# A Use Case Grammar for Requirements Specification

***Marinos Georgiades***
*University of Tartu*
*Tartu, Estonia* *marinos.georgiades@ut.ee*

**Abstract**

This paper proposes a use case grammar for the specification of functional requirements. The proposed grammar is defined in EBNF, tested in ANTLR and provides syntactic and semantic rules for writing use case specifications in semi-formal natural language. Such formalization not only helps to make the expression of requirements more disciplined, understandable, well-structured and validated, but it also makes easier their conversion into diagrammatic notations, such as use case and sequence diagrams. It also helps in reducing the time to identify and specify requirements, in diminishing redundancies, inconsistencies and omissions, and, generally, in producing better requirements.

**Keywords:** use case modeling, requirements formalization, use case grammar, software engineering, requirements engineering

## 1. Introduction

Use case modeling is a popular way of describing functional requirements, in the field of software engineering. One reason for its popularity is that a well-written use case is relatively easy to read, since it is written in natural language following a scenario style. However, easy-to-read does not also mean easy-to-write. The most common mistakes in writing use cases are related, on the one hand, to the omission of information, such as omitted use case actions, subjects of sentences, and related use cases (e.g., extending or included ones), and, on the other hand, to the use of redundant information such as assumptions about the user interface design, duplicate actions, and actions with technical or out-of-scope details. Table 1 depicts examples of such mistakes:

**Table 1.** Bad and good examples of use case writing for use case "Create prescription".

| Bad specification | | Good specification |
|---|---|---|
| … | | … |
| 3. Input name.<br>4. Input address.<br>5. Input ID. | } | 3. **Patient** inputs personal information (name, address, ID). |
| | | 4. **System validates** patient information. |
| 6. Select medicines and quantities. | → | 5. **Doctor** selects medicines and quantities. [**Extension** point. UC Receive advice] |
| 7. Click **button** to submit prescription. | → | 6. **Doctor** submits prescription. |
| | | 7. **System** validates medicine availability and quantity. |
| 8. **Connect to database** and save prescription form. | → | 8. **System** saves prescription. |

- Steps 3,4 and 5 on the left column make redundant use of action “input”.
- The subjects/primary actors were wrongly omitted from the steps on the left column.
- Step 6, left column: extending use case “Receive advice” was wrongly omitted.
- Step 7, left column: interface details should be omitted at Requirements level; to be defined at Design level (e.g., “Doctor submits prescription by using button/stylus/voice).
- Step 8 on the left column wrongly goes into technical, low-level details.
- Validation steps were wrongly omitted on the left column; they have been included correctly in the right column (steps 4 and 7)

Existing use case driven analysis (UCDA) approaches often result in poorly defined use case models for the following reasons: (i) lack of semantic and syntactic preciseness for identifying the main elements of the use case model, such as use cases, actors, basic and alternative flow actions, pre- and post- conditions, and relationships [1]; (ii) use of free natural language (NL) to define the use case specifications, which leads to ambiguities and inconsistencies, conflicts with domain terminology and implementation details (jargon-contaminated use cases) which make the specifications difficult to be maintained and also understood by customers. El-Attar and Miller [2] recognize that these problems produce low quality information systems (ISs); (iii) limited support of dedicated software tools, which makes UCDA a time-consuming and error-prone activity. The lack of a formal semantics hinders the use of automated tools for use case modelling, refinement, verification and transformation, since informal NL is inherently complex, vague and ambiguous.

This paper presents a grammar that provides syntactic and semantic rules for writing use case specifications in semi-formal natural language. The proposed grammar utilizes fundamental information system elements, such as the functions for creating, altering, storing, presenting and transmitting information, and, therefore, it can be applied in the IS domain. The presented grammar is written in extended Backus–Naur form (EBNF); expressing the use case model in EBNF allows to formally prove key properties, such as well-formedness and closure, and, hence, help validate the semantics. Furthermore, the proposed grammar has been developed with Another Tool for Language Recognition (ANTLR), which is a parser and translator generator tool that lets one define language grammars in EBNF-like syntax as well as running concrete examples of the defined grammar and isolating any mistakes. To sum up, such a grammar not only helps to make expression of use case-based requirements more disciplined, understandable and organized, but it also makes easier their conversion into diagrammatic notations, such as use case and sequence diagrams.

The rest of this paper is structured as follows: Section 2 summarizes related work, Section 3 elaborates on the proposed grammar, and Section 4 presents conclusions and future goals.

## 2. Related Work

A number of structured techniques for the description of use cases have been proposed. In Eriksson et al.'s work [3], a tabular representation is used, and in Leite et al.'s [4], a structured natural language is presented for use case description. These structured representations provide a generic formalization of the use case (UC) specification template, hence not a clear formalism of the use case specification elements, and especially the basic and alternative flow actions. Osaki et al. [5] provide some textual statements representing actor and system behavior, however, in addition to the lack of action types included in their work as well as of other use case elements, the proposed actions are not placed in any order or sequence, and they also lack the well-formedness of a grammar with syntactical and semantic rules. Moreover, some formal techniques such as formal grammars [6], statecharts [7, 8] and scenario trees [8] were introduced for the description of use cases or their scenarios. Although such formal representations facilitate formal analysis, they are difficult for analysts and users to understand and use.

Savi et al. [9, 10] have developed a grammar in BNF that covers only three types of actions; these actions are "Actor (user) prepares data for system operation execution", "Actor call system to execute system operation", and "System returns results of system operation execution". In addition to the lack of action types included in their grammar, user actions are placed as a separate group from system actions within their use case specification, contrary to the grammar proposed in this paper that it associates a use case with a specific action block, and according to the type of the action block (Create, Alter, Erase, Read), specific user and system actions, combined in a specific sequence, are used to form the main flow of the use case. Golbaz et al. [11] also propose a use case grammar

which is defined in XML, but their work lacks covering any specific types of actions. However, their XML grammar and the fact that they use XSLT to transform their use cases from a concrete syntax expressed in semi-formal language to that of a natural language-like form provide useful insights for future work related to the grammar proposed in this paper.

Jacobson [12] has also suggested the following well-known combination of use case actions:

- The primary actor sends request and data to the system
- The system validates the request and the data
- The system alters its internal state
- The system responds to the actor with the result.

This is a generic approach, lacking, on the one hand, the well-formedness of a grammar with syntactical and semantic rules, and, on the other hand, of additional and more specific user and system actions together with the rest of the elements of a use case specification, all defined under a use case grammar.

## 3. The proposed grammar

The proposed EBNF-based grammar has been developed with the **An**other **T**ool for **L**anguage **R**ecognition (ANTLR) parser generator. ANTLR is a parser and translator generator tool that lets one define language grammars in EBNF-like syntax. In particular, this work has used ANTLRWorks, which is a grammar development GUI environment for ANTLR grammars. It combines a grammar-aware editor with an interpreter for rapid prototyping and a language-agnostic debugger for isolating grammar errors [13]. The following paragraphs elaborate on the abstract syntax of the most important rules of the proposed grammar as well as on the concrete syntax of several rules. Section 3 concludes with a complete concrete syntax for the use case *Create Prescription*.

The proposed grammar starts with defining the root parser rule *UseCaseSpecification* (Table 2). It defines the main elements of a use case specification including the use case identification through the parser rule *useCase*, the use case description through the parser rule *description*, the primary and secondary actors through the parser rule *actors*, preconditions and postconditions through the parser rules *preconditions* and *postconditions*, respectively, the main flow through the parser rule *mainFlow*, and the exceptions through the parser rule *exceptions*.

**Table 2.** Abstract syntax of the *useCaseSpecification* parser rule

```
useCaseSpecification
        :       useCase
                description
                preConditions
                actors
                mainFlow
                exceptions
                postConditions ;
```

The *useCase* parser rule defines a use case with a unique identifier and a use case name (Table 3). In EBNF, the question mark symbol denotes optional behavior that may happen only once, so the use of the unique identifier is optional.

**Table 3.** Abstract syntax of the *useCase* parser rule

```
useCase
        : ucID? ucName
        ;

ucID
        :       'UC' INTEGER   ;
```

Each use case is related to one domain concept, namely the Information Object (IO), and it

also falls under a use case type; these two (IO, type) make up the use case name which is defined by the parser rule below (Table 4).

**Table 4.** Abstract syntax of the *ucName* parser rule.

```
ucName
        :       ucType io       ;
```

Note: The parser rules *ucType* and *io* are presented in Tables 6 and 5, respectively.

According to Georgiades and Andreou [14]:

> *An information object (IO) is a digital representation of a tangible or intangible entity—described by a set of attributes—which the users need to manage through creating, modifying, reading, and erasing its instances, and be notified by the messages each instance (IOi) can trigger*[1]. [p. 269]

Information objects are distinguished into the following categories: business role (as a type of animate [supertype] entity, e.g., doctor), inanimate entity (e.g., car), procedure (e.g., translation), document (e.g., book), event (e.g., appointment), site (e.g., country, hospital), and state (e.g., disease). This distinction makes easier the identification and organization of the basic elements of an information system and their association with use case elements (e.g., business roles[people]→actors, *ucType*+IO[function]→use case), their relationships, and the identification of the attributes of an IO, which (viz. the attributes) have to be processed by use cases. It is worth noting that by specifying the type of an IO, the analyst will be able to identify its attributes more easily (more information on this is out of the scope of this paper). Table 5 below presents the abstract syntax of the parser rule *io* for the Information Object.

**Table 5.** Abstract syntax of the *io* parser rule

```
io
        :       ioName ioType?
        ;

ioName
        :       CAPITALIZED_NOUN | 'Authentication'
        ;

ioType
        :       'type' '"' IO_TYPE '"'
        ;

Where

IO_TYPE : 'business role' | 'inanimate'| 'procedure'| 'document'| 'event'| 'site'|
'state';

CAPITALIZED_NOUN : ('A'..'Z') ('a'..'z'|'A'..'Z'|'_')*;
```

Note: Concrete syntax examples of the *io* parser rule could be "Prescription" and "Examination", or "Prescription type document" and "Examination type procedure".

The parser rule *ucType* (Table 6) indicates the types of functions which are applicable to an information object. *ucType* use cases follow the nine use case patterns proposed by Georgiades and Andreou [14], namely (a) Create information object (IO), (b) Correct IO, (c) Alter IO state, which leads to the more specialized patterns Cancel IO, Archive IO, and Complete IO, (d) Erase IO (e) Read IO, (f) Read IO intra-report | inter-report, (g) Read supporting information, (h) Notify, and (i) Authorize. It is worth noting that as *ucType* use cases are applied to electronic information following the abovementioned nine patterns, strong verbs of the domain vocabulary, such as "enroll" and "register", should be examined if they can be transformed to information objects in the form of their corresponding nouns. For example, "enroll" derives the noun "enrolment", and "register" derives the noun "registration". As a result, a use case titled "Enroll in Seminar" by conventional approaches (e.g., [15]), leads to the use of the IOs Enrolment and Seminar, under the proposed grammar, as well as to the use of the relevant *ucType* use cases for each IO. For example, for the IO

[1] An IO is conceived and processed at an abstraction level, while an IOi is conceived and processed at a factual level; instances of the same IO differ only in the values of their attributes.

*Enrolment*, there are the use cases *Create Enrolment*, *Correct Enrolment*, *Cancel Enrolment*, *Archive Enrolment*, *Complete Enrolment*, etc., and for the IO Seminar, there are the use cases *Create Seminar*, *Correct Seminar*, *Cancel Seminar*, *Archive Seminar*, *Complete Seminar*, etc.

**Table 6.** Abstract syntax of the *ucType* parser rule

```
ucType
        :       'create' | 'creates' | alter | read | 'erase' | 'erases' | notify
        ;

alter
        :       'correct' | 'corrects' | alter_expanded
        ;

alter_expanded
        :       'cancel' | 'cancels' | 'archive' | 'archives' | 'complete' |
'completes' | VERB
        ;


read
        :       'read' | ('read' 'report' 'about') | 'reads' | ('reads' 'report'
'about')
        ;

notify
        :       ('send' | 'sends') 'Notification' ('about' 'the' ('creation' |
'altering' | 'reading' | 'erasing'))? 'of'
        ;
```

Table 7 below shows the abstract parser rule *description* and its concrete syntax for the use case *Create Prescription.*

**Table 7.** Abstract and a concrete syntax of the *description* parser rule

```
description
        :       'Description' ':' primaryActor ucName '.' (((secondaryActor) ('and'
secondaryActor)*) (('provides' 'or' 'verifies') | ('provide' 'or' 'verify'))
'relevant' 'information.')? (secondaryActor ('and' secondaryActor)* ('is an intended
recipient' | 'are intended recipients') 'of' io '.')?

Description: Doctor creates Prescription. Patient and Medical_system provide or verify
relevant information. Pharmacist is an intended recipient of Prescription.
```

Actors are defined through the *actors* parser rule, as depicted below (Table 8). Specifically, the rule allows the definition of one primary actor and zero or more secondary actors.

**Table 8.** Abstract syntax of the *actors* parser rule

```
actors
        :       'Actors' ':' primaryActor (',' secondaryActor)*
        ;
```

Primary actors fall into three categories, and secondary actors fall into two categories, as defined by the rules in Table 9 below.

The Section 3.2 complete concrete syntax of the use case *Create Prescription* presents the concrete syntax of the actors *doctor*, *patient* and *pharmacist*, based on the corresponding abstract rules depicted in Tables 8 and 9. For clarification purposes, the same concrete syntax is depicted here, too:

```
Actors: Doctor (type: creator), Patient (type: accompaniment), Pharmacist (type:
  intended recipient)
```

To identify the actors involved in each use case, the functional roles provided by Georgiades and Andreou [16] are utilized. Indicatively, a *Create* use case involves the functional roles *creator*, *accompaniment*, *intended recipient*, and *notifiee*. The creator is played by a primary actor, while the accompaniment, intended recipient and notifiee are played by secondary actors. The creator is responsible for setting (or confirming) the values of a number of particular attributes (required and optional) of an IOi (e.g., *Doctor* is the

creator of prescriptions in the UC *Create Prescription*). The accompaniment participates in close association with the creator to help in the creation of an instance of the IO (e.g., *Patient* provides to the *Doctor* information about his/her physical condition/pain so that an examination is created through the *UC Create Examination*. Another accompaniment in this use case could be the *Nurse* that provides help during the examination, including the completion of the relevant examination form. The intended recipient (IR) takes action after being notified about the creation of an IOi. The action to be taken needs to fulfil the purpose of having/using the IO, and the fulfilment is achieved by creating or altering instances of other related IOs. For example, in the UC *Create Prescription, Pharmacist* is one IR of the *Prescription* IO, because after the creation of a *Prescription* IOi, the pharmacist will fulfil the purpose of having/using the prescription (the purpose is to provide medicines to the patient) by altering the relevant *Medicine* IOi (since the medicine handed to the patient must be removed electronically from the system, by reducing its quantity).

**Table 9.** Abstract syntax of the *primary* and *secondary* actors parser rules

```
primaryActor
        :       actorName paType?
        ;

actorName
        :       CAPITALIZED_NOUN
        ;

paType
        :       '(' 'type:' PA_TYPE ')'
        ;

secondaryActor
        :       actorName saType?
        ;

saType
        :       '(' 'type:' SA_TYPE ')'
        ;

Where

PA_TYPE :       'creator' | 'alterer' | 'experiencer';

SA_TYPE :       'accompaniment' | 'intended recipient' | 'notifiee';

CAPITALIZED_NOUN : ('A'..'Z') ('a'..'z'|'A'..'Z'|'_')*;
```

To derive the actors from their functional roles, specific questions need to be made. The following are indicative question patterns for identifying (i) the creator: Who should create an <IO> ? Who has the responsibility for the creation of a(n) <IO>?; and (ii) the accompaniment: Who should assist the <Creator> to create an <IO>? How does the <Accompaniment> help the <Creator> during the creation of an <IO>?

Additionally, the collaboration between a primary actor and an accompaniment can derive both *include* and *extend* relationships, where extending or included use cases are invoked by their base use cases and involve the accompaniments. These use cases are called complementary. For example, during the creation of a prescription, the doctor may need to ask for the assistance of another doctor/counsellor or of a medical database system, in order, for example, to choose between two medicines for the treatment of a patient. In this case the counsellor and the medical system are accompaniments that provide feedback, and the use cases *Read Counsellor* and *Read Medical System Report* extend the behaviour of the use case *Create Prescription*.

Moreover, the use of the intended recipient helps in identifying preconditions by checking if the primary actor of one use case is an IR in another related (preceding, specifically) use case. For example, the primary actor of *UC Create Examination* is *Doctor; Doctor* is also an IR in the latter use case, as a doctor will be notified about the creation of an examination and take some action by creating or altering instances of another IO, that is, *Prescription;* therefore, in this example, a precondition of the UC *Create Prescription* is "Examination is in Complete state (from the related [preceding] use case *Create Examination*)."

Another type of precondition refers to the primary actor (e.g., the *Creator*) that initiates the use case. Normally, the system must check that the primary actor is authenticated or authorized to initiate the use case. For example, for the UC *Create Prescription*, "Doctor is authorized" is a precondition. The abstract syntax of preconditions is depicted in Table 10 below. Similarly, the abstract syntax of postconditions is depicted in Table 11.

**Table 10.** Abstract syntax of the *preConditions* parser rule

```
preConditions
        :       'Pre-conditions' ':' (INTEGER '.' precondition)*
        ;

precondition
        :       statePre | authe_pre | autho_pre | other_pre
        ;

statePre
        :       io 'is at' state 'state' '.' 'Triggered by' useCase (',' useCase)* '.'
//useCase is an essentially preceding use case which causes the initiation of the
state.
        ;

state
        :       'Cancelled' | 'Pending' | 'Complete' | SMALL_NOUN
        ;

authe_pre
        :       primaryActor 'is authenticated' '.'
        ;

autho_pre
        :       primaryActor 'is authorized' '.'
        ;

other_pre
        : IDENT
        ;

SMALL_NOUN : ('a'..'z') ('a'..'z'|'_')*;
```

**Table 11.** Abstract syntax of the *postConditions* parser rule

```
postConditions
        : 'Post-conditions' ':' (INTEGER '.' postCondition)*
        ;

postCondition
        :       io 'is at' state 'state' '.'
        ;
```

### 3.1. Main Flow

The main flow parser rule *mainFlow* defines the actual execution of the use case. It includes one of the action blocks *actionBlockCreate*, *actionBlockAlter*, *actionBlockErase* and *actionBlockRead*. Below (Table 12) is the abstract syntax of the main flow parser rule.

**Table 12.** Abstract syntax of the *mainFlow* parser rule

```
mainFlow
        :       'Main' 'Flow' ':' actionBlockCreate | actionBlockAlter |
actionBlockRead | actionBlockErase
        ;
```

Moreover, use case actions are divided into two main types: (i) user actions which are performed by the users, and these are: input, request, submit, and confirm; (ii) system actions which are performed by the system, and these are: validate, save, prompt, notify, and calculate. All these actions extend the ones provided by Georgiades and Andreou [14]. Their corresponding parser rules will be presented further below in this paper.

Additionally, according to the type of the action block (Create, Alter, etc.), there is a difference on what user and system actions are used as well as their sequence. Below (Table 13) is the sequence of the actions of the *actionBlockCreate*.

**Table 13.** Abstract syntax of *actionBlockCreate*

```
actionBlockCreate
        :       initialUA
                validateAuthorization?
                (presentCreateSA extended_by? Includes?)
                (inputUA extended_by* validate_inputSA?)+
                submitUA
                validateSA
                saveSA
                (notifySA includes?)? ;
```

The parser rule *actionBlockCreate* starts with an initial user action, that is, a request to create a new information object (Table 14). Table 15 depicts a concrete syntax validated in ANTLR.

**Table 14.** Abstract syntax of the parser rule *initialUA*

```
initialUA :     'UA' INTEGER '.' primaryActor 'requests to' ucName   '.';
Note: For actionBlockCreate, the ucType (part of ucName) value is "Create".
```

**Table 15.** A concrete syntax of the parser rule *initialUA*

```
UA 1. Doctor requests to create Prescription.
```

The parser rule *validateAuthorization* (mentioned in Table 13) refers to a system action that will check if the actor is logged in and authorized to create this information object. As previously mentioned, in EBNF the question mark symbol denotes optional behavior that may happen only once. Subsequently, the parser rule *presentCreateSA* involves another system action, that is, the system will prompt the user, with a form, to complete the latter and, thus, create an instance of the IO. The abstract syntax of this rule is depicted in Table 16, while a concrete syntax is presented in Table 17.

**Table 16.** Abstract syntax of the parser rule for prompting a user to fill a form and create an IOi

```
presentCreateSA :       'SA' INTEGER '.' 'The system prompts' primaryActor 'to
                    input the required and optional attribute values of the' io '.' ;
```

**Table 17.** A concrete syntax of the parser rule for presenting a form to create an IOi

```
SA 2. The system prompts Doctor to input the required and optional attribute values of
the Prescription.
```

The parser rule extended_by (in Tables 13 & 18) denotes that the Create IO use case is extended by another use case (or more – denoted by a succeeding asterisk). In such a case, the phrase "Extension point." must be written on the right of the action that triggers the extension, followed by the ID and name of the extending use case. Similarly, the parser rule Includes (in Tables 13 & 18) denotes that the Create IO use case includes another use case (or more). The full example presented in section 3.2 includes the concrete syntax for invoking two extending use cases and one included use case. The paragraph below expands more on the extended_by rule.

**Table 18.** Abstract syntax of the parser rules "extend" and "include" relationships.

```
extended_by
        :       '[' 'Extension point.' useCase ']'
        ;

includes
        :       '[' 'via' useCase ']'  ;
```

Following, the parser rule *inputUA* (in Table 19) refers to the user's action of inputting data to the form for the creation of the IOi. *inputUA* is followed by the optional rule *extended_by** as depicted in the main flow of the Create action block in Table 13, which denotes that inputting data could be extended by zero or more other use cases that provide

supporting information, such as reports and documents; such supporting use cases could trigger the involvement of secondary actors. For example, for the UC *Create Prescription*, for the action *Input medicine*, the doctor might need guidance by a medical guide or a medical counsellor, hence UC *Create Prescription* is extended by UC *Read Counsellor Report*; the same action is also extended by the use case *Read Examination Report* (see the complete, concrete example in section 3.2). *inputUA* is also followed by the optional rule *validate_inputSA?* denoting that inputting data could be validated. System action 4 “The System validates PatientID of Prescription.” of the full concrete example is such a validation action. It is noteworthy to mention that in EBNF, the plus sign (+), appearing in Table 13 regarding *inputUA*, indicates "one or more occurrences."

**Table 19.** Abstract syntax of the parser rule for inputting form data

```
inputUA
        :       'UA' INTEGER '.' primaryActor 'input' | 'inputs'
        ioAttribute '.' (secondaryActor ('provides' | 'verifies') 'this information'
'.')?   ;
```

*ioAttribute* determines the value added to the form for an attribute of the IOi. If there are more than one attribute values to be added, then an iteration of this addition will take place as depicted in Table 19 and in the example of section 3.2. Table 20 below presents the abstract syntax of the IO attribute.

**Table 20.** Abstract syntax of the parser rule for naming an attribute

```
ioAttribute
        :       CAPITALIZED_NOUN 'of' io
        ;
CAPITALIZED_NOUN : ('A'..'Z') ('a'..'z'|'A'..'Z'|'_')*;
```

The *submitUA* user action parser rule, which follows, denotes that the user submits the form, while the next rule *validateSA* defines a system action denoting that the system checks the values of the attributes, and if everything is correct, it saves the new information object instance through the parser rule *SaveSA* (Tables 13, 21 and concrete example of section 3.2).

**Table 21.** Abstract syntax of the parser rules for submitting, validating and saving the form data

```
submitUA :      'UA' INTEGER '.' primaryActor 'submits the form of the' io '.'      ;
validateSA :    'SA' INTEGER '.' 'The' 'System validates the attributes of the
                submitted form.         ;
saveSA  :       'SA' INTEGER '.' 'The System saves the form' '.'     ;
```

Furthermore, the system, through the parser rule *notifySA* (Tables 22 and 13), notifies the primary actor and any interested secondary actors about the creation of the information object. Step 10 of the example of section 3.2 below presents a concrete syntax of *notifySA*.

**Table 22.** Abstract syntax of the parser rule for notifying other actors about the creation of the IO

```
notifySA :      'SA' INTEGER '.' 'The System' notify io 'to the following' actors '.';

notify
        :       ('send' | 'sends') 'Notification' ('about' 'the' ('creation' |
'altering' | 'reading' | 'erasing'))? 'of'
        ;
```

Finally, the parser rule *exceptions* (Table 23) currently covers the case of incorrect input values. This parser rule will be expanded in the future to cover more cases of alternate and exception scenarios, especially regarding validation checks and errors.

**Table 23.** Abstract syntax of the parser rule exceptions.

```
    exceptions
        : 'Exceptions' ':' (INTEGER '.' INTEGER '.' (INTEGER '.')? exception)*
        ;
    exception
        :       ioAttribute 'is_incorrect' ':' 'Invalid' 'input' '.' ;
```

### 3.2. Example

Below is the concrete syntax of the use case *Create Prescription* developed and validated in ANTLR, based on the abstract syntax described in this paper.

**UC** 1 **create** Prescription =
**Description**: Doctor **creates** Prescription. Patient **and** Medical_system **provide or verify relevant information**. Pharmacist is **an** *Intended Recipient* **of** Prescription.
Pre-conditions: 1. Examination is at **Complete state**. Triggered by **UC** 11 **create** Examination.
2. Doctor is authorized.
**Actors**: Doctor (**type**: **creator**), Patient (**type**: **accompaniment**), Pharmacist (**type**: **intended recipient**)
**Main Flow**: **UA** 1. Doctor **requests to create** Prescription.
**SA** 2. **The system prompts** Doctor to **input the required and optional attribute values of** Prescription.
**UA** 3. Doctor **inputs** PatientID **of** Prescription.
**SA** 4. **The System validates** PatientID **of** Prescription.
**UA** 5. Doctor **inputs** Medicine1 **of** Prescription. [Extension point. **UC** 12 **Read** Examination report] [Extension point: **UC** 22 **Read** Counsellor Report]
**UA** 6. Doctor **inputs** Medicine2 **of** Prescription. [Extension point. **UC** 12 **Read** Examination report] [Extension point: **UC** 22 **Read** Counsellor Report]
**UA** 7. Doctor **submits the form of** Prescription.
**SA** 8. **The System validates the attributes of the submitted form**.
**SA** 9. **The System saves the form**.
**SA** 10. **The System sends Notification about the creation of** Prescription to **the** following **Actors**: Doctor, Patient, Pharmacist. [**via UC** 15 **send Notification of** Prescription]
**Exceptions**: 4.1. PatientID **of** Prescription **is_incorrect**: **Invalid input**.
8.5.1. Medicine1 **of** Prescription **is_incorrect**: **Invalid input**.
8.6.1. Medicine2 **of** Prescription **is_incorrect**: **Invalid input**
Post-conditions: 1. Prescription is at **Pending state**.

## 4. Conclusions and future work

This paper presented a grammar for the formalization of the use case model. The proposed grammar, which has been defined in EBNF, and developed and tested in ANTLR, provides syntactic and semantic rules for writing use case specifications in semi-formal natural language. Use cases can be written in ANTLR, in the form of concrete syntax, and are validated against the already defined abstract syntax, which has been described in this paper. One of the main aspects of this grammar is that it associates a use case with a specific action block, and according to the type of the action block (Create, Alter, Erase, Read), specific user and system actions, in a specific sequence, are used to form the main flow of the use case. Conclusively, the proposed grammar, within its powerful development and testing environment, helps to make expression of use case-based requirements more disciplined, understandable, well-formed, correct and complete, and it also makes easier their conversion into diagrammatic notations.

Future work will involve the extension of the presented grammar with transformation rules in order to automatically create use case and sequence diagrams. Furthermore, other platforms or languages will be explored for the representation and transformation of the proposed grammar, such as XML and XSLT. Also, XQuery could be used for querying and processing use case data, in order to come to useful conclusions, especially about metrics such as the number of actors and use cases of a project, or the average number of actions or steps regarding various projects. When these measurements are taken on a regular basis, they can be used for project estimation purposes. Additional future work will involve the expansion of the existing rules for the alternate and exception scenarios to cover more cases, especially regarding validation checks and errors. One more aspect that will be investigated is the enhancement of the proposed grammar with further rules or features that will allow the enrichment of the use case specifications (especially their scenarios) with supplementary details (e.g., about user interface or data model elements) important for the implementation of the use cases. Finally, future work will deal with

utilizing and testing the proposed grammar in several real-world projects, and the resulting use cases will be evaluated for correctness, completeness and consistency.

## References


1. Sinnig, D., Chalin, P., Khendek, F.: LTS semantics for use case models. ACM Symposium on Applied Computing (2009)
2. El-Attar, M., Miller, J.: Matching Antipatterns to Improve the Quality of Use Case Models. In: Proceeding of the 14th IEEE International Requirements Engineering Conference (RE'06), pp.99-108 (2006)
3. Eriksson, M., Börstler, K., Borg, K.: Marrying Features and Use Cases for Product Line Requirements Modeling of Embedded Systems. In: Proceedings of the Fourth Conference on Software Engineering Research and Practice (SERPS'04), Sweden, pp.73-82 (2004)
4. Leite, J., Rossi, G., Balaguer, M., Kaplan, G., Hadad, G., Oliveros, A.: Enhancing a Requirements Baseline with Scenarios. In Proceedings of Requirements Engineering, Annapolis, USA (1997)
5. Osaki, T., Kobayashi, A., Kato, T.: Writing Use-Case with a Minimal Set of Words. In Proceedings of the 5th IEEE/ACIS International Conference on Computer and Information Science, pp. 393–398, (2006)
6. Hsia, P., Samuel, J., Gao, J., Kung, D., Toyoshima, Y., Chen, C.: Formal approach to Scenario Analysis, IEEE Software, 11(2), 33-41 (1994)
7. Glinz, M.: An Integrated Formal Model of Scenarios Based on Statecharts. In Proceedings of 5th European Software Engineering Conference, Sitges, Spain, Springer (Lecture Notes in Computer Science 989), pp. 254-271 (1995)
8. Seybold, C., Meier, S., Glinz, M.: Scenario-driven modeling and validation of requirements models. In: 5th ICSE International Workshop on Scenarios and State Machines: Models, Algorithms and Tools, Shanghai, pp. 83-89 (2006)
9. Savic, D., Antovic, I., Vlajic, S., Stanojevic, V., Milic, M.: Language for Use Case Specification. In: IEEE 34th Software Engineering Workshop, Limerick, Ireland, pp. 19-26 (2011) doi: 10.1109/SEW.2011.9.
10. Savic, D., Vlajic, S., Lazarevic, S., Antović, I., Stanojevic, V., Milic, M., Silva, A.R.: Use Case Specification Using the SILABREQ Domain Specific Language. Comput. Informatics 34, 877-910 (2015)
11. Golbaz, M., Hasheminasab, A., Daneshpour, N.: An XML Definition Language to Support Use Case-Based Requirements Engineering. Proceeding of the International Multiconference of Engineers and Computer Scientists (IMECS 2008), Vol. 1 (2008)
12. Jacobson, I., Christerson, M., Johnsson, P., Overgaard, G.: Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley, Wokingham, England (1992)
13. ANTLRWorks: The ANTLR GUI Development Environment, https://www.antlr3.org/works/, Accessed February 2, 2024
14. Georgiades, M., Andreou, A.: Patterns for Use Case Context and Content. In: Favaro, J., Morisio, M. (eds) Safe and Secure Software Reuse. ICSR 2013. Lecture Notes in Computer Science, vol 7925, Springer, Berlin, Heidelberg (2013) https://doi.org/10.1007/978-3-642-38977-1_18
15. Cockburn, A.: Writing Effective Use Cases. Addison-Wesley. ISBN: 0201702258 (2000)
16. Georgiades, M., Andreou, A.: Formalizing and Automating Use Case Model Development. The Open Software Engineering Journal 6, 21-40 (2012)