Improved Processing of Textual Use Cases: Deriving Behavior Specifications

Jaroslav Drazan[1] Vladimir Mencl[1,2]

[1]Charles University, Faculty of Mathematics and Physics
Department of Software Engineering, Distributed Systems Research Group
Malostranske namesti 25, 118 00 Prague 1, Czech Republic
{drazan,mencl}@nenya.ms.mff.cuni.cz, http://nenya.ms.mff.cuni.cz/

[2]United Nations University
International Institute for Software Technology
mencl@iist.unu.edu, http://www.iist.unu.edu/

Abstract. The requirements for a system are often specified as textual use cases. Although they are written in natural language, the simple and uniform sentence structure used makes automated processing of use cases feasible. However, the numerous use case approaches vary in the permitted complexity and variations of sentence structure. Frequently, use cases are written in the form of compound sentences describing several actions. While there are methods for analyzing use cases following the very simple SVDPI (subject-verb-direct object ... indirect object) pattern, methods for more complex sentences are still needed. We propose a new method for processing textual requirements based on the scheme earlier described in [13]. The new method allows to process the commonly used complex sentence structures, obtaining more descriptive behavior specifications, which may be used to verify and validate requirements and to derive the initial design of the system.

1 Introduction

The requirements for a system are usually specified as textual use cases. Even though specified in natural language, the use case writing guidelines ask to use a limited subset of the natural language (English in the case we consider), making automated processing of textual use cases feasible. The motivation is either to check the use cases for adhering to style [15,20], check the requirements for consistency [5], or to leverage the information contained in the use cases to aid with designing the system [10,11,12], by deriving parts of its initial design from the use cases.

1.1 Textual Use Cases: Brief Overview

A use case specifies how a *System under Discussion (SuD)* will be used, in terms of all the possible scenarios of how SuD will interact with its surrounding *actors*.

This work was partially supported by the Grant Agency of the Czech Republic project 201/06/0770 and by the HighQSoftD project funded by the Macao Science and Technology Development Fund.

In a textual use case, the most typical flow is specified as its *main success scenario*, a sequence of steps. Each step should be a simple English sentence and describe an action initiated either by an actor or by SuD. The steps should describe only actions; any error handling and alternate flows should be specified in the extensions and variations of the use case (fig. 1). Importantly, the use case writing guidelines [2, 7] ask for a uniform sentence structure, which makes automated processing feasible.

1.2 Earlier Work

In [13], we proposed a method for deriving behavior specifications from textual use cases. Based on the simple and uniform sentence structure employed in use cases [2, 7], we established premises that (1) a single step of a use case describes a single action, which is either a request communicated between an actor and SuD, or an internal action of SuD, and (2) such an action is described by a simple English sentence.

Based on the fixed sentence structure, we had developed a small set of principles to obtain the principal attributes of the sentence (action type, actor, and event token) from the parse tree of the use case step description. We employed readily available natural language processing tools; the key component is a statistical parser [3], which allows us to obtain a phrase structure parse tree for a sentence (fig. 2). Nodes of the parse tree show how the sentence is composed of *phrases*; in each node, its *phrase tag* determines the type of the phrase – S for *sentence*, VP for *verb phrase*, NP (or NPB) for (*basic*) *noun phrase*. The leaves of the tree are the words of the sentence (preserving the left-to-right order); each word is labeled with its *part-of-speech* (POS) tag, determining both the word type and its grammar form used. In Fig. 2, VBZ stands for a verb in the z-form, NN for a noun, and NNP for a proper noun.

We briefly illustrate our original method on the parse tree of the sentence "Seller submits item description" (fig. 2). We start by identifying the *subject*, which is the leftmost noun phrase subordinate to the top-level sentence (S) node, and should be either an actor or "System" (in our example, the subject is the actor *Seller*). The first verb in the top-most verb phrase (VP) is the *principal verb* of the sentence, *submits* in this case. The noun-phrase below the VP node ("item description") is the *direct object* of the sentence. Next we decide the *action type*. As the subject of the sentence is an actor, we conclude that this sentence is a *receive action*. If the subject refers to SuD (e.g., via the keyword "System"), the action is a send action in case we identify an actor as the indirect object, or an internal action of SuD otherwise.

Use Case: #1 Seller submits an offer
SuD: Marketplace
Primary Actor: Seller
Supporting Actor:
Trade Commission
Main success scenario specification:
1. Seller submits item description.
2. System validates the description.

Extensions:
2a Item not valid.
2a1 Use case aborts.
Variations:
2b Price assessment available
2b1 System provides the seller with a price assessment.

Fig. 1: Fragment of use case *"Seller submits an offer"*

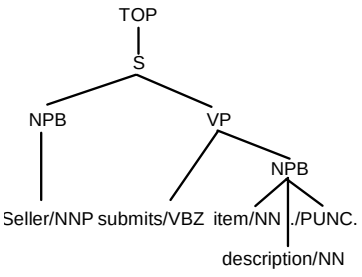

Fig. 2: Parse tree of the sentence: *"Seller submits item description."*

We finally represent this action with the token ?SL.submitItemDescription. The method is described in detail in [13] and [14], also featuring an elaborate case study.

1.3 Goals and Structure of the Paper

While our original method [13] is applicable to sentences following the guidelines of [2] and [7], there is a strong motivation to broaden its range of use. Industry use cases [9,19] often do not adhere to these guidelines. Often, a complex sentence expresses several actions and may also include a condition. Yet, even such sentences follow observable patterns. Abstracting from the common patterns and proposing rules applicable to a broad range of industrial use cases is the main focus of this paper.

For complex sentences, there is an increased risk that the statistical parser would return an incorrect parse tree. An additional goal is to propose a metric to evaluate parse tree quality and select the best parse tree if more then one is available.

The paper is structured as follows: in Sect. 2, we address the first goal, proposing an improved method to parse a complex sentence. In Sect. 3, we address the second goal by proposing a metric evaluating a parse tree. We evaluate our method in Sect. 4, discuss related work in Sect. 5, and draw a conclusion in Sect. 6.

2 Parsing Complex Use Case Steps

Our earlier work, as described in Sect. 1.2, has limitations to be addressed. In spite of the use case writing guidelines asking for a simple sentence form, in many cases, even in books respected as authority on object-oriented methodology [9,19], use case steps are often specified as complex or compound sentences, specifying several actions, and often also a condition. In this section, we first articulate the revised premises for our analysis, then we describe the information we aim to extract from a parse tree, and finally we describe the rules for processing a parse tree. We illustrate our method on sentences from sample use cases included in [9] and [19], both respected in industrial practice. Fig. 3 demonstrates a sentence starting with a condition, Fig. 4 shows a compound sentence consisting of two independent clauses, and Fig. 5 shows a sentence with two verb phrases, one of them featuring two indirect objects.

2.1 Use Case Sentence Structure.

We base our method on the following premises:

Premise 1 (content): A use case step starts with zero or more conditions and describes actions which are either (a) interaction between an actor and SuD, (b) internal actions or (c) special actions.

Premise 2 (compound sentence structure): A use case step is a simple or compound sentence (having several independent clauses). In a compound sentence, each independent clause must satisfy Premise 3 on its own,

Premise 3 (simple sentence structure): Each simple sentence (or independent clause of a compound sentence) is in present tense and active voice, and follows the SVDPI

pattern (*Subject ... verb ... direct object ... indirect object*). However, contrary to our earlier assumptions [13], (1) the sentence may contain more than one verb phrase and (2) each verb phrase may contain more than one direct and/or indirect object.

Note that we revise our premises to handle a broader category of use cases. The use cases our method originally dealt with satisfy also the new premises.

2.2 Meaning of a Use Case Step Sentence

In [13], we represented one use case step with a single action; its principal attributes were its type (send, receive, internal, or special), actor involved (for a send or receive action), predicate (main verb), and its representative object.

We now extend this model to handle the multiplicities considered. We represent a use case step with a sequence of *actions*, further called *action-set*. An action is determined by its *principal attributes*, which also consist of its *type*, *actor*, *predicate*, and *representative objects*. Contrary to our earlier work, an action can now contain zero or more representative objects. In case it contains more than one representative object, the action is a *compound action* (and its representative objects are a sequence of noun phrases). This additional information can be used, e.g., in a CASE tool in constructing an initial design of the target system. We also allow a use case sentence to have no representative object, to accommodate sentences like "User logs in".

The type of an action can be one of the following:
Send, ***receive*** or ***internal***: apart from the changed multiplicity of the representative object, these actions have the same meaning and principal attributes as in [13]. A send action represents a request sent by SuD to an actor. A receive action represents a request received by SuD from an actor. These are together called *operation request* actions and describe the request by the predicate (verb), the set of representative objects (corresponding to information passed in the request), and the actor involved. An internal action of SuD is described only by the predicate and the representative objects.
Special action: same as in [13], a special action changes the control flow of a use case, and can be either a *terminate* action ("Use case ends.") with no attributes, or a *goto* action ("Use case resumes with step 5.") with a single attribute *target step*.
Condition action: we introduce a condition action to represent the conditions our new method permits at the beginning of a use case step. The only attribute of a condition action is the parse tree of the condition (a subtree of the use case sentence parse tree).
Illegal action: we introduce this action to represent a sentence violating the use case sentence structure patterns; its only attribute *message* describes the violation detected.
Unknown action: this action represents a sentence where no known pattern (not even a violation) could be recognized, and is typically the result of a parser failure.

2.3 Analyzing a compound sentence

According to premise 1, a use case step is specified in a single sentence describing conditions and actions. We first identify all the conditions. In the case of a compound sentence, we split the sentence into its independent clauses. We follow by identifying all actions in either the simple sentence or each of the independent clauses.

Conditions. A condition is a sentence starting with a subordinate conjunction (IN) such as "if" or "when". We identify as conditions the left-most subordinate sentence node (SBAR) directly contained in the main sentence node (S) and its following SBAR nodes possibly separated by either punctuation nodes (PUNC) or coordinating conjunctions (CC); the clauses identified as conditions are not further considered in the analysis. In Fig. 3, the subtree starting in the SBAR node (*"If information changed includes credit card information,"*), is identified as the condition.

Compound sentences. A compound sentence consists of independent clauses, each describing one or more actions. Independent clauses are sentence nodes (S) contained within the top-level sentence node, joined with a coordinating conjunction (CC). We afterwards handle each independent clause separately, treating it as a simple sentence. In Fig. 4, the compound sentence *"Customer pays and System handles payment."* contains two independent clauses, "Customer pays" and "System handles payment".

2.4 Analyzing a simple sentence

When analyzing either a simple sentence or an independent clause of a compound sentence, we refer to the element analyzed as "sentence". We assume each sentence contains in its sentence node (S) two sub-nodes, a noun phrase (NP) describing the subject and a verb phrase (VP) describing one or more actions. If this structural constraint does not hold, we evaluate the sentence as an *unknown action* (unless it is a special action). The sentence node may also contain additional nodes, but all the information relevant for our analysis is contained within the NP and VP nodes.

Subject. We identify the subject in a way very close to our original method. According to the simple structure of English use case sentences, the subject has to be the first noun phrase (NP) node of the sentence. We look for (exact) match between the sequence of nouns (NN) and adjectives (JJ) from this noun phrase and the list of actors, extended with predefined keywords ("System", "User", "Use case", and "Extension"). The keyword "System" refers to SuD, "User" to the primary actor of the use case, and "Use case" and "Extension" identify a special action. In Fig. 4, the subject of the first independent clause is Customer, an actor of the use case, while the subject of the second independent clause is the keyword "System", referring to SuD.

Verb. A sentence can describe several actions, each specified through a separate predicate (verb). If the verb phrase (VP) of the sentence contains no verb (VB*) nodes, but directly

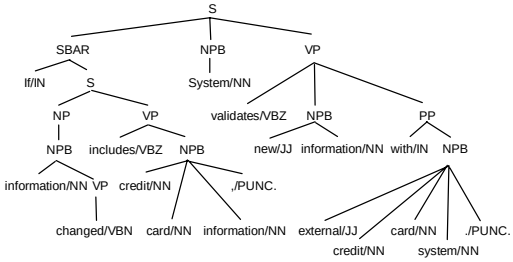

Fig. 3: Parse tree of the sentence: *"If information changed includes credit card information, system validates new information with external credit card system."* (step 8 of use case 2 in [19])

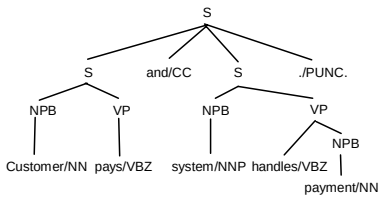

Fig. 4: Parse tree of the sentence: *"Customer pays and System handles payment."* (step 7 of use case 1 in [9]).

contains nested verb phrases, we process each of them separately as if it was a separate sentence (all with the same already identified subject). In Fig. 5, the top-level verb phrase has two nested verb phrases, both of them ("logs ... sale" and "sends sale ...") are processed separately.

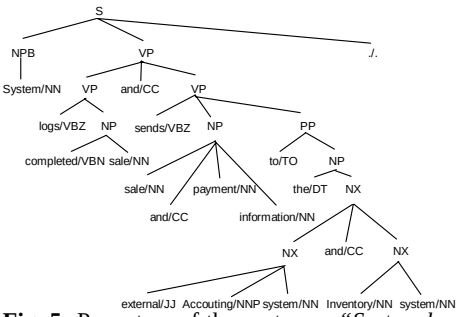

Fig. 5: Parse tree of the sentence: "*System logs completed sale and sends sale and payment information to the external Accounting system and Inventory system.*" ([9], step 8, use case 1)

The first verb in the verb phrase is the candidate to become the predicate, but we first check whether it is a *padding verb*, such as "asks" or "chooses" (based on a predefined list). If so, and if the current verb phrase directly contains an alternate verb phrase containing a verb (VB*), the *alternate* verb becomes the predicate, as it (and not the padding verb) describes the action to be performed. For example, in "*System asks the Supervisor to validate the seller.*" (CS1-5 in [17]), "asks" is a padding verb, while "validate" describes the request (sent to the actor Supervisor) and should be used in the event token.

Indirect objects. Indirect object is the recipient of the use case action, and should be an actor. We identify as indirect object a noun phrase nested within the (original) verb phrase node (i.e., the VP node containing the padding verb if there was one). We then aim to identify additional indirect objects. As multiple indirect object noun phrases are often grouped in a single noun-phrase node (such as in Fig. 5), we search for additional indirect objects only within the parent node of the first identified one. In Fig. 5, we identify indirect objects "Accounting system" and "Inventory system".

Identifying the indirect object is crucial for determining the type of the sentence, if the action is initiated by SuD. If no indirect object is found, the sentence is an internal action of SuD. When an indirect object matching an actor is found, the sentence is a send action addressed to the actor. When multiple indirect objects are found, the sentence is a sequence of actions, one for each actor.

Direct objects. Direct objects represent the data passed or processed in an action. A direct object is also a noun phrase; for direct objects, we however search only within the VP node containing the actual predicate used (excluding NP nodes already identified as indirect object). After finding the first direct object, we look for additional direct objects within its parent node, based on the same grouping patterns as for indirect objects. The direct objects are used as the representative object in the action (or each of the actions) determined by the verb and the indirect object. If multiple direct objects are found, the action is a compound action. In the parse tree in Fig. 7, we identify the direct objects "rebate form" and "rebate receipt".

2.5 Example

We now illustrate the principles of our analysis on the use case sentence "System logs completed sale and sends sale and payment information to the external Accounting and Inventory system" (parse tree in Fig. 5). We first detect that the sentence is a

simple sentence — its top-level sentence node (S) contains no S or SBAR subnodes, and contains a noun phrase (NP) followed by a verb phrase (VP). We identify the noun phrase as the subject, and it matches the keyword "System". As the verb phrase contains two nested verb phrases but no verb node, we process the two verb phrases separately. The first verb phrase contains one verb node (logs/VBZ) and one noun phrase describing a direct object, "sale". The second verb phrase consists of a verb node (sends/VBZ), of a noun phrase describing its direct object ("sale and payment information"), and of a preposition phrase describing two indirect objects ("Accounting system" and "Inventory system"), each matching an actor name. We finally construct the resulting action set, containing the internal action "#logSale" for the first verb phrase, and two send actions, "!AS.sendSalePaymentInformation" and "!IS.sendSalePaymentInformation", for the second verb phrase.

2.6 Additional Sentence Analysis Issues

Illegal actions. Throughout the analysis of a sentence, we look for violations of the use case writing guidelines. *Actor to actor communication*, such as in *"Cashier asks customer for a credit payment signature"* (step 7b6 of UC1 in [9]), violates Premise 1 — and should not occur in the use cases describing observable behavior of SuD. Further, use case writing guidelines strongly discourage use of *passive voice or modal or auxiliary verbs*. We evaluate a verb phrase with such violations as an illegal action; we however continue processing the remaining parts of the sentence to possibly detect additional violations, and we report the analysis results for the whole sentence.
Special Actions. A step may also describe a special action, changing the control flow of a use case. We identify a special action in the same way as in [13] — if the subject is "Use case" or "Extension", or the sentence has no subject at all. The verb must be a keyword, such as "terminate" or "abort" for a terminate action, or "continue" or "resume" for a goto action, where the target step is detected based on a pattern following the static sentence structure. A detailed description can be found in [13].

3 Metric: Selecting the Best Parse Tree

Industrial use cases we parse often contain compound or complex sentences. There is an increased risk that the statistical parser employed will return an incorrect parse tree for such a sentence. While this might be solved by employing several independent parsers, there is so far no mean to automatically select the correct parse tree from a collection of several parse trees. In this section, we introduce a metric to evaluate the parse trees, and a criterion to select the best parse tree based on this metric.

We base the metric on our observation that our method can in most cases extract more information from a correct parse tree of a sentence than from an incorrect one. We thus compute the action set for every parse tree of a sentence, use our metric to evaluate the complexity of each action set, and select the parse tree with the most complex action set. An action set consists of one or more actions, featuring principal attributes. The metric assigns to each action a score representing the complexity of its principal attributes, and the score of an action set is the sum of scores of all its

actions. Hence, as an incorrect parse tree will allow less information to be extracted, it will receive a lower score, and the metric will give preference to a correct parse tree over an incorrect one.

Metric definition. The score of an action set is the sum of scores of individual actions included in the set. The score of a single action depends on its type and its principal attributes.

The score of an **unknown** action is -1000; -100 is assigned to an **illegal** action, 1 to a **condition** action, 5 to a **terminate** action, and 6 to a **goto** action. We define the score of the direct (representative) objects of an action (*direct-objects-score*) as the number of words in all its direct objects + number of direct objects. The score of an **internal** action is 3 + direct-objects-score, score of a **send** action is 4 + direct-objects-score, and score of a **receive** action is 3 + direct-objects-score + number of words in its indirect object.

Motivation. The metric gives preference (positive score) to an action set containing no illegal or unknown action, but a very low negative score to unknown and illegal actions. Among these, illegal action is preferred, as it is typically obtained from a correct parse tree of an incorrect sentence. The metric prefers parse trees with more actions over parse trees with less actions of the same sum of complexity (each action gets additional score for its type). The metric prefers more indirect objects over less indirect objects with the same words (a new action is constructed for every indirect object) and finally, it prefers more direct objects over less direct objects with the same words (it adds the number of direct objects to the direct-objects-score). These rules follow the common parser failures of (1) joining several noun phrases into a single noun phrase and (2) turning an independent clause into a dependent one – while the inverse mistakes are rare.

Example. We illustrate the metric on two parse trees of the use case sentence "*System presents the rebate forms and rebate receipts for each item with the rebate.*" (step 9a1 of use case 1 in [9]). In Fig. 6, two indirect objects are incorrectly merged into a single noun phrase, while in Fig. 7, they are correctly parsed as separate noun phrases. We obtain the action sets [#presentRebateFormRebateReceipt] for the first parse tree and [#present<RebateForm,RebateReceipt>] for the second, both consisting of just one action, but with two direct objects in the second case. Consequently, the score of the second action set is by 1 higher (9 vs. 8), and the second parse tree is selected.

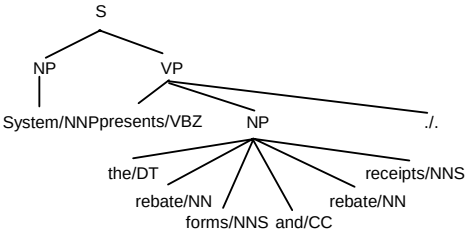

Fig. 6: Fragment of incorrect parse tree of sentence UC1-9a1, merging two direct objects into a single noun phrase.

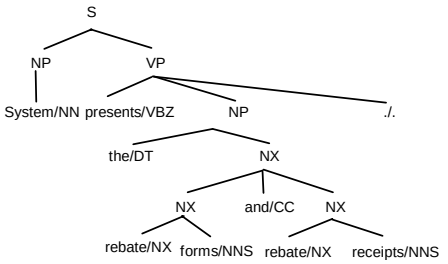

Fig. 7: Fragment of correct parse tree of sentence UC1-9a1, showing two direct objects.

4 Evaluation & Summary

Our method, as presented in this paper, has been substantially extended to accommodate complex sentence structures, widely used in the industrial practice. In particular, the new features of our method are: (1) support for compound sentences, (2) identification of conditions, (3) handling multiple verb phrases, (4) multiple indirect objects, and (5) multiple direct objects. When detecting the direct and indirect objects, our method (6) introduces a grouping rule to reduce the chances of a mismatch between a direct and indirect object. Further, our method (7) detects violations of use case writing rules, and (8) overcomes a single parser failure with a metric to select from several parse trees.

Note that although we do identify conditions in the parse tree, we do not aim to interpret the conditions in the derived behavior specifications. Instead, we represent the choice described by the condition with condition token, constructed as an estimate of how, e.g., a programmer would name a boolean variable representing the condition.

In our earlier work [13], we focused on deriving behavior specifications, such as Behavior Protocols [18] or UML State Machines. In this paper, we have presented an improved method for deriving the set of actions described by a single, more complex step. The algorithm for transforming the structure of a use case into a behavior specification now needs to be revisited, to leverage the additional structural information available in the actions obtained and also in a parse tree (such as conjunctions joining independent clauses or verb-phrases). However, also due to space constraints, we only present the method within the scope of a single use case step.

We have developed a prototype tool [4] implementing the improved method proposed in this paper . We have evaluated the method on a substantial collection of use cases (307 use case steps), consisting of examples from methodology sources [9, 19] respected by the industry, and also on the collection of sample use cases used in our earlier work [17]. Our new method has proven to be more reliable also on our original test data, where it has improved both in identifying the action type and actor, and in estimating the event token. Part of the improvement is due to the ability of our new method to select a correct parse tree for sentences where our original method failed, when the only parser used produced an incorrect parse tree. The detailed results obtained in our case study are available in the appendices of [4]. In this case study, we have identified the limitations of our method — overall, our method failed on 31 sentences, keeping our error rate at the level of 10%. Note that these errors include sentences where the sentence structure assumptions were violated, but our method was not able to detect this violation; for sentences adhering to the use case guidelines, the success rate is likely to be even higher.

5 Related Work

UCDA. Analysis of textual use cases is also addressed in the Use-Case driven Development Assistant (UCDA) [10, 11], which employs a parser and matches the parse trees with pre-defined sentence patterns, with the goal to derive collaboration diagrams and class diagrams from use cases. Although the basic principles used in

UCDA are similar to our approach, there are several significant differences. The UCDA method is built upon a simple shift-reduce parser, which inherently introduces stronger restriction on the sentence structures possibly used.

Instead of handling complex sentences directly, UCDA employs several *reconstruction patterns* to break a complex sentence into several simpler sentences processed independently. Unfortunately, [11] only lists the patterns without further elaboration. From the little information available, we assume that UCDA can handle compound sentences and multiple direct objects, but cannot handle neither multiple indirect objects, nor multiple verb phrases, such as in our example in Fig. 5.

We are convinced that our method, designed for statistical natural language parsers, can handle more variations in the parse tree structure, and is more applicable to industrial use cases. Also, as our method is independent of a single parser, our metric can further increase reliability by selecting from several available parse trees.

Requirements processing. In [12], textual requirements are analyzed with the goal to construct a knowledge base, consisting of terms, manually classified as a function, an entity, or an attribute. The knowledge base is later used to automatically generate design artifacts such as the object and data models. This method however targets a broad range of requirements, and does not utilize the specific properties of use cases, where it is possible to extract more information from the known sentence structure.

Controlled languages. An alternative approach to automatically processing natural language requirements specifications is to employ a precisely defined restricted subset of the natural language (English), a *controlled language*. Richards et al [21] propose a controlled language with simple rules on word forms and sentence structure, based on the principles of the broadly accepted use case writing guidelines. The RECOCASE tool [20] assists the writer in conforming to this controlled language. In [15], a controlled language and a rule-based parser are used to analyze NL requirements with the goal to assign Logical Form to requirement specifications; focus is put on resolving parsing errors and ambiguities. In the controlled language selection, it is pointed out that a too restrictive controlled language may be irritating and hard to use (and read). We have avoided using a controlled language in order to minimize the burden imposed on use case writers, allowing them to use English with no formal restrictions, assuming that the basic rules for writing use cases (such as using active voice, present tense) are preserved.

Evaluating use cases. Fantechi et al [5] focus on analyzing the quality of use cases, by identifying and quantitatively measuring specific defects, such as *vagueness*, *ambiguity* and *unexplanation*. They also employ linguistic tools and techniques, with the goal to check conformance to the Simplified English controlled language. In our work, we focus on analyzing the use cases, and we give use case writers more flexibility in the language they use. To avoid false alarms, we report only doubtless guideline violations.

Parser combining. The work of Henderson and Brill [8] also addresses the issue of obtaining the correct parse tree when several parse trees are available. The proposed technique, *parser combining*, is based on constituent voting and should produce a parser with a higher constituent match than any of the original parse trees. However, the resulting parse tree may not conform to the grammar of the respective language (English). Such parse trees are thus not suitable for our analysis method, where we rely on the parse tree conforming to the use case sentence patterns.

6 Conclusion & Future Work

We have presented an improved method for deriving behavior specifications from textual use cases, applicable to a significantly broader category of use cases, including use cases presented in the literature widely accepted in the industrial practice [9, 19].

In [13], we had presented a method for deriving behavior specifications from textual use cases, based on the simple sentence structure recommendations [2]. However, industry use cases often do not follow these guidelines precisely, and employ more complex sentence structures. The improved method presented in this paper allows to handle such use cases, by recognizing compound sentences, steps including a condition, and steps describing multiple actions, either through multiple verb phrases or multiple indirect objects, and also identifies multiple direct objects.

We have further enhanced the method to handle a possible parse failure by introducing a metric to select the best parse tree from several parse trees available.

Our method has several possible applications. The derived behavior specifications may be used to validate the use cases, by simulating the behavior of the future system. Further, after achieving a sufficient level of precision, the derived behavior specifications may be used to check the requirements model consistency, and may be also employed in a CASE tool for rapid prototyping. Here, multiple representative objects of an action, a new feature of our method, can be used for designing data structures in data modeling. In addition, our method now also allows to detect specific use case style violations; this feedback allows the use case writers to improve their use cases.

***Future work and open issues*:** We have developed a prototype implementation of the new method, made publicly available as a part of [4]. With our ongoing aim to employ the actions derived from use cases in rapid prototyping, a remaining challenge is to integrate the method into an existing CASE tool, where we may employ the interactive use case writing environment [6] implementing our earlier method. Further, there are several opportunities to extend the method itself. The coordinating conjunctions linking phrases or independent clauses (typically "AND" or "OR") convey additional information, which might be used in deriving the behavior specifications. Also, precision in distinguishing between direct and indirect objects might be improved by employing a parser providing more detailed relations among the verb and its objects such as [1], [22], and [23], or by using a valence dictionary.

References

[1] Blaheta D., Charniak, E., *Assigning Function Tags to Parsed Text*, Proceedings of the 1st Annual Meeting of the North American Chapter of Association for Computational Linguistic, pp. 234-240, Seattle, May 2000

[2] Cockburn, A.: *Writing Effective Use Cases*, Addison-Wesley Pub Co, ISBN: 0201702258, 1st edition, Jan 2000

[3] Collins, M.: *A New Statistical Parser Based on Bigram Lexical Dependencies*, in Proceedings of ACL 1996, University of California, Santa Cruz, CA, USA, 24-27 June 1996, pp. 184-19, Morgan Kaufmann Publishers

[4] Drazan, J.: *Natural Language Processing of Textual Use Cases*, Master's thesis, advisor: Vladimir Mencl, Charles University, Feb. 2006.

[5] Fantechi, A., Gnesi, S., Lami, G., Maccari, A.: *Application of Linguistic Techniques for Use Case Analysis*, in Proceedings of RE 2002, pp. 157-164, Sep 9-13, 2002, Essen, Germany. IEEE CS 2002
[6] Fiedler, M., Francu, J., Ondrusek, J., Plsek, A.: *Procasor Environment: Interactive Environment for Requirement Specification*, Student Software Project, supervisor: Mencl, V., http://nenya.ms.mff.cuni.cz/~mencl/procasor-env/, Charles University, Sep 2005.
[7] Graham, I.: *Object-Oriented Methods: Principles and Practice*, Addison-Wesley Pub Co, 3rd edition, Dec. 2000
[8] Henderson, J. C., Brill, E.: *Exploiting diversity in natural language processing: Combining parsers*, in Proceedings of the Fourth Conference on Empirical Methods in Natural Language Processing (EMNLP-99), pp. 187-194, June, 1999, College Park, Maryland, USA
[9] Larman, C.: *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*, Prentice Hall PTR, 2nd edition, 2001
[10] Liu, D., Subramaniam, K., Eberlein, A., Far, B. H.,: *Automating Transition from Use-Cases to Class Model*, IEEE Canadian Conference on Electrical and Computer Engineering (CCECE 2003), May, 2003
[11] Liu, D., Subramaniam, K., Eberlein, A., Far, B. H.,: *Natural Language Requirements Analysis and Class Model Generation Using UCDA*, in Proceedings of IEA/AIE 2004, pp. 295-304, LNCS 3029, Springer, May 2004
[12] MacDonell, S. G., Min, K., Connor, A. M., *Autonomous Requirements Specification Processing using Natural Language Processing*, Proceedings of the 14th International Conference on Adaptive Systems and Software Engineering (IASSE05), 2005
[13] Mencl, V.: *Deriving Behavior Specifications from Textual Use Cases*, in Proceedings of Workshop on Intelligent Technologies for Software Engineering (WITSE04, part of ASE 2004), Linz, Austria, Oesterreichische Computer Gesellschaft, Sep 2004.
[14] Mencl, V.: *Use Cases: Behavior Assembly, Behavior Composition and Reasoning*, Ph.D. Thesis, advisor: Frantisek Plasil, Jun 2004
[15] Osborne, M., MacNish, C. K. : *Processing Natural Language Software Requirement Specifications*, in Proceedings of ICRE'96, pp. 229-237, Apr 15 - 18, 1996, Colorado Springs, Colorado, USA, IEEE CS, 1996
[16] Plasil, F., Mencl, V.: *Getting "Whole Picture" Behavior in a Use Case Model*. Transactions of SDPS: Journal of Integrated Design and Process Science 7(4), pp. 63-79, Dec 2003
[17] Plasil, F., Mencl, V.: *Use Cases: Assembling "Whole Picture Behavior"*, TR 02/11, Dept. of Computer Science, University of New Hampshire, Durham, 2002
[18] Plasil F., Visnovsky, S.: *Behavior Protocols for Software Components*. IEEE Transactions on Software Engineering 28(11), Nov 2002
[19] Rational Software Corporation (IBM), *Rational Unified Process*, version 2003.06.01.06, 2003, http://www-130.ibm.com/developerworks/rational/products/rup/
[20] Richards, D.: *Merging individual conceptual models of requirements*, Requir. Eng. 8(4): 195-205, 2003
[21] Richards, D., Boettger, K., Aguilera, O.: *A Controlled Language to Assist Conversion of Use Case Descriptions into Concept Lattices*, Proceedings of AI 2002, Canberra, Australia, Dec 2-6, 2002, LNCS 2557 Springer 2002
[22] Schneider, G.: *Extracting and Using Trace-Free Functional Dependencies from the Penn Treebank to Reduce Parsing Complexity*, in Proceedings of Treebanks and Linguistic Theories (TLT) 2003,pp. 153-164, Växjö, Sweden, Växjö University Press 2003
[23] Stevenson, M.: *Extracting Syntactic Relations using Heuristics*, ESSLLI98 - Workshop on Automated Acquisition of Syntax and Parsing, pp. 248-256, 1998.