DERIVING BEHAVIOR SPECIFICATIONS FROM TEXTUAL USE CASES

Vladimir Mencl

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

Abstract

The design of a software system or component starts with specifying its requirements; traditionally, use cases written in natural language are used for this task. While this makes use cases easily readable, 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 efforts usually attempted to utilize all the information possibly contained in a use case specification, thus facing the complexity of a 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 (component) modeled by a use case model, as well as to collect the list of operations accepted and requested by the entity; this may aid with defining the interfaces of a component.

1. Introduction

1.1. Textual Use Cases

The design of a software system or component starts with specifying its requirements; typically, use cases are employed for this task. 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. Although different methodologies [7, 3, 11] propose slightly different approaches to use cases, the common consensus [3, 10] is that a use case is usually specified as a sequence of steps forming the main success scenario specification, with extensions and variations specifying additional scenarios. Traditionally, natural language is used to describe the actions taken in a step; the main motivation for using natural language is to make use cases readable to a wide audience.

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

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.

(labeled with a number) describes an action that contributes to the goal of the use case. Alternative successful flows are specified in the variations section; exceptional situations (error handling) are specified as extensions. Both variations and extensions are attached to a step (by its label); their specification starts with a condition (guard), followed by a sequence of steps forming a nested block.

Besides laying out the structure of a use case, recommendations for writing use cases [3, 10] also provide guidelines imposing a simple and uniform structure for the sentences specifying the steps of a use case. Thus, although natural language processing (NLP) is a complex and difficult task in general, employing NLP tools to extract information from textual use cases may yield surprising results.

Use Case: M1 Seller submits an offer
Scope: Marketplace
SuD: Marketplace Information System
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

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 of a natural language sentence. A *phrase structure* parse tree captures the structure of the sentence according to the grammar of the respective natural language (English in this case); leaves reflect the words of the sentence (in left-to-right order), while intermediary nodes represent *phrases*. Figure 2 shows the parse tree obtained for step 1 of the use case shown in Fig. 1. There, the nouns "item" and "description" constitute 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) and also 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). 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" (fig. 2).

1.3. Goals of the Paper

In [21], we proposed guidelines for manually transforming a textual use case into a Pro-case, a notation for use cases based on the formal specification method Behavior Protocols [20], developed within our research group. In this paper, we describe how readily available tools for natural language processing can be employed to accomplish this task in an automated way.

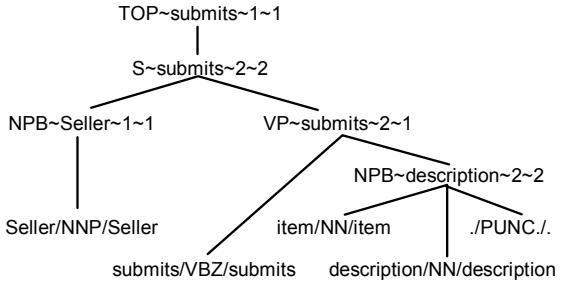

Figure 2: Parse tree of the sentence*: "Seller submits item description"*

The key goal is to utilize the simple and uniform structure of sentences specifying the action of a step in a use case to extract the principal attributes of the action from the parse tree of the sentence.

Subsequently, we demonstrate how a textual use case can be transformed into a Pro-case, specifying the communication and internal actions of the entity described by a use case model.

The paper is structured as follows: Section 2 focuses on obtaining the action principal attributes from a parse tree; in Sect. 3, we employ these results to convert a textual use case into a behavior specification. We evaluate our approach in Sect. 4 and discuss related work in Sect. 5; we conclude the paper and outline future work in Sect. 6.

2. Analyzing a Use Case Step

The use case writing guidelines prescribe a uniform sentence structure for steps of a use case; readily available linguistic tools allow to obtain a parse tree for such a sentence. In this section, we demonstrate how the *principal attributes* of the action (i.e., the action type, actor and event token) can be obtained from the parse tree; in this analysis, we use the subject and the direct and indirect object of the sentence.

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 [21] and the complete set of use cases published in [22] (minor modifications have been done for this case study). Figure 1 shows the first use case of this model; the scope diagram [3, 21] in Fig. 3 shows the entities involved. We use only a minimal domain model, consisting of the names of entities (as shown in Fig. 3) and the list of conceptual objects: "item", "offer", "price assessment", "price" and "payment method".

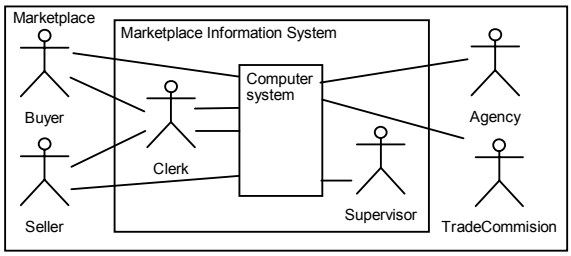

Figure 3: Scope diagram of the Marketplace system

2.1. Use Case Sentence Structure Premises

The following premises, derived from the use case writing guidelines [3, 7, 10], form the basis of the conversion described in this paper:
Premise 1: 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.
Premise 2: Such action is described by a simple English sentence following a uniform pattern.

These premises are supported not only by the core works on writing use cases [3, 7], but similar recommendations can also be found in recent research [5, 6, 9, 12, 19, 24] on natural language use case processing. In particular, premise 1 is supported by [3], requiring a step to describe *"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, [7] requires a step to describe a *"single atomic task"*.

As for premise 2, both [3] and [7] requires the sentence to follow the *SVDPI* pattern: *"The sentence structure should be absurdly simple" ... "Subject ... verb ... direct object ... indirect object"*. Such a sentence structure is also prescribed by the controlled language proposed in [24].

2.2. Linguistic Tools Used

We employ the well-accepted statistical parser [4] 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; we have also considered the parser developed by E. Charniak [2] and the MINIPAR parser [13] developed by D. Lin. For assigning POS-tags, we use the MXPost tagger [23]; note that a tagger may be included as a part of the parser in other settings [2]. In addition, we also employ the Morphological tools [16] developed at the University of Sussex to obtain the *lemma* (base form) of a word.

2.3. Determining Action Type

Step types. From premise 1, we conclude that a step of a use case specifies an operation *request*, either *received* by SuD from an actor, or *sent* by SuD to an actor, or an *internal action* of SuD. Providing support for this conclusion, the well-accepted object oriented methodologies [11] recommend deriving lists of operations of conceptual objects and system sequence diagrams from (textual) use cases.

In a sentence following the SVDPI pattern (premise 2), subject is the entity performing the action and 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 these rules are discussed in Sect. 2.5.) The first issue in analyzing the sentence describing a use case step is to determine the action type; the key lead is the subject.

Subject. In an SVDPI sentence, the main sentence node ("S") of the parse tree contains a *noun-phrase* node and a *verb-phrase* node (in this order); the first noun-phrase 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" refers to the primary actor of the use case (optionally specified in the use case header). In addition, the keywords "Use case" and "Extension" are recognized; these keywords indicate that the step describes a *special action* (sect. 2.5). A special action may also have no subject, e.g., "Resume with step 6". Note that a failure in identifying the subject 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.

If the entity referred to by the subject is an actor of the use case, the step describes a request sent by the actor to SuD. Thus, from the point of view of SuD, the step describes receiving a request from the actor. **Example:** In Fig. 2, subject is the name of the entity Seller; 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. We check 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 an apostrophe. In case the indirect object of the sentence is an actor, the action is 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.

2.4. Estimating Method Call

After determining the action type and the actor involved, the next goal is to construct an event token to represent the action in a machine processable behavior specification. Following the widely accepted recommendations to construct method names from the verb and the object of the sentence (as in system sequence diagrams in [11]), we construct the event token from the *principal verb* and the *representative object* description.

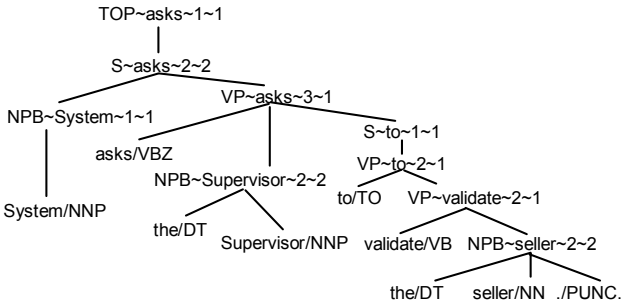

Figure 4: Parse tree of *"System asks the Supervisor to validate the seller."* (step 5 of use case CS1 in [22]).

Verb. The principal verb is the headword of the topmost verb phrase; an exception to this rule is a *padding verb*. A padding verb is only a syntactical construct, while the actual task is described by the subsequent verb; in this case, the principal verb is the next verb in the phrase. We have assembled a list of patterns identifying padding verbs: "ask", "be", "select to", "choose to". Note that we do not consider multiple padding verbs in a sentence, and we do not apply the padding-verb rule if there are no subsequent verbs in the sentence.
Example. In *"System asks the Supervisor to validate the seller."* (fig. 4), "asks" is a padding verb, and "validate" is the principal verb, describing the task requested. In Fig. 2, "submits" is the principal verb.

Representative object. We identify the direct object as the first basic noun phrase subordinate to the principal verb; we do not consider phrases already used as the subject and the indirect object. We select relevant words from the direct object to obtain the representative object description. In this task, we employ the list of conceptual objects: in case the direct object contains the name of a conceptual object, the word(s) forming its name are used, otherwise, we use all nouns in the direct object noun phrase.
Example: In Fig. 2 ("Seller submits item description."), the direct object noun phrase consists of the nouns "item" and "description". The word "item" is a conceptual object and is therefore used as the representative object description.

Event token. We construct the event token from the principal verb and the representative object description. Technically, we concatenate base forms of these words; we employ a naming convention to use the first word (the verb) in lowercase, and capitalize the first letter of each subsequent word.
Example: In Fig. 2, for the principal verb "submit" and representative object description "item", the resulting event token is **`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. A *special action* changes the flow of control in a use case and is typically used in an extension (for simplicity, we use the term extension to refer to both extensions and variations). We have identified two kinds of special actions; a *terminate* action causes the enclosing block to terminate, while a *goto* action transfers control to a particular step, identified by a step label. By convention, a special action is used solely as the last step of a use case extension. Special actions are specified with very simple sentences, closely following a predefined pattern (also recommended in [3]); typically used constructs are "Use case aborts" (for a terminate action) or "Resume with step <label>"(goto action). In these patterns, the subject is a predefined keyword such as "Use case", or the sentence has no subject at all; the verb is also one of only a few predefined words.

Terminate. A terminate action terminates the flow in the enclosing block. A *propagating* terminate action terminates the whole use case; otherwise, the action terminates only the enclosing block. A terminate action may be an *aborting* action, capturing a failure. We identify a terminate action when the subject matches the keyword "Use-case" (propagating action) or "Extension" or the sentence has no subject, and at the same time, the verb matches one of the predefined patterns: "abort", "end" and "terminate"; the pattern "aborts" in addition indicates an aborting action.

Goto. A goto action transfers control to a particular step in the enclosing block. The sentence follows a very simple pattern; the 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 predefined 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 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.

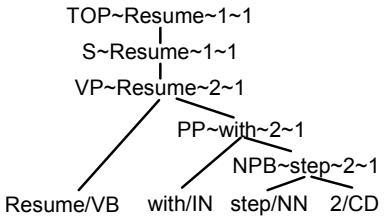

Figure 5: Parse tree of sentence: *"Resume with step 2"*

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 (e.g., variation 2b in Fig. 1); this is also common in use case practice. The guidelines in [3, pp. 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

Each extension (and variation) has a textually-specified triggering condition. We do not aspire to interpret conditions expressed in natural language. Opposite to the steps of a use case, little is known about the way extension conditions are specified. In [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)". As an achievable goal, we envision representing the condition with an explanatory identifier, named in a way, e.g., a programmer would name a variable representing the condition. We have already outlined rules to extract descriptive information from the condition specification to construct a *condition label* by selecting significant words; e.g., for the condition "Order is not valid", these rules yield the condition label `offerNotValid`. However, this deserves more attention in our future work; in this paper, we represent conditions with a label constructed solely from the extension identifier (as shown later in Fig. 8). Please note that we aim to translate to behavior specification mechanisms that either do not feature conditions, or that model a condition as a black-box internal (condition) event. Thus, the constructed condition label plays only an informative role.

2.7. Analyzing Use Case Steps: Summary

We have demonstrated that is possible to obtain the principal attributes of an action specified by a step of a textual use case from the parse tree of the sentence specifying the step. 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 target. Figure 6 shows the automatically obtained event tokens (left), compared to the results previously obtained by hand in [21] (right). The event tokens match on action type and actor identified (except for step 8, relying on context) and provide similar estimates of event names. In Fig. 6, ? denotes an event received, ! an event sent, and # (resp. τ) an internal event; an event token is composed of connection name (actor identifier) and event name.

Discussion on failures. When a part 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.

3. Converting Textual Use Cases to Pro-cases

Having constructed the event tokens, we approach the second step in deriving a behavior specification from a use case. From the wide range of formal techniques (possibly ranging from state machines with transitions labeled by method calls as used in UML [18], to process algebras), we choose Pro-cases [21], based on Behavior Protocols (BP) [20] for the following reasons: (i) same as for use cases, Pro-cases 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) and (iii) behavior protocols are designed to specify behavior of software components (matching the use of use cases for nested entities).

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 of course be inherently imprecise and is only an estimate of the actual behavior specification. But, regardless that, the estimate can be of high value to developers.

3.1. Use Cases vs. Pro-cases: Structure

Pro-cases specify behavior in terms of atomic events emitted (!), absorbed (?) and internally processed (τ) by an entity; the syntax stems from regular expressions. A primary expression specifies a single event; among the operators are (in priority order): `*` for repetition, `;` for sequencing, and `+` for alternative (and additional operators not used here). Figure 8 shows a Pro-case automatically constructed with our tool; Figure 9 shows the Pro-case manually created for the same use case in [21]. Note that in Fig. 8, "#" is used instead of τ. Also, nested calls are not detected (expressed with curly braces in Fig. 9, but not present in Fig. 8).

Conveniently, the behavior of a Pro-case (a set of finite sequences of events) forms a regular language. This permits reasoning on consistency of Pro-case specifications; the compliance relation is decidable and a verifier tool is available [14]. Note that conditions are not supported in Pro-cases, this is actually a trade-off for the regularity of the language. Thus, in a Pro-cases, we instead represent

1	?SL.submitItem	1.	?sic.submitItem
2	#validateDescription	2.	τValidateItem
3	?SL.adjustPrice	3.	?sic.submitPrice
4	#validateSeller	4.	τValidateSeller
5	#verifySeller	5.	τVerifySellerHistory
6	!TC.validateOffer	6.	!tradecom.validate
7	#listOffer	7.	τListOffer
8	#respondAuthorization-Number	8.	!sellernotify.putAuthNr
		Extensions & Variations	
2a1	%ABORT	2a1	Null //Abort
2b1	!SL.providePrice-Assessment	2b1	!sellernotify.putPrice-Assessment
5a1	%ABORT	5a1	Null //Abort
6a1	%ABORT	6a1	Null //Abort

Figure 6: Event tokens obtained for use case M1 (fig. 1) automatically (left) and manually in [21] (right)

conditions with a *condition event* – a special case of an internal event. The semantics is that a condition event may only be processed when the condition holds; except for this, the condition is kept outside of the behavior model. 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 a use case into a Pro-case by first constructing a finite automaton representing the use case (transitions are labeled with the event tokens already obtained); afterwards, we derive the Pro-case as a regular expression generating the same language as the automaton. The description of this process is elaborated in detail in [15]; here, we only briefly illustrate the idea.

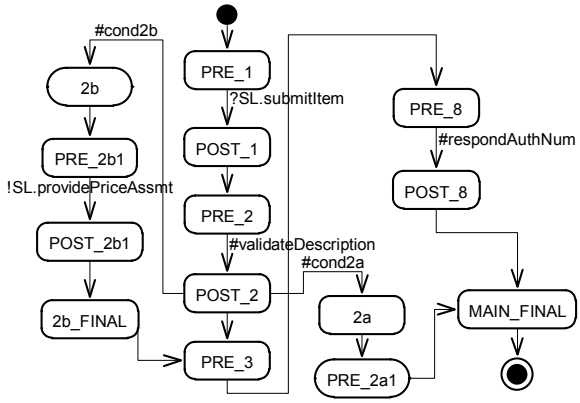

Figure 7: Part of the automaton constructed for the use case "Seller submits an offer"

We start processing the main success scenario specification (the top-level block). To correctly capture the flow of a use case, we create a *pre-i* and *post-i* state for each step *i* in the block; for each block, we also create a *block final* state. We add a *lambda transition* (no event processed) from each *post-(i-1)* to *pre-i*. A step *i* describing a communication or internal action is mapped into a transition from *pre-i* to *post-i* labeled with the event token of the step *i*. A goto special action results into a lambda transition from *pre-i* to the *pre*-state of the target step. A terminate special action maps to a lambda transition from *pre-i* to the final state of the enclosing block; propagating terminate actions are handled in a special way (adding a transition to the final state of the enclosing block until the top-level block is reached). For each extension *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 process the extension block in the same way. We illustrate the construction in Fig. 7; the automaton fragment shown corresponds to the first two steps of the main success scenario and the associated extensions. To acquire the Pro-case, we use the generic algorithm for deriving a regular expression from a generalized nondeterministic finite automaton described, e.g., in [26]. Figure 8 shows the Pro-case automatically obtained with our prototype tool for the use case displayed in Fig. 1.

4. Evaluation & Open Issues

Evaluation. We have implemented this conversion in a prototype tool and evaluated our approach in a case study on a set of use cases published in [22], developed within our previous work [21] proposing

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

Figure 8: 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 9: Pro-case manually created for the use case "Seller submits an offer"

manual conversion of textual use cases to Pro-cases (thus, without considerations for automatic processing). The use cases were understood by the tool with only minor modifications – mostly typo corrections; several sentences in the original use case model did not follow the guidelines (use active voice, write short sentences) and had to be corrected (change to active voice, simplify sentences). 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 case requirement specifications.

Open issues. The way event tokens are constructed should be enhanced to yield matching event tokens for complementary send / receive actions in use case models of communicating entities. Also, at least minimal support for (pre-declared) synonyms should be included. 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; to address this, extensions to the underlying behavior protocols notation are being investigated.
As use cases have a primarily informative role, the execution semantics of use cases is given only by general consensus, leaving open, e.g., how to identify an extension that is executed *instead* (and not after) the action of the step, or whether multiple extensions attached to the same step may be triggered.

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 [24], with the goal to discover relations among concepts. To ease the work of the natural language parser employed, the authors propose a controlled language with simple rules on word forms and sentence structure, which coins the same principles as the general use case writing guidelines; 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 [12] 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. In [6], Abstract State Machines are synthesized from natural language requirements; contrary to our approach, the CICO domain specific parser [5] used employs domain specific rules and annotations. The Simple Interfacing (SI) framework described in [8] puts focus on creating user interface from use cases, assuming direct translation of

natural language use cases into SI code; however, details on processing the textual use cases are not discussed. The COLOR-X project [1] proposes an event model based on a low-level language closely reflecting a natural language. Although only manual processing of requirements is considered, the project takes an approach similar to our transformation, identifying events by detecting verbs in the specifications and also using a lexicon of already know phrases. Based on the event model constructed, a State Transition Diagram is generated.

The work in [9] 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 with UML model classes) is constructed based on already discovered matches and the UML design specification (finding the most appropriate match). In [19], 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.

The author of [17] provides a formalization of the Object-Oriented Analysis process, in particular of creating the object and behavior models. The formalization is based on linguistic structures, with focus on the Spanish language. The author identifies linguistic patterns and their corresponding conceptual patterns. The goal is to provide a formalization of the process; automation of the analysis is not considered. In a similar way, [25] argues that the requirements engineering process should be supported by a CASE tool based on a linguistic approach and formalizes the linguistic mechanisms used by analysts. Rule-based parsing is employed to generate a conceptual schema; later, the requirements are validates by paraphrasing the original requirements – generating natural language sentences from the conceptual schema.

6. Conclusion & Future Work

We have described a way to derive a behavior specification from a textual use case by employing linguistic tools and by identifying principal attributes of the action described by a use case step based on the parse tree of the sentence specifying the step. Subsequently, we convert a use case into a Pro-case (employing the Behavior Protocols notation [20]). We have implemented this conversion in a prototype tool, successfully recognizing the action type and the communicating actor and providing reasonable estimates of the event token. For a use case model, the tool yields (an estimate of) the behavior specification of the entity, as well as the list of events to be handled by the entity; these may be employed in designing the interfaces to the entity in the initial design stage. Thus, it is possible to write use cases in natural language, readable to a wide audience, and at the same time enjoy at least some of the benefits of formal behavior specifications.

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 entered. The feedback from the user on the quality of the parse might be used in subsequent processing; also, the user might instead rephrase the sentence, improving the clarity of the use case. Also, the interaction between 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. 9). Another area to explore is simulating the execution of a use case specification, based on the behavior specification obtained, with the goal 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] Burg, J. F. M., van de Riet, R. P. : COLOR-X: Linguistically-based Event Modeling: A General Approach to Dynamic Modeling, in Proceedings of CAiSE 1995, Jyväskylä, Finland, June 12-16, 1995, LNCS 932, Springer 1995

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

[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] Gervasi, V.: The Cico domain-based parser. TR-01-25, University of Pisa, Dipartimento di Informatica, Nov 200

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

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

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

[9] 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

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

[11] 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

[12] 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

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

[14] Mach, M., Plasil, F.: Addressing State Explosion in Behavior Protocol Verification, Accepted for publication in proceedings of SNPD'04, Beijing, China, Jun 2004

[15] Mencl, V.: Converting Textual Use Cases into Behavior Specifications, Tech. Report No. 2004/5, Dep. of SW Engineering, Charles University, Prague, Aug 2004

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

[17] Moreno, A.M.: Object-Oriented Analysis from Textual Specifications, in Proceedings of SEKE'97, Madrid, Spain,1997

[18] OMG: UML Superstructure, version 2.0, Final Adopted specification, ptc/03-08-02, http://www.omg.org/uml/

[19] 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

[20] Plasil F., Visnovsky, S.: Behavior Protocols for Software Components. IEEE Trans. Software Eng. 28(11): (2002)

[21] 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

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

[23] 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

[24] 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

[25] Rolland, C., Proix, C. : A Natural Language Approach for Requirements Engineering, in Proceedings of CAiSE'92, Manchester, UK, May 12-15, 1992, LNCS 593, Springer 1992

[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.