Converting Textual Use Cases into Behavior Specifications

[Technical Report 2004/5]

Vladimir Mencl

Charles University, Faculty of Mathematics and Physics
Department of Software Engineering, Distributed Systems Research Group
Malostranske namesti 25, 118 00 Prague 1, Czech Republic
mencl@nenya.ms.mff.cuni.cz, http://nenya.ms.mff.cuni.cz,
phone: +420 2 2191 4232, fax: +420 2 2191 4323

Abstract

Traditionally, natural language is used for writing use cases. While this makes use cases easily readable to users, it neither permits reasoning on requirement specifications (written as use cases), nor employing the use cases in deriving an initial design in an automated way. While employing linguistic tools to analyze use cases has already been considered, such attempts usually attempted to utilize all the information possibly contained in a use case specification, thus facing the complexity of natural language. Yet, in a use case, the sentence describing a use case step adheres to a simple prescribed structure, and describes an action, which is either a communication action (among entities involved in the use case), or an internal action.

In this paper, we describe how the principal attributes of the action described by a use case step can be acquired from the parse tree of the sentence specifying the step; we employ readily available linguistic tools for obtaining the parse tree. Having identified the communication actions permits us to construct an (estimate of) behavior specification of the entity modeled by a use case model (in the form of a Behavior Protocol [21]), as well as collect the list of operations accepted and requested by the entity; this may aid with defining the entity's interfaces in an initial design.

1. Introduction

1.1. Textual Use Cases

A use case describes how an entity (the *System under Design, SuD*) cooperates with other entities (*actors*) by communicating and performing internal actions to achieve a particular goal. Use cases are mainly used to specify the behavior of a future system as a part of the requirement specification. While different methodologies [10, 3, 14] propose slightly different approach to use cases, the common consensus [3, 13] is that a use case is usually specified as a sequence of steps forming the main success scenario specification, with extensions and alternatives specifying additional scenarios. Traditionally, natural language is used for describing the actions taken in a step. Here, the main motivation for using natural language is to make use cases readable and comprehendible to a wide audience, including users of the future system. Figure 1 shows a textual use case, following the template proposed in [3].

The most typical scenario of a use case is specified in the main success scenario specification (top-level block), where each step describes an action that contributes to achieving the goal of the use case; steps in a block are labeled with a step number. Alternative successful flows are specified in the sub-variations section; exceptional situations (error handling) are specified as extensions. Both variations and extensions are attached to a step (by its label), start with a condition (guard), followed by a sequence of steps forming a nested block;

This work was partially supported by the Grant Agency of the Czech Republic project 201/03/0911. The results will be applied in the OSMOSE/ITEA project.

here, step labels are prefixed by the extension/variation label. A nested block typically ends with a special action – either moving the flow to a step within the enclosing block, or causing the enclosing block to end. (In case no explicit special action is used, the flow implicitly resumes with the next step of the enclosing block). With special actions, extensions can be used to model repetition or skip a part of the enclosing block. Figure 1 demonstrates most of the features described here.

Besides laying out the structure of a use case, recommendations for writing use cases [3, 13] also provide guidelines imposing a simple and uniform structure for the sentences specifying the steps of the use case. Thus, although natural language (NLP) processing is a complex and difficult task in general, employing NLP tools to extract information from textual use cases may yield surprising results.

Use Case: #1 Seller submits an offer
Scope: Marketplace
SuD: Marketplace Information System
Level: Primary Task
Primary Actor: Seller
Supporting Actor: Trade Commission
Main success scenario specification:

1. Seller submits item description.
2. System validates the description.
3. Seller adjusts/enters price and enters contact and billing information.
4. System validates the seller's contact information.
5. System verifies the seller's history
6. System validates the whole offer with the Trade Commission
7. System lists the offer in published offers.
8. System responds with a uniquely identified authorization number.

Extensions:
2a Item not valid
 2a1 Use case aborts
5a Seller's history inappropriate
 5a1 Use case aborts
6a Trade commission rejects the offer
 6a1 Use case aborts
Sub-variations:
2b Price assessment available
 2b1 System provides the seller with a price assessment.

Figure 1: Textual Use Case "Seller submits an offer"

1.2. Linguistic Tools Available

Readily available linguistic tools permit to acquire a parse tree from a natural language sentence. A parse tree captures the structure of the sentence, according to the grammar of the respective natural language (English in this case). In a *phrase structure* parse tree, the leaves reflect the words of the sentence (preserving the left-to-right order), while intermediary nodes represent *phrases* constituting the structure of the sentence. Figure 2 shows a parse tree obtained for the sentence of step 1 of the use case "Seller submits an offer" shown in Fig. 1. There, the nouns "item" and "description" (together with ".") constitute a *noun phrase* (denoted NP; in this special case, the phrase is a *basic noun phrase*, denoted NPB), which together with the verb "submits" forms a *verb phrase* (VP). The parse tree also shows the headword of each phrase (e.g., the verb "submits" for the verb phrase). The numbers following the phrase label and the headword are the number of sub-nodes and the index of the sub-node containing the headword.

As a prerequisite to parsing, the part-of-speech (POS) of each word has to be determined (the word type and the role it plays in the phrase structure). There is a choice between several sets of POS tags; the linguistic tools we employ utilize a subset of the CLAWS tagset [2]. In Fig. 2, the POS-tag of "submits" is VBZ (verb in the "s" form), "item" and "description" are nouns (NN); "Seller" is a proper noun (NNP). A related task is obtaining the *lemma* (base form) of a word, based on its actual word form and the POS tag. E.g., the lemma of "submits/VBZ" is "submit" (shown in Fig. 2).

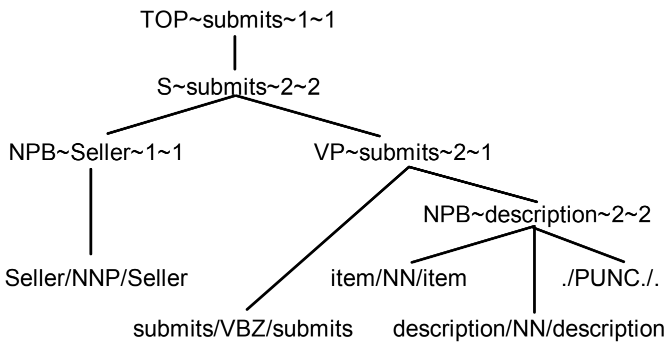

Figure 2: Parse tree of the sentence*:*
"Seller submits item description"

1.3. Goals of the Paper

In [22], we described transformation of textual use cases into Pro-cases, a notation for use cases based on the formal specification method Behavior Protocols [21]. We proposed guidelines for performing this conversion by hand. In this paper, we describe how readily available tools for natural language processing can be employed to accomplish this task in an automated way. The key goal is to describe how the simple and uniform structure of sentences used to describe steps of a use case can be utilized to extract the principal attributes of the action described by the sentence from its parse tree. We achieve this task with only a minimal domain model, consisting only of names of the entities modeled by use cases and optionally of names of conceptual objects.

Subsequently, we explore the options of converting a textual use case into a behavior specification. We demonstrate how a textual use case can be transformed into a behavior protocol, specifying the communication and internal actions of the entity described by a use case model.

The paper is structured as follows: Section 2 describes how to identify the action described by a single step in a use case specification from the parse trees of the sentence describing the step. In Sect. 3, we explore employing these results in the conversion of a use case into a behavior specification. We evaluate our approach and discuss open issues in Sect. 4, analyze related work in Sect. 5; we conclude the paper and outline future work in Sect. 6.

2. Analyzing a Use Case Step

The guidelines for writing use cases prescribe a uniform sentence structure. Readily available linguistic tools allow to obtain a parse tree for such a sentence, showing how the sentence is composed of phrases, types of the phrases, headwords and POS-tags of words (leaves of the tree).

In this section, we show how the type of the action described by a step can be obtained from the parse tree, as well as the principal attributes of the action. We show how this information can be obtained from the verb, the subject and the direct and indirect object of the sentence (employing the information from the domain model).

We demonstrate our conversion on a use case model describing several entities involved in an electronic marketplace. This use case model has been already used in [22]; the complete set of use cases is available in [23]. Figure 1 shows the first use case of this model (with step 1 slightly rephrased for demonstration purposes). Figure 3 shows the scope diagram [3, 22] of the Marketplace system. The domain model we used in our analysis contains entity names: "Seller", "Buyer", "Clerk", "Supervisor", "Credit Verification Agency", "Computer System", "Marketplace Information System", "Martketplace" and "Trade Commission"; the list of conceptual objects is: "item", "offer", "price assessment", "price" and "payment method".

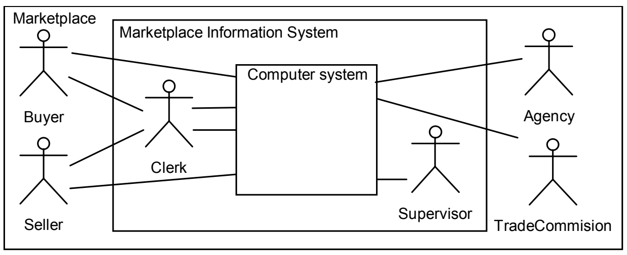

Figure 3: Scope diagram of the Marketplace system

2.1. Use Case Sentence Structure Premises

Based on the guidelines [3, 10, 13] for writing use cases, we derive the following premises, forming the basis of the conversion described in this paper.

Premise 1: An action described by a step of a textual use case describes either (a) communication between an actor and SuD (a request being sent or information passed), or (b) an internal action performed.

Premise 2: Such action is described by a simple English sentence, adhering to a uniform structure pattern.

Support for these premises can be found in [3, 10], similar recommendations are also found in recent research [6, 8, 9, 12, 15, 20, 25] on natural language use case processing.

In particular, premise 1 is supported by [3], pp. 90-97, where it is required that a step describes *"a simple action in which one actor accomplishes a task or passes information to another actor."*. Further, it is asserted that each step clearly indicates which actor performs the action; the actor has to be the subject of the sentence. In a similar vein, [10] pp. 410 requires that a step describes a *"single atomic task"*.

For premise 2 we find even more evidence. [3] specifically requires that *"The sentence structure should be absurdly simple: Subject ... verb ... direct object ... prepositional phrase."*. [10] explicitly defines task action grammar *SVDPI*, imposing sentence structure *"Subject ... verb ... direct object(s) ... preposition ... indirect object(s)"*. Such sentence structure is also prescribed by the controlled language proposed in [25].

2.2. Linguistic Tools Used

We employ the well-accepted statistical parser [4, 5] developed by M. Collins at the University of Pennsylvania. While there are several approaches to parsing (rule-based parser, statistical parser), we choose the Collins parser for its high accuracy (over 88% constituent accuracy and over 90% accuracy on dependencies – on generic English text) and for the robustness of its parsing algorithm; also, the Collins parser is using the well-established PennTreebank notation.

We have also considered the statistical parser developed by E. Charniak at the Brown University [1], which also outputs phrase structure trees (but, unfortunately, does not identify phrase heads), and the principle-based parser MINIPAR [16] developed by D. Lin at the University of Alberta (employing a proprietary format).

We use the MXPost tagger [24] for assigning POS-tags (note that with other natural language processing systems, this may be a part of the parser [1]). In addition, we also employ the Morphological tools [18] developed at the University of Sussex to obtain the *lemma* (base form) of the words.

2.3. Action Type & Communication Information

Step types. From premise 1, we conclude that a step of a use case specifies an operation to be performed, either an *internal action* of an entity, or processing a *request*, either *received* by SuD from an actor, or *sent* by SuD to an actor. Providing support for this conclusion, the well-accepted object oriented methodologies [14] recommend deriving lists of operations of conceptual objects and system sequence diagrams from (textual) use cases.

Based on premise 2, we assume that the sentence describing a step of a use case adheres to the SVDPI pattern described in Sect. 2.1. In such a sentence, subject is the entity performing the action; the verb describes the action. Further, the *direct object* of the sentence describes the data being passed. Moreover, for a request, the *indirect object* of the sentence is the entity receiving the request (if specified). Exceptions to this rule are discussed in Sect. 2.5.

The first issue in analyzing the sentence describing a step is to determine the action type. Here, the key lead is the subject of the sentence; in certain cases, the verb is also used.

Subject. Supposing the sentence adheres to the prescribed structure and was successfully parsed by the parser, the main sentence node ("S") of the parse tree contains a *noun-phrase* node and a *verb-phrase* node (in this order). We assume that the first noun-phrase (which may consist of several words) is the subject of the sentence. We match the sequence of nouns ("NN*") in this phrase with the names of entities involved as Actors in the particular use case and with a set of predefined keywords – phrases with special meaning, typically used in textual use cases. The term "System" is frequently used to refer to SuD; in a similar way "User" may be used to refer to the primary actor of the use case (the "Primary Actor" header may designate an entity as the primary actor of the use case). Further, the keywords "Use case" and "Extension" are recognized; these keywords indicate that the step describes a *special action*; such a step will be handled in a special way (discussed in Sect. 2.5). A sentence with no subject is also treated as a special action, e.g., "Resume with step 6".

Thus, finding a match for the subject noun-phrase yields the entity active in the step – either the subject is directly the name of an entity, or it is a keyword that in the context of the use case refers to a concrete entity. Note that not having found a match suggests that the sentence was badly written, or, possibly, that the statistical parser failed on the (otherwise correct) sentence; these issues are discussed in Sections 2.7 and 4.

In case the entity referred to by the subject is an actor involved in the use case, the step describes a request sent by the entity to SuD. Thus, from the point of view of SuD, the step describes receiving a request from the actor.

Example. In the parse tree in Fig. 2, the subject is the name of the entity "Seller"; thus the sentence describes a receive action.

Indirect object. In case the subject of the sentence refers to SuD, the step describes either a request issued by SuD toward an actor, or an internal action of SuD. In further analysis, we will look whether the name of an actor is an indirect object of the sentence. For simplicity, we consider as indirect object a noun-phrase subordinate to the main verb-phrase that matches the name of an actor (or the "User" keyword). We consider certain special cases and exceptions, e.g., the actor name must not be in the possessive case, i.e., directly followed by the POS element (marked by an apostrophe). In case the indirect object of the sentence is an actor, the step describes a request sent by SuD to this actor; otherwise, the step describes an internal action.

Example. In Fig. 4, the subject refers to SuD (keyword "System") and the indirect object refers to an entity ("Supervisor"); thus, the sentence describes a send action. The step 2 of the use case in Fig. 1 ("System validates the description.") describes an internal action.

When matching the subject and indirect object noun-phrases with entity names, we consider only SuD and the actors declared in the use case header. We choose this approach for simplicity and manageability; alternatively, corrections to use case headers might be proposed when a noun-phrase matches the name of an entity not declared in the use case header.

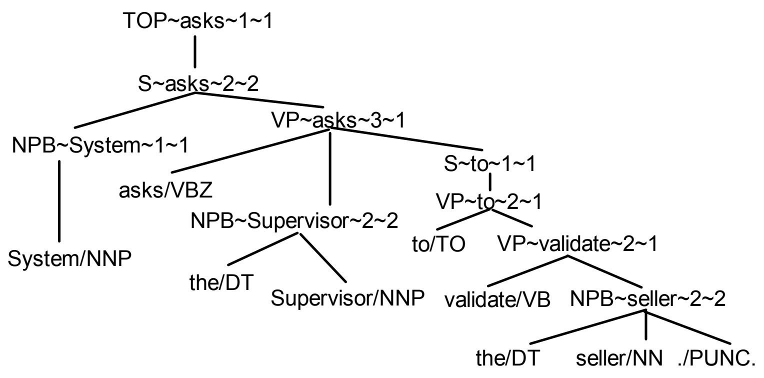

Figure 4: Parse tree of *"System asks the Supervisor to validate the seller."* (step 5 of use case CS1).

2.4. Estimating Method Call

After determining the type of the action, direction of communication and the actor involved, the next goal is to construct an event token to represent the request in a machine processable behavior specification. Adhering to the widely accepted recommendations and practice to construct method names from verb and object of the sentence (as in system sequence diagrams in [14]), we construct the token from the *principal verb* and the *representative object* description.

Verb. The principal verb is the headword of the topmost verb phrase, unless it is a *padding verb*. In this case, we skip the padding verb and use the next verb in the phrase (possibly nested in a subordinate verb-phrase). For example, let us consider the following sentence: *"System asks the Supervisor to validate the seller."*(step 5 of use case CS1 in [23]); the parse tree is shown in Fig. 4.

This sentence describes a request sent by SuD (keyword "System") to the actor Supervisor. Here, the verb "asks" is only a syntactical construct, and the verb "validate" actually describes the task requested. We have assembled a list of patterns identifying padding verbs; the list contains word sequences: "ask", "be", "select to", "choose to". Note that we compare the words in a pattern by the base form (lemma); matching POS-tag is also required. Also note that we do not consider multiple padding verbs used sequentially in a sentence, as such constructs are not used in practice – and such a sentence structure would clearly be a violation of the use case writing guidelines. On a technical note, we do not apply the padding-verb rule if there are no subsequent verbs in the sentence.

Example. In Fig. 2, the verb "submits" is the principal verb of the sentence; for further use, we consider the base form (lemma) of the verb, i.e., "submit". In Fig. 4, "asks" is a padding verb and "validate" is the principal verb (already in base form).

Direct Object. The motivation for acquiring the representative object of the sentence is to obtain words describing the data passed in the request, to be subsequently used in estimating the event token. We obtain the direct object noun phrase of the sentence, and use the relevant words from this noun-phrase. We consider as the direct object noun phrase the first basic noun phrases subordinate to the principal verb; we avoid phrases already identified as the subject and the indirect object. (Technically, we skip a noun phrase if any of its constituent words has already been used – as the subject, the verb or the indirect object. As a fall-back rule when no direct object is found, we try all noun phrases of the sentence).

In this process, we employ the list of conceptual objects (a part of the domain model). In case the direct object noun phrase contains the name of a conceptual object, all words from the noun phrase that are a part of a matching name of a conceptual object are used as the representative object description; otherwise, we use the sequence of all nouns in the direct object noun phrase.

Example: In Fig. 2 ("Seller submits item description."), the direct object noun phrase contains nouns "item" and "description". As the list of conceptual objects (part of the domain model) contains the word "item", the representative object description selected is "item".

Event token. After the principal verb and the representative object description is identified, we construct the event token; the event token is created from the principal verb and the representative object name. Technically, we concatenate base forms of these words, employing a naming convention to use the first word (the verb) in lowercase, and capitalize the first letter of each subsequent word.

Example: For the sentence in Fig. 4, the verb is "validate" and the representative object is "Seller", thus, the resulting token is **`validateSeller`**; in a similar way, for Fig. 2, with verb "submit" and representative object noun "item", we get the event token **`submitItem`**.

2.5. Special Actions

While most steps in a use case describe communication or internal actions, certain steps are an exception to this rule.

Typically used in extensions of a textual use case, some steps describe "meta-actions" changing the flow of the use case. Commonly used patterns of such sentences are "Use case aborts" or "Resume with step <label>"; such patterns are also recommended by the guidelines for writing use cases [3]. The key point here is that such sentences have a very simple form and are a nearly-constant syntactic construct. In particular, the subject of such sentences is either a predefined keyword (such as "Use case"), or the sentence has no subject at all; moreover, even the verb of the sentence is only one of a few predefined verbs.

We have identified two kinds of special actions, *terminate* actions and *goto* actions. Both are used only in use case extensions and variations and change the flow of control in the *enclosing block*, the block containing the step to which the extension is attached (the main success scenario specification for first-level extensions). By convention, a special action is used as the last step of a use case extension (and only there). A terminate action causes the enclosing block to terminate, while a goto action transfers control to a particular step, identified by a step label.

Terminate. A terminate action terminates the flow in the enclosing block. When the keyword "Use-case" is used as the subject, the action is a *propagating* terminate action and terminates the whole use case; otherwise (when the keyword "Extension" is used), the action terminates only the enclosing block. This distinction applies only when multiple levels of extensions are used; e.g., a terminate action used as the last step of a second-level extension terminates the first-level extension and flow continues with the next step of the main success scenario specification. A terminate action may be an *aborting* action, capturing a failure. In Fig. 1, the sentence "Use case aborts." used in steps 2a1, 5a1 and 6a1 terminates the flow of the use case.

We identify a terminate action when the verb matches on of the predefined patterns: "abort", "end" and "terminate". Moreover, the pattern "aborts" also indicates that the terminate action is an aborting action (this may be used in subsequent processing of the extracted information).

Goto. A goto action transfers control to a particular step in the enclosing block. The sentence describing a goto action follows a very simple pattern, solely declaring the transfer of control to a particular step. Such a sentence either has no subject, or the subject is one of the keywords triggering a special action: "Use case" or "Extension". The verb is one of the following keywords: "continue", "repeat", "resume" and "retry" and is followed by a reference to the target step. In this pattern, there is a noun-phrase subordinate to the main verb-phrase (possibly via a prepositional phrase as in Fig. 5). The noun-phrase has a "step/NN" and a "CD" (cardinal digit) node; the value of the CD

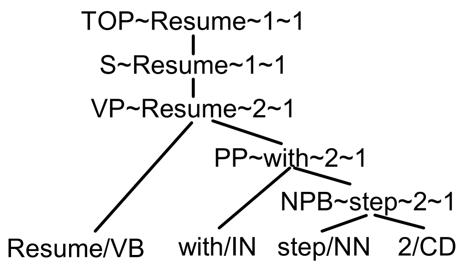

Figure 5: Parse tree of sentence: *"Resume with step 2"*

node is the label of the target step of the goto action. Note that semantics of a goto action is not influenced by the subject of the sentence.

As mentioned earlier, special actions are used as the last step of an extension. In case an extension does not end with a special action, we assume an implicit goto action to the step immediately following the step to which the extension is attached; this is common in use case practice (e.g., variation 2b in Fig. 1). The use case writing guidelines [3], (p 108) say that *"Usually, it is obvious" [where the story goes].* In rare situations, the flow should continue with the step to which the extension is attached, e.g., to retry an action that failed. While it may be obvious to a human reader from the steps of the extension, our approach requires explicitly adding a special action step – which will only improve clarity of the use case.

2.6. Conditions of Extension and Variations

The specification of an extension of a use case starts with the extension condition – a textual description of the situation in which the extension applies. We do not aspire to interpret conditions expressed in natural language. While there are clear recommendations for sentence structure of steps of a use case, little can be said about how the extension conditions are specified. In the guidelines [3], the sole recommendation is that *"Condition should describe what the system detects (not what happened)."*; the examples given are "Invalid PIN", "Network is down", "The customer does not respond (time-out)".

Although we do not intend to interpret the conditions, we extract descriptive information from the condition specification to construct a *condition label* that may be used to represent the condition in a behavior specification, and provide a single explanatory identifier for the condition, named in a way a programmer would name, e.g., an attribute representing the condition.

Contrary to our approach to processing use case steps, in this process, we only remove words that do not influence the meaning of the condition specification: prepositions (IN) "of", with", determiners (DT) "any", "the", punctuation (PUNC), the infinitival "to" (TO). However, we preserve words that are essential in meaning of the condition: adverbs (RB) "not", "no", "too" and determiner "no". We also remove padding verbs as described in Sect. 2.4. Further, we identify basic noun phrases (NPB) that contain a name of a conceptual object (as in identifying the direct object in Sect. 2.4); for such phrases, we use only the words matched, and remove other words of the phrase. Finally, we join all the remaining words into a single label; to preserve the meaning of the condition, we use the actual word forms used instead of the base form (lemma).

Example. Let us consider the condition M3-2a "Order is not valid", with the parse tree shown in Fig. 6. The subject noun-phrase matches the conceptual object "Offer"; the verb "is" is a padding verb and is removed, the adverb "not" and adjective "valid" are used, yielding the condition label **`offerNotValid`**.

Note that currently, we aim to translate to behavior specification mechanisms that either do not support conditions, or that model a condition as a black-box internal (condition) event. Thus, we construct the condition label only to play an informative role.

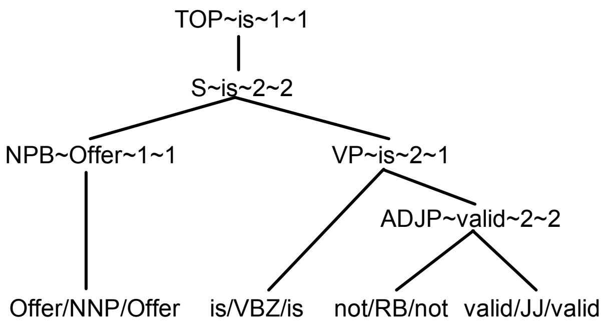

Figure 6: Parse tree of condition *"Offer not valid"*

2.7. Analyzing Use Case Steps: Summary

In this section, we have demonstrated that by employing the parse tree of the sentence specifying a step in a textual use case specification, it is possible to obtain the type and principal attributes of the action described by the sentence. The action may be an internal action, a communication action (where we also obtain the communicating actor), or a special action. For internal and communication actions, we construct an event token, representing an estimate of a future method name; for a special action, we obtain the type and the target (in case of a goto action). For extension conditions, we construct a condition label as a descriptive representation of the condition. Figure 7 shows the automatically obtained event tokens (left), compared to previously manually created tokens (right). The event tokens match on action type and actor identified (except for step 8, relying on context) and provide similar estimates of event tokens.

Discussion on failures. When one of the parts of a sentence is not found (subject, verb, indirect object, direct object), the cause may be either improper structuring of the sentence (too complex, not adhering to the requirement for simple structure), or the statistical parser may have failed at the particular sentence; this is more likely for complex sentences. Adhering to the writing guidelines (which also ask for simple sentences) leads to improved success rate of our tool; this is later discussed in the evaluation in Sect. 4.

1	?SL.submitItem
2	#validateDescription
3	?SL.adjustPrice
4	#validateSeller
5	#verifySeller
6	!TC.validateOffer
7	#listOffer
8	#respondAuthorizationNumber
2a1	%ABORT
2b1	!SL.providePriceAssessment
5a1	%ABORT
6a1	%ABORT

1.	?sic.submitItem	
2.	τValidateItem	
3.	?sic.submitPrice	
4.	τValidateSeller	
5.	τVerifySellerHistory	
6.	!tradecom.validate	
7.	τListOffer	
8.	!sellernotify.putAuthNr	
Extensions & sub-variations:		
2a1	Null	//Abort
5a1	Null	//Abort
6a1	Null	//Abort
2b1	!sellernotify.putPrice-Assessment	

Figure 7: Action types, attributes and event tokens automatically obtained for the use case "Seller submits an offer" (left) and created by hand in [22] (right)

3. Converting Textual Use Cases to Pro-cases

In the previous section, we described obtaining the type and principal attributes of actions specified by use case steps. Here, we describe the second step in converting a use case to a behavior specification. From the wide range of formal techniques (possibly ranging from state machines with transitions labeled by method calls as used in UML [19], to process algebras), we choose Pro-cases [22], based on Behavior Protocols (BP) [21] for the following reasons: (i) same as for use cases, BP are also designed for high readability, (ii) behavior protocols also specify behavior in terms of events sent, received and internally processed (matching our interpretation of use cases) (iii) behavior protocols are designed to specify behavior of software components (matching the use of use cases for nested entities).

Of course, as the event tokens obtained are only an estimate of a future method name, and as we are employing a statistical natural language parser, the resulting behavior specification will be inherently imprecise and only is an estimate of the actual behavior specification, but, regardless that, can be of high value to developers.

3.1. Use Cases vs. Pro-cases: Structure

Pro-cases (Behavior protocols) specify behavior in terms of atomic events emitted (!), absorbed (?) and internally processed (τ); the syntax stems from regular expressions. A primary expression specifies a single event; among the operators are `;` for sequencing, `+` for alternative and `*` for repetition (and additional operators not used here). Figures 9 and 10 both show a Pro-case (note that in the automatically constructed Pro-case in Fig. 9, "#" is used instead of τ).

Conveniently, the behavior of a Pro-case (a set of finite sequences of events) forms a regular language. This permits reasoning on Pro-case specifications; the compliance relation is decidable and a verifier tool is available [17]. Note that conditions are not supported in Pro-cases, this is actually a trade-off for the regularity of the language. Thus, when converting use cases to Pro-cases, we cannot directly represent conditions of extensions and variations; instead conditions are represented as a *condition event* – a special internal event. The semantics of condition event is, that it may be processed "only when the condition occurs"; in a particular run, once the condition event is processed, it may be processed any number of times. Note that internal events are not considered in the compliance relation (and thus, neither is the condition event).

3.2. Creating a Pro-case

We convert the structure of use case into a Pro-case by first constructing a finite automaton representing the use case, where transitions are labeled with the events as identified in Sect. 2; afterwards, we derive a regular expression generating the same language as the automaton.

Constructing finite automaton. We start by creating an initial and final state of the automaton, after which we process the main success scenario specification (the top-level block). To correctly capture the flow of a use case (and to avoid introducing new unexpected scenarios), we create a *pre-i* and *post-i* state for each step *i* in a block containing *n* steps; for each block, we also create a *block final* state. Next, we process the top-level block according to the following rules (for simplicity, we also access the initial state as POST-0 and the block final state as PRE-(n+1)):

1. For each step *i* ($1 \leq i \leq n$), we add a *lambda transition* (no event processed) from POST-(i-1) to PRE-i
2. We add a lambda transition from the block final state to the final state.
3. For each step *i* not describing a special action, we add a transition from PRE-*i* to POST-*i* labeled with the event representing step *i*.
4. For each step *i* describing a goto special action, we add a lambda transition from PRE-i to the PRE-state of the target of the step *i.*
5. For each step *i* describing a terminate special action, we add a lambda transition from PRE-*i* to the final state of the enclosing block.

   In case of a propagating terminate action, we process all containing blocks of the enclosing block: starting with the enclosing block and until we reach the top-level block, we add a transition from the currently reached block's final state to its containing block's final state and proceed with the containing block; all these propagation transitions are labeled with the condition event of the innermost extension containing step *i*.
6. For each extension/variation *e* attached to step *i*, we create an *extension initial* state e_{init}, and add a transition from POST-*i* to e_{init} labeled with the condition event of *e*. We consider e_{init} and PRE-(*i*+1) as the initial and final state respective and process the extension block according to the rules 1-6.

We illustrate the construction in Fig. 8; the parts shown correspond to a fragment of the main success scenario and the extensions 2a and 2b. The states "2a" and "2b" are the extension initial states; the *_FINAL states are the final states of the respective blocks.

Deriving a regular expression. We use the generic algorithm for deriving a regular expression (RE) from a generalized nondeterministic finite automaton (GNFA) described, e.g., in [26]. Very briefly, under certain assumptions which hold in our case (no incoming transitions to the initial state and no outgoing transitions from a single outgoing state), the algorithm prescribes that until there is only an initial and a final state, one of the other remaining states (q_r) is removed. In each such step, for every pair of states q_i and q_j (different from q_r), if there is a transition from q_i to q_r and from q_r to q_j, we replace the label of transition from q_i to q_j with the regular expression $e(q_i,q_j) + e(q_i,q_r)\ ;\ e(q_r,q_r)^*\ ;\ e(q_r,q_j)$, where $e(q_k,q_l)$ is the label of the transition from q_k to q_l if there is such, or $\varnothing$ otherwise. At the end, the label of the only transition from the initial to the final state is the RE sought.

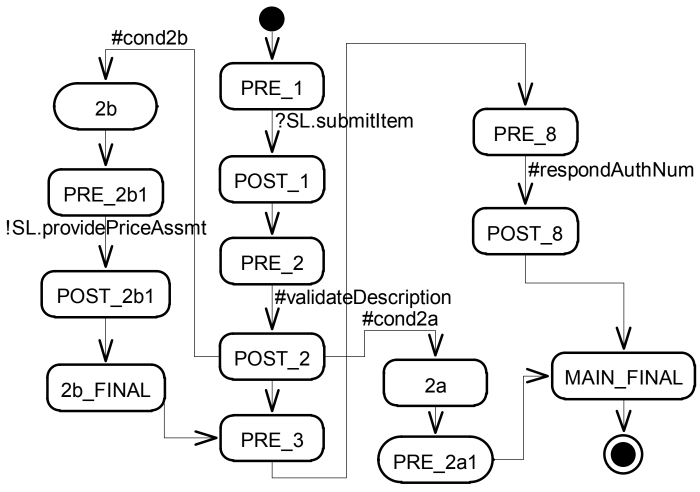

Figure 8: Part of the automaton constructed in the conversion of the use case "Seller submits an offer" to a Pro-case

Here, the only remaining issue is deciding on the order in which states are removed. The order may significantly influence the form (and length) of the resulting RE (but not the language generated by the RE). With the goal to minimize redundancy in the resulting RE (and thus, enhance readability and manageability), we developed criteria for selecting the state to be removed: for each state, we evaluate redundancy introduced by removing the state; in each step, we remove the state introducing the least redundancy. Enhancing the criteria to efficiently and reliably achieve the least redundancy in the resulting Pro-case is subject of future research. Figure 9 shows the Pro-case automatically obtained for the use case in Fig. 1 with our prototype tool.

```
?SL.submitItem ; #validateDescription ;
(  #cond2a
+  ( NULL + #cond2b ; !SL.providePriceAssessment )
   ; ?SL.adjustPrice ; #validateSeller ; #verifySeller ;
   (  #cond5a
   +  !TC.validateOffer ;
      (  #cond6a
      +  #listOffer ; #respondAuthorizationNumber )
   )
)
```

Figure 9: Automatically created Pro-case for the use case "Seller submits an offer"

```
?sic.submitItem    { τValidateItem ; ( NULL +
   τPriceAssessmentAvailable ;
   !sellernotify.putPriceAssessment + τInvalidItem ) } ;
(  ?sic.submitPrice { τValidateSeller ;
      τVerifySellerHistory ; ( !tradecom.validate ; (
      τListOffer ; !sellernotify.putAuthNr +
      τTradeComValidateFailed ) + τVerifyFailed )
   } + τInvalidItem
)
```

Figure 10: Pro-case manually created for the use case "Seller submits an offer"

4. Evaluation & Open Issues

Evaluation. We have implemented the conversion described here in a prototype tool and evaluated our approach in a case study on a set of use cases published in [23], developed within our previous work [22] proposing manual conversion of textual use cases to Pro-cases. With only a minor modification, the use cases were understood by the tool, yielding an estimate of the behavior specification of the subsystems described, as well as the list of events, which may be used in designing the interfaces to the subsystem in the initial design stage.

In [22], we have demonstrated the conversion on the use case "Seller submits an offer" (shown also here in Fig. 1, with only minor modifications). Figure 7 shows the event tokens obtained by our tool (left) and originally created by hand (right) in [22]. Further, in Fig. 9, we show the Pro-case obtained with our tool, while Fig. 10 shows the manually created Pro-case ([22]). Currently, the prototype implementation does not create the condition labels yet; we use the extension label instead (e.g., "#cond2a"). Also, nested calls are not detected (expressed with curly braces in Fig. 10, but not present in Fig. 9).

The modifications of the use cases done were mostly typo corrections. Several sentences in the original use case model did not adhere to the use case writing guidelines (use active voice, write short unambiguous sentences), and had to be corrected in order to produce correct results with the linguistic tools and our prototype tool (simplify sentences, change to active voice).

We intend to perform tests on an industrial use case specification; however, such specifications are usually considered as confidential and are hard to obtain.

Lessons for use case writers. From the case study already done, we have learned that for textual use cases to be machine-processable, the generally accepted use case writing guidelines have to be adhered to. In particular, the sentences have to be simple, describing only communication between SuD and an actor (or an internal action), and use of synonyms has to be avoided. In addition, it is necessary to avoid relying on context (from previous steps), even where the context would be obvious to a human reader.

Thus, use case writers have to learn to write machine-processable textual use cases, but doing so will result into more readable, clear and less ambiguous use cases.

Open issues. While our tool provides satisfactory preliminary results, certain issues remain open. In particular, enhancing the construction of event tokens to yield the same event token for complementary send / receive actions in use case models of communicating entities. Also, at least minimal support for (pre-declared) synonyms would be helpful.

The current implementation does not support including another use case. While this can be done as "macro substitution" (upon the automata), the issue of how to properly handle a failure of the included use case remains open. This triggers another open issue: how to handle failure in Pro-cases; extensions to the underlying behavior protocols specification mechanism are being investigated.

As use cases have a primarily informative goal, the execution semantics of use cases is given only by general consensus. Thus, another open issue is whether multiple extensions attached to the same step may be executed in a sequence; additionally, it is not clear how to identify an extension/variation that is executed *instead* (and not after) the action of the step.

5. Related Work

Controlled languages. To ease the task of processing natural language documents, a well-established approach is to employ a *controlled language* for writing specifications, restricting the grammar of a natural language (NL) to use only certain word forms and sentence structures, as well as reducing the vocabulary. A well-known industry standard, used in the aviation industry, is the AECMA Simplified English; the term Simplified English is also often used for controlled languages in general. There, the goal of controlled languages is to ease understanding of a document for non-native English readers, as well as facilitate unambiguous translations into other languages.

Processing specifications in natural language.
Understanding natural language use cases is explored in [25], with the goal to discover relations among concepts. The authors also employ a natural language parser, to ease the work of the parser, they propose a controlled language with simple rules on word forms and sentence structure, which coin the same principles as the general guidelines for writing use cases proposed elsewhere; a tool is implemented to aid the writer in conforming to the restrictions imposed by the controlled language.

A Two-Level Grammar (TLG) is used in [15] to translate natural language requirements into a knowledge base (in TLG), which is subsequently translated into VDM. This is further used in [27] for automatic extraction of QoS requirements from natural language specifications; the approach employs the complete domain vocabulary and a tailored parsing scheme.

The CICO domain specific parser [8] is used in [7] and [9] to discover dependencies among actions and relations and between actions and values in the system. Contrary to our approach, CICO uses domain specific rules and annotations.

The Simple Interfacing (SI) framework described in [11] puts focus on creating user interface from use cases, with the intention to verify that implementation is consistent with requirement specification; the approach assumes direct translation of natural language use cases into SI code.

The authors of [6] employ linguistic techniques to analyze quality of use cases. Criteria and metrics for evaluating the quality of use case specifications are defined; defects measured include, e.g., *vagueness* and *unexplanation*. The goal is to evaluate quality of use cases and check their conformance with the Simplified English controlled language.

The work in [12] targets checking consistency of natural language requirement specifications with UML designs. Instead of employing a natural language parser, the parsing code is captured as axioms in a knowledge base. An interesting point in this approach is that the lookup table (for matching natural language terms and UML model classes) is constructed based on already discovered matches and the UML design specification (finding the most appropriate match).

In [20], a controlled language and a rule-based parser are used to analyze NL requirements with the goal to assign Logical Form (LF) to NL 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 to use (and read), as well as hard to learn to follow.

6. Conclusion & Future Work

We have described a way to convert a textual use case specification into a behavior specification by employing linguistic tools and by identifying the type and principal attributes of the action described by a use case step based on the parse tree of the sentence specifying the step. In the subsequent stage, the structure of a use case is converted into a Pro-case (employing the Behavior Protocols notation [21]). The conversion described in this paper is implemented in a prototype tool, providing reasonable accuracy on recognizing the action type and communicating actor; the accuracy of estimating the event token is hard to evaluate. By employing the uniform structure of sentences in a use case specification, it is possible to define a conversion scheme utilizing only the principal information contained in a use case specification and to construct a behavior specification.

Thus, it is possible to write use cases in natural language, readable to a wide audience, and enjoy at least some of the benefits of formal methods at the same time.

Future work: There is a strong potential in developing interactive tools employing the conversion described in this paper. In an interactive use case writing tool, the event token might be offered immediately after a use case step is written; here, feedback from the user on the quality of the parse might be used in subsequent processing. Also, the tool might offer several parses (highlighting the most probable ones). Based on the parse(s) offered, the user might instead rephrase the sentence, improving the clarity of the use case. Also, the interaction of the user and the tool might facilitate maintaining event tokens consistent across the use case model. Future extensions of the analysis include identifying nested calls (expressed with curly braces in the manually created Pro-case in Fig. 10).

Another area to explore is simulating the execution of a use case specification, based on the behavior specification obtained. Such a simulation might be used to validate the requirement specification by generating test scenarios.

Last but not least, we also aim to perform a case study on an industrial use case requirement specification.

References

[1] Charniak, E.: Statistical Techniques for Natural Language Parsing. AI Magazine 18(4): 33-44 (1997)

[2] CLAWS part-of-speech tagger for English, http://www.comp.lancs.ac.uk/ucrel/claws/

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

[4] Collins, M.: A New Statistical Parser Based on Bigram Lexical Dependencies. ACL 1996: 184-191. 34th Annual Meeting of the Association for Computational Linguistics, 24-27 June 1996, University of California, Santa Cruz, California, USA, Proceedings. Morgan Kaufmann Publishers

[5] Collins, M.: Head-Driven Statistical Models for Natural Language Parsing. PhD Dissertation, University of Pennsylvania, 1999.

[6] Fantechi, A., Gnesi, S., Lami, G., Maccari, A.: Application of Linguistic Techniques for Use Case Analysis, Proceedings of RE 2002, pp. 157-164, Sep 9-13, 2002, Essen, Germany. IEEE Computer Society 2002

[7] Gervasi, V., Nuseibeh, B.: Lightweight validation of natural language requirements. Softw., Pract. Exper. 32(2): 113-133 (2002)

[8] Gervasi, V.: The Cico domain-based parser. Technical Report TR-01-25, University of Pisa, Dipartimento di Informatica, November 2001.

[9] Gervasi, V.: Synthesizing ASMs from natural language requirements. In Proc. of the 8th EUROCAST Workshop on Abstract State Machines, pages 212-215, February 2001.

[10] Graham, I.: Object-Oriented Methods: Principles and Practice, Addison-Wesley Pub Co, ISBN: 020161913X, 3rd edition December 2000

[11] Kantorowitz, E., Tadmor, S.: A Specification-Oriented Framework for Information System User Interfaces. OOIS Workshops 2002: 112-121

[12] Kozlenkov, A., Zisman, A.: Are their Design Specifications Consistent with our Requirements?, Proceedings of RE 2002, pp. 145-156, Sep 9-13, 2002, Essen, Germany. IEEE Computer Society 2002, ISBN 0-7695-1465-0

[13] Kulak, D., Guiney, E.: Use cases: requirements in context, Addison-Wesley, Pub Co, ISBN: 0-201-65767-8, May 2000

[14] Larman, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, Prentice Hall PTR, ISBN: 0130925691, 2nd ed, 2001

[15] Lee, B.-S., Bryant, B.R.: Automated conversion from requirements documentation to an object-oriented formal specification language, Proceedings of SAC 2002, March 10-14, 2002, Madrid, Spain, ACM 2002

[16] Lin, D.: MINIPAR, http://www.cs.ualberta.ca/~lindek/

[17] Mach, M., Plasil, F.: Addressing State Explosion in Behavior Protocol Verification, in Proceedings of SNPD'04, Beijing, China, Jun 2004

[18] Minnen, G., Carroll J., Pearce, D.: Applied morphological processing of English, Natural Language Engineering, 7(3), pp. 207-223, (2001)

[19] OMG: Unified Modeling Language: Superstructure, version 2.0, Final Adopted specification, ptc/03-08-02, http://www.omg.org/uml/

[20] Osborne, M., MacNish, C. K. : Processing Natural Language Software Requirement Specifications, Proceedings of ICRE 1996, pp. 229-237, April 15 - 18, 1996, Colorado Springs, Colorado, USA. IEEE Computer Society

[21] Plasil F., Visnovsky, S.: Behavior Protocols for Software Components. Transactions on Software Engineering, IEEE, vol 28, no 11, Nov 2002

[22] Plasil, F., Mencl, V.: Getting “Whole Picture” Behavior in a Use Case Model, in Transactions of the SDPS: Journal of Integrated Design and Process Science, vol. 7, no. 4, pp. 63-79, Dec 2003, ISSN-1092-0617

[23] Plasil, F., Mencl, V.: Use Cases: Assembling “Whole Picture Behavior”, TR 02/11, Dept. of Computer Science, University of New Hampshire, Durham, 2002

[24] Ratnaparkhi, A.: A Maximum Entropy Part-Of-Speech Tagger. In Proceedings of the Empirical Methods in Natural Language Processing Conference, May 17-18, 1996. University of Pennsylvania

[25] 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

[26] Rozenberg, G. (ed), Salomaa, A. (contrib.): Handbook of Formal Languages: Word, Language, Grammar, Springer Verlag; April 1997, ISBN: 3540604200

[27] Yang, C., Bryant, B. R., Burt, C. C., Raje, R. R., Olson, A. M., Auguston, M.: Formal Methods for Quality of Service Analysis in Component-Based Distributed Computing, Proceedings of IDPT 2003, Dec 2003, Austin, Texas, U.S.A.