description                                                              Resource    Path
Warnings (2 items)
  Conditional statement:  recommend use of use-case exceptions instead   withdrawal  ATM
  Use case statement has more than one action (number of actions = 2)   withdrawal  ATM
```

(a) A screenshot of OAE with the example snippet processed.



(b) The final output of the Linguistic Engine in BPMN notation.

Figure 3: The Online Analysis Environment

To assist the authors of a UCD in understanding the inferred conceptual model, the OAE provides visualizations of the flow using the Business Process Modeling Notation (BPMN) [14] – an example is shown in Figure 3. From a process perspective, the UCD authors would use an iterative process consisting of UCD editing, performing validation checks, and visualizing BPMN until a satisfactory UCD is developed[3].

4 Experiment

An experiment was conducted to evaluate the linguistic engine and to answer the following research questions:

Q-Accuracy: What is accuracy of the current configuration of the linguistic engine? In order to assess *Q-Accuracy*, we used the standard metrics of precision ($Prec$) and recall (Rec) computed as:

$$Prec = \frac{|A_c \bigcap A_g|}{|A_g|}$$

and

$$Rec = \frac{|A_c \bigcap A_g|}{|A_c|}$$

[3]Due to space considerations, we do not provide detailed discussion of the OAE.

where A_c represent the set of use case actions in a correct model and A_g represent the set of those in the corresponding automatically generated model. Note that by measuring the LE's performance on use case actions, we are able to evaluate LE's ability to identify actors, parameters and their relation to the actions.

Q-Translation: What is effectiveness of translation of the current configuration of the linguistic engine? To answer *Q-Translation* we evaluated the fraction of information not lost in translation. The estimate for the translation effectiveness was computed as follows:

$$T = \frac{|A_c \bigcap A_u \bigcap A_g|}{|A_c|}$$

where A_u represent the set of actions that are under-specified (that miss information on either parameters or actors) in the automatically generated model.

Q-Filtering: How effectively does the current configuration of the linguistic engine filter noise? The filtering effectiveness was computed as:

$$F = \frac{n}{N}$$

where N is the number of net instances of noise in text and n is the number of such instances that were ignored by the analysis engine.

Q-Pronoun: How effectively does the current configuration of the linguistic engine resolve pronominal anaphora? The anaphora resolution effectiveness was computed as:

$$P = \frac{m}{M}$$

where M is the net number of pronoun usages in text and m is the number of correct resolution of such pronouns.

4.1 Experiment Process

We solicited UCDs from business analysts and other practitioners in the industry. We also collected use cases from refereed publications [3, 6, 8, 9, 21]. Eventually, we amassed 31 industrial applications and 10 academic applications. The use cases described applications in industrial sectors of *Aviation, E-commerce, ERP, Finance, Gaming, Health Care* and *IT*. A sample of 80 UCDs was created by randomly selecting 57 UCDs from industrial applications and combining it with 23 academic UCDS. We ensured that the test population has at least 1 representative from each application in the population.

To prepare the UCDs for the experiment we manually extracted the relevant text from the original files. While doing so we preserved format of the text in the form of enumerated lists and indentations, if any. For the tables in the original UCDs, we extracted entries from the relevant cells and recreated the text. We corrected for 6 minor grammatical and 14 spelling errors.

	# Sentences		% Labeled		# Itemized lists		Max Nesting Depth		# Indentations		If Tabulated	
	Ind	Acad	Ind	Acad	Ind	Acad	Ind	Acad	Ind	Acad	Ind	Acad
High	77	16	100%	8.8%	5	2	2	2	3	1	-	-
Avg	12.84	7.57	9%	6.1%	1.09	0.17	-	-	-	-	-	-
One	-	-	22.8%	17.4%	54.4%	13.0%	56.1%	13.0%	15.8%	4.3%	33.3%	0.0%

Table 1: The structural information of the test population

The UCDs were analyzed using the LE. To compute A_g and A_u all actions in the generated model were evaluated they are correctly specified, mis- identified or under- specified. Additionally, to compute A_c we analyzed the generated model for unidentified actions. The measurements were repeated for cross-validation.

4.2 Subject Use Case Descriptions

UCDs in our test population varied significantly in their structure. In order to characterize and report the structure of the text in the test population, we examined each UCD for the following attributes (see Table 1): Total number of sentences; Fraction of sentences that are labeled with names indicating their purpose or location in the text, e.g. "Insert Action", "Step 5", etc; Number of itemized or bulleted lists;Deepest level of nesting in hierarchical itemized lists, (0-none, 1-sentence level, 2-sub-sentence level, etc.); Number of instances where indentations were used to depict scope of a sequence of actions as opposed to explicit sentences/keywords (e.g.,"begin", "end"); Whether or not the UCD is described in tables.

In Table 1 we summarize our findings for academic (*Acad*) and industrial (*Ind*) UCDs. Notice that the academic use cases are relatively less noisy (less number of itemized lists, less fraction labeled, no tabulation etc.). There was prevalent use of templates in UCDs collected from the practitioners. This is the primary reason that many of the industrial UCDs had tabulated descriptions while none of the academic UCDs had them.

Also, to characterize the test population with regards to its content we identified the sentences that were not "Interaction Sentences" – *i.e.*, the sentences that did not describe an action by an agent (see Table 2). To facilitate reporting we classify the "non-Interaction Sentences" into the following types: Programming constructs, that resemble a code directive or a command e.g., `GOTO, Do-while, Repeat-until`; Non functional Sentences, sentences that define some nonfunctional attribute e.g., `System shall be fast enough to ...`; Execution Sentences that describe result of executing a use case sentence, e.g., `The query returns 40 listings`; Definitive sentences, that define attributes of an entity e.g., `The online customer must have an account id`; and Sentences in colloquial English, e.g., `If yes print ''yes'' else quit`. The academic use cases consistently contained fewer non-interaction sentences. This confirmed the intuition that academic use cases conformed to the recommended best practices. Notice also that highest observed percentage of Colloquial Sentences and Definitive Sentences (66.7%), owing to one very detailed UCD. On an average, however, the academic use cases are still more adherent to the best practices. Therefore, in the following section we compare the performance of the analysis engine on academic use cases to that of the industrial use cases. This is with the intention to contrast the performance of the analysis engine on use cases that adhere to best practices with the performance on those that don't.

4.3 Evaluation

Due to lack of any pre-existing studies, we are unable to do a comparative evaluation. Instead, we would try and quantify in absolute terms the performance of LE. Table 3 summarizes the results of our measurements. Apart from the measurements of precision ($Prec$), recall (Rec), translation effectiveness (T), filtering effectiveness (F) and anaphora resolution effectiveness (P), we also report on the number of occurences of noisy text (N). To compare performance of the LE on use cases more adherent to recommended best practices to those that are less adherent, we present our results in separate buckets of industrial (Ind) and academic (Acad) use cases. To characterize the spread of our measurements we present the averages, the standard deviations, the minimum observed values, the maximum observed values, percent of use cases that had the maximum values and percent of the use cases on which LE performed less than $average - 1 \times SD$.

Q-Accuracy: Precision is a measure of exactness or fidelity of information retrieval, whereas recall is a measure of completeness. Thus in our context, a 100% precision means that all identified actions in a generated model are correct. A 100% recall, on the other hand, implies that all actions described in the use case text have been identified in the generated model. The LE gave an average of 89.4% precision and 92.5% recall. This means that on an average, the LE misidentified 10.6% of actions and missed out only 7.5% of the actions described in the input text. Note that the precision and recall numbers on academic use cases are better than the industrial use cases – this confirms the intuition that the LE has higher accuracy for better quality use cases. In fact, only 36.8% industrial use cases returned 100% precision and just 57.9% of them had 100% recall –compare against only 100-82.6 = 17.4% academic use cases (4 out

	% Program Constr.		% Non-functional Sent.		% Execution Sent.		% Definitive Sent.		% Colloquial Sent.	
	Ind	Acad	Ind	Acad	Ind	Acad	Ind	Acad	Ind	Acad
High	28.6%	10.0%	50.0%	30.0%	50.0%	30.8%	25.0%	66.7%	42.9%	66.7%
Avg	2.6%	0.4%	3.7%	1.3%	7.1%	2.0%	4.3%	3.8%	6.0%	3.8%
One	14.0%	4.3%	19.3%	4.3%	36.8%	8.7%	31.6%	8.7%	38.6%	8.7%

***Legend**: High=Highest Percentage in any UCD of our test population; Avg=Average for all UCDs in our test population; One= % of UCDs having at least 1 sentence with the examined property*

Table 2: The test population content

of 23) not having 100% precision or recall. However, when we look at the number of use cases for which the LE performed less than $average - 1 \times SD$, we find that only for 15.8% use cases the LE had a precision less than (86.1-14.8 =) 71.3% and, likewise, only 15.8% of them had a recall less than 78.1% recall. Considering that we allowed minimal alterations to text during data preparation, these results are highly encouraging.

Q-Translation: While precision and recall just evaluates the ability to retrieve concepts from the text, the translation effectiveness reports additionally on the quality of retrieval. A 100% effectiveness implies that all correctly identified actions in the generated model have the same information content as in the text form. The LE has a combined translation effectiveness (T) of 84.2%. It has a better T for academic use cases than for industrial use cases.

Q-Filtering: The filtering effectiveness (F) is very high (90.9% overall, 89.7% industrial and 100% academic) for the current LE. The average instances of noise N in industrial use cases is 5.8. This is very high, considering that the average number of sentences in them is 12.84 (see Table 2). This justified our choice of shallow parser over a deep parser.

Q-Pronoun: A total of 41.3% use cases had one or more instances of pronoun usage (not shown in Table 3). This justifies the need to plug in a anaphora resolver in the LE pipeline. Overall, the anaphora resolver had an effectiveness (P) of 77.9%. As expected, for better quality use cases it performed better. The average accuracy of the original algorithm as reported in [18] is around 75%. With the specialization, the accuracy is increased to 95.2% for academic use cases but only at a cost of reduced accuracy for industrial use cases.

4.4 Analysis of Errors

We drilled down into low-score use case descriptions to identify the possible causes of failure. We counted 171 failures in the analysis of our test population. The primary reason for the failures were classified into the following list. 139 of the 171 failures could be attributed to the following recurring reasons:

Co-mingled non-Interaction Sentences were the largest contributors to the failures, accounting for 40.7% of failures. While we safely handled 53.2% of the total non- Interaction sentences, the classes of sentences that we failed to handle were "labels", passive-voiced definitive and imperative sentences and the passive- voiced business rules.

Spatial patterns like scoping through indentations and itemizations into lists contributed to 20.7% of failures. The spatial patterns obfuscated the sentence boundaries. This contributed to unidentified actions or parameters and misidentification of the actors.

Incorrect English, due to misspelt words, programming language elements, colloquialisms and other orthographic characteristics, confused the lexical processor in 20.4% of the situations. This subsequently affected configurational analysis in the shallow parser and eventually affected the model builder.

Information spread in multiple sentences is a source of confusion primarily for the model builder. It contributed to 17.8% of the failures. Consider the following snippet. `The system presents the user with a confirmation page. The confirmation page presents to the user a Record Number.` While a human reader may be able to relate these sentences to a unique action of output of "the confirmation page" from "the system" to "the user", the model builder is fairly naive in such correlation. It would instead yield two separate output actions, one from "the system" and the other from "the confirmation page".

Recursive structures in English sentences are a source of confusion for the shallow parser in 0.4% of situations. This is, in fact an inherent limitation for the parser (arising out of our choice of FS cascades). For instance consider the following sentence:

```
System displays the amount updated
using the information entered by the
user using the preferred currency
format.
```

The parser failed in recursively relating the colored *clauses*. The model-builder component corrects for some of those errors, but not when there is a long distance relation between clauses. Consider the example cited above, the model builder fails to recognize that the 'blue' clauses relate to the "display" of page and instead it relates the last two clauses to "the information".

Some of the limitations that we find above can be addressed in future. For instance, we can plug-in analytic engines in the LE that filters out irrelevant sentences like labels more effectively. The spatial patterns can be pro-

		Prec	***Rec***	***T***	***N***	***F***	***P***
Combined Avg		89.4%	92.5%	84.2%	4.40	90.9%	77.9%
Average	Ind	86.1%	91.3%	80.9%	5.58	89.7%	71.4%
	Acad	97.4%	95.5%	92.2%	1.48	100.0%	95.2%
SD	Ind	14.8%	13.2%	20.5%	9.88	19.0%	40.0%
	Acad	6.1%	10.0%	12.3%	4.61	0.0%	14.3%
Max	Ind	100.0%	100.0%	100.0%	56	100.0%	100.0%
	Acad	100.0%	100.0%	100.0%	22	100.0%	100.0%
Min	Ind	47.8%	50.0%	5.9%	0	25.0%	0.0%
	Acad	80.0%	71.4%	50.0%	0	100.0%	57.1%
% of max	Ind	36.8%	57.9%	26.3%	N/A	65.9%	50.0%
	Acad	82.6%	82.6%	60.9%	N/A	100.0%	88.9%
%<(Avg-SD)	Ind	15.8%	15.8%	10.5%	N/A	18.2%	20.8%
	Acad	17.4%	17.4%	8.7%	N/A	100.0%	11.1%

Table 3: The summary of measurements

cessed by enabling the UDE to accept template based inputs. This should also ease the handling of multi-sentence descriptions, as the scope of description will then be evident from the templates.

4.5 Threats to Validity

All empirical studies suffer from threats to their internal and external validity. For this work, we are primarily concerned with threats to internal validity, especially *experimenter bias*, since some of the counting of modeling elements are subjective to experimenter's understanding of the system. An independent evaluation of our results, possibly during customer studies, will be conducted to cross-validate our findings.

Although a lot of care was taken to create a representative population, there still remains uncertainty about applicability of our results in general. This is due to the very nature of natural language text, and can be alleviated through repeated case studies and experimentations.

5 Related Work

In this section, we focus on two works [11, 22] most closely related to ours. Fliedl *et al.* argue for a more abstract (and less domain-dependent) 'interlingua'—conceptual predesign schema [11]. This is not unlike our use of a meta-model, in seeking to remain as domain-agnostic as possible. Fliedl *et al.* [11] rely on the linguistic notion of thematic grids, and the ability to tag verbs with their affinities for semantic labels like (Fillmorean) AGENT, THEME, and GOAL; subsequent processing would associate nominal arguments in the vicinity of the verb with appropriate slots in the verb frame; later processing still would map ACTORS and THEMEs to eg. actors and service parameters. What is, in effect, a supertagger provides lexical services not just for traditional morphosyntactic analysis, but also projection of relational lexemes (typically verbs) into a space of relatively fine-grained semantic classes. This is the basic information for associating predicates with their arguments; the ability to do that outside of any specific domain semantics makes it possible to claim relatively broad applicability of the system analytic component.

Still, a system like this remains dependent on the size and scope of its (semantic) lexicon. Even if this has been instantiated for 16,000 (German) entries, encountering an out-of-vocabulary verb would expose the system's fragility—due to its reliance upon a semantic verb class tag. This is in marked contrast to our approach, where semantic relationships are derived directly from configurational and grammatical information in a shallow parse, which can always be assumed to be available, as it is itself constructed on the basis of high-precision POS tagging, which can be relied to yield a tag stream even in out-of-vocabulary situations (this is, in fact, one of the particular strengths of an RRM-based tagger).

Additionally, it is not clear what exactly are the mechanisms for collecting and manipulating the set of nominal entities in the vicinity of a tagged verb, in the process of assembling the verb frame; frame-based 'parsers' can be sensitive to subtle syntactic characteristics, and without a performance evaluation, it is hard to assess the robustness and scalability of Fliedl *et al.*'s tagged output post-processing. Finally, we note that dependence upon a semantic lexicon typically translates into problems in retargeting a system to novel applications and/or languages—in contrast to a staged pipeline like ours, where POS taggers for different languages can be assumed (or relatively easily trained), and FS-based shallow parsers can be developed with relatively small effort.

Similar observations hold for Rolland & Achour's approach [22]. They adopt a different framework for domain-free semantic representation, namely that of (a variant of) case grammar, where a nominal entity can fill multiple case slots. Whatever their argument for case-based representation (namely the close relationship between a case-based semantics of language and a generic use case model), such representation requires, in a similar fashion to the frame-based one, access to an instantiated semantic lexicon with entries for thousands of semantic patterns.[4] Questions arise again (especially in the absence of performance analysis of their "prototype tool") about the robustness and scalability of a case grammar parser, adaptability across domains, and transportability a cross languages.

6 Conclusions and Future Work

We presented 1) a novel application of UIMA pipeline architecture and a set of heuristics for extracting and relating

[4]Such patterns are not only lexically anchored, and in order to represent the semantics of a clause in a use case specification a semantic pattern needs to be additionally labeled with its position in an ontology of semantic patterns, which mediates between the linguistic semantics and the use case semantics. This would make it very hard to argue for deriving such a lexicon by some (semi-)automatic means.

key concepts in textual use case descriptions and 2) results of a first large scale analysis of industrial textual use cases. Our approach is able to extract information from diverse set of sentences that are prevalent in industrial use cases, improves the robustness of analyses results, and leads to a domain neutral means of analyzing requirements for various application domains. We also described the specific set of heuristics we have implemented in a prototype. We have used this prototype to analyze 80 use cases that were defined in industrial and academic environment and compared the two use case populations and evaluated effectiveness of our approach. The results indicate that our approach is very promising.

For future work, we would like to apply validation checks in our current prototype to the industrial use cases and compare the defect results with those reported in [26]. We would also like to explore the *testability* of the industrial use cases from the perspective of automated test generation techniques such as those in [17]. Another direction for us to pursue involves extending our current analytics pipeline to extract relevant information from non-Interaction type of sentences in UCDs and relate it to the model extracted from Interaction sentences.

References

[1] V. Ambriola and V. Gervasi. On the systematic analysis of natural language requirements with circe. *Automated Software Engg.*, 13(1):107–167, 2006.

[2] R. K. Ando. Exploiting unannotated corpora for tagging and chunking. In *Proceedings of ACL-04*, 2004.

[3] A. Bertolino, A. Fantechi, S. Gnesi, G. Lami, and A. Maccari. Use case description of requirements for product lines. In *Proceedings of REPL 02*, pages 12–18, 2002.

[4] B. Boguraev and M. Neff. An annotation-based finite state system for UIMA: User documentation and grammar writing manual. Technical report, IBM T.J. Watson Research Center, Yorktown Heights, New York, 2007.

[5] B. Boguraev and M. Neff. Navigating through dense annotation spaces. In *Proceedings of the Sixth International Conference on Language Resources and Evaluation (LREC-2008)*. Marrakech, Morocco, may 2008.

[6] A. Cockburn. *Writing Effective Use Cases*. Addison-Wesley, Boston, MA, USA, 2000.

[7] R. Dale. Industry watch. *Natural Language Engineering*, 11(4):435–439, December 2005.

[8] A. H. Dutoit and B. Paech. Developing guidance and tool support for rationale-based use case specification, 2001.

[9] A. Fantechi, S. Gnesi, G. Lami, and A. Maccari. Application of linguistic techniques for use case analysis. In *RE 2002*, pages 157–164, 2002.

[10] D. Ferrucci and A. Lally. UIMA: an architectural approach to unstructured information processing in the corporate research environment. *Natural Language Engineering*, 10(4), 2004.

[11] G. Fliedl, C. Kop, H. C. Mayr, A. Salbrechter, J. Vöhringer, G. Weber, and C. Winkler. Deriving static and dynamic concepts from software requirements using sophisticated tagging. *Data Knowl. Eng.*, 61(3):433–448, 2007.

[12] V. Gervasi and B. Nuseibeh. Lightweight validation of natural language requirements. *Software Practice and Experience*, 32:113–133, 2002.

[13] G. Grefenstette. Light parsing as finite state filtering. pages 86–94, 1999.

[14] O. M. Group. Business process modeling notation version 1.1. *http://www.bpmn.org/Documents/BPMN1-1Specification.pdf.*

[15] J. Hammerton, M. Osborne, S. Armstrong, and W. Daelemans. Introduction to special issue on machine learning approaches to shallow parsing. *J. Mach. Learn. Res.*, 2:551–558, 2002.

[16] J. B. Jorgensen and C. Bossen. Executable use cases: Requirements for a pervasive health care system. *IEEE Softw.*, 21(2):34–41, 2004.

[17] M. Kaplan, T. Klinger, A. Paradkar, A. Sinha, C. Williams, and C. Yilmaz. Less is more: A minimalistic approach to uml model-based conformance test generation. In *ICST '08*, pages 82–91, 2008.

[18] C. Kennedy and B. Boguraev. Anaphora for everyone: Pronominal anaphora resolution without a parser. In *COLING*, pages 113–118, 1996.

[19] E. Ortner and B. Schienmann. Normative language approach - a framework for understanding. In *ER '96*, pages 261–276, 1996.

[20] S. P. Overmyer, B. Lavoie, and O. Rambow. Conceptual modeling through linguistic analysis using lida. In *ICSE*, pages 401–410, 2001.

[21] B. Paech. The four levels of use case description. *REFSQ*, 98, 1998.

[22] C. Rolland and C. B. Achour. Guiding the construction of textual use case specifications. *Data Knowl. Eng.*, 25(1-2):125–160, 1998.

[23] A. Sampaio, R. Chitchyan, A. Rashid, and P. Rayson. Ea-miner: a tool for automating aspect-oriented requirements identification. In *ASE '05*, pages 352–355, 2005.

[24] A. Sinha, M. Kaplan, A. Paradkar, and C. Williams. Requirements modeling and validation via bi-layer use case descriptions. In *MODELS'08*, 2008. To appear.

[25] M. Stickel and M. Tyson. Fastus: A cascaded finite-state transducer for extracting information from natural-language text. In *Finite-State Language Processing*, pages 383–406. MIT Press, 1997.

[26] F. Törner, M. Ivarsson, F. Pettersson, and P. Öhman. Defects in automotive use cases. In *ISESE '06*, pages 115–123, New York, NY, USA, 2006. ACM.

[27] J. Whittle and P. K. Jayaraman. Generating hierarchical state machines from use case charts. In *RE'06*, pages 16–25, 2006.

[28] W. M. Wilson, L. H. Rosenberg, and L. E. Hyatt. Automated analysis of requirement specifications. In *ICSE '97*, pages 161–171, 1997.

[29] T. Zhang, F. Damerau, and D. E. Johnson. Text chunking based on a generalization of Winnow. *Journal of Machine Learning Research*, 2:615–637, 2002.