Procasor Project

User Documentation

Procasor Project: User Documentation

Table of Contents

Acknowledgment

First of all we would like to record a special credit to Vladimír Mencl, whose paper [1],[2] was the inspiration of this project. Vladimir Mencl was also the supervisor of the project; his councils, friendly and patient attitude had a substantial influence on progress of the development.

Special thanks go to the Center for Computational Linguistics, in particular to Prof. Jan Hajic and Dr. Jan Curin for the help with putting the linguistic tools together.

Also we would like to thank our friends and families for their support.

Foreword

The Procasor project is a student Software Project at the Department of Software Engineering, Faculty of Mathematics and Physics, Charles University.

1. The Procasor Team

The Procasor team is the group of authors that cooperate on the development of the Procasor project.

Project Supervisor:

Vladimir Mencl , `<mencl@nenya.ms.mff.cuni.cz>`

Team Members:

Michal Fiedler , `<mfiedler@centrum.cz>`

Jan Francu , `<jenikff@centrum.cz>`

Jiri Ondrusek , `<gorduin@seznam.cz>`

Ales Plsek , `<aplsek@gmail.com>`

Contact Information:

Procasor team

Department of Software Engineering

Faculty of Mathematics and Physics, Charles University

Malostranske namesti 25

11800 Prague 1

Czech Republic

`<procasor@nenya.ms.mff.cuni.cz>`

http://nenya.ms.mff.cuni.cz/~mencl/procasor

Chapter 1. Brief Overview

Specifying requirements is the first step in the process of designing a software system or a component. For the task of such specification use cases written in natural language are usually used . While this method provides a gentle and swift way of creating an easily readable specification, use cases written in natural language can not be used in reasoning on requirement specification or employing the use cases in deriving an initial design in an automated way. Although the use cases contain a huge amount of relevant information about software system or component being specified, it is difficult to obtain a precise formal specification, for it is expressed in natural language.

The Procasor project attempts to harmonize the creation of requirement specifications employing use cases with creation of precise formal specifications of a software system or a component. To meet this goal, the Procasor includes an interactive environment which provides a comfortable development of a use case model and also the generation of a precise formal specification. Moreover Procasor allows to develop natural language use case model and precise formal specification at the same time, which demands only slightly increased effort required to write the use case model.

Procasor includes Linguistic analysis algorithms which employ integrated linguistic tools in order to analyse use cases. We implement the construction of behavior specifications in two notations - Behavior Protocols and UML State Machines. Procasor incorporates an interactive environment that provides user feedback during the process of the precise formal specification development.

1.1. How To Use This Book?

The goal of this book is to introduce all the features provided by Procasor. To reflect the goal, the book is organized as follows.

In the Background chapter [Chapter 2] we provide an explanation of the basic ideas and problems which are important to be understood before using Procasor.

The Procasor project chapter [Chapter 3] introduces all the ways the application can be used. This chapter is essential for understanding what the Procasor project can offer to the user.

Chapters Linguistic tools application [Chapter 4], Use Case Step Analysis [Chapter 5], Generating Token [Chapter 6] and Constructing behavior specifications [Chapter 7] explain individual parts of Procasor in detail.

User Interface [Chapter 8] describes features of the Procasor environment.

Installation manual can be found in [Appendix A].

Before the user starts using Procasor, we recommend to read chapters Getting started [Appendix B] and MarketPlace Project Example [Appendix C] as well. These chapters provide basic suggestions for working with Procasor and demonstrate functionality of Procasor on the Marketplace project example, included in the Procasor distribution.

For better orientation in this document, we provide also the Index [Appendix B].

Chapter 2. Background

2.1. Use Cases

The use case is the term in the software engineering, which is used beside the terms UML (Unified Modeling Language), entity, diagrams, scenario and many others. In this chapter we provide a gentle introduction into the use cases topics, mainly focused on the topics which are related to the Procasor program. For more complex information about UML and use cases we recommend an excellent book by A. Cockburn, Writing Effective Use Cases [3] and the UML website [4]. If you are familiar with use cases and UML you can skip this chapter, but we suggest you read it.

The use cases are a set of documents which are usually written when the requirement specification is gathered. The described system (*System under design, SuD*) will perform some tasks. In this process the system communicates with other entities - actors and other components. When performing the task, the system follows a scenario. For each task, we write a use cases to record the sequences of actions performed during the task.

First, we decide, which entities will act in the use case, we record them as the actors of the use case. Then, we write down the sequence of actions performed in case of successful execution of the task. We call this sequence the main success scenario. The individual actions are called use case steps. We give every step a number, according to its position in the sequence, we call this number a step label.

In addition, we collect branches, that reflect possible diversions from the typical task execution. For every branch, we define at which step of the main scenario the branch starts. We write down a condition, which describes the difference from the typical execution. Thereafter, we record the actions executed when the condition holds true, we call them branch steps. The condition gets a label consisting of the extended step label and a letter (e.g. 2a), the branch steps append an ordinal number to the step label of the condition (e.g. 2a1). We distinguish two types of branches - extensions and variations. Extensions represent actions performed in addition to the extended step, while variations are executed instead of the step.

A branch can be attached to more steps than only one. The steps can be specified by a range (the step label being e.g. 2-6a). All steps in main scenario can be represented by an asterisk (the step label being e.g. *a).

As an example, we will take a Marketplace Information system managing buying and selling the goods. With a use case, we will model the process that allows the user (Seller) to enter some item on market. The main scenario of this use case will contain describing the item, description validation, entering the price and contact and validation of the contact and verification users history and validation of the offer with Trade Commission. Then the system lists the offer and responds with unique authorization number. The step of validation of item's description will have an extension saying, that if the user wants to insert an item which is not permited on this market, the transaction is aborted. And the step of verification of item's description will have an extension saying, that if the user has an inappropriate history, the transaction is aborted. The use case for this example is written below.

Figure 2.1. Use Case Example

Use Case: Seller submits offer
Scope: Entering an item.
SuD: Marketplace Information System
Primary Actor: Seller
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 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.

Often, we describe the system from various level of detail. For example, we may first design the system as a whole, describe its communication with external entities (actors). In the second step, we may design individual components of the system, their internal actions, and communication between them in detail. In this manner, entities described in the process of requirement gathering create a hierarchy.

A single use case only describes the actions performed during a single task. We need to employ additional technique to specify the overall behavior of a component. In Procasor, use case expressions are used. These expressions serve for combining the information contained in more use cases. You can find more information about the use case expressions in Section 3.1 and in the Chapter 8.

This information is the starting stage for understanding the Procasor program. As you carry on with this user manual you will get on more advanced stages and you will understand more details of Use cases.

2.2. Specifying Component Behavior with Behavior Protocols - Pro-cases

Pro-cases [2], based on Behavior Protocols [5] are means of specifying the behavior of a component. For the described component, they enumerate all possible valid sequences of events.

Pro-case syntax stems from regular expressions. Atomic events emitted, absorbed or internally processed by an entity are the atomic expressions. These atomic events are written as follows:

- Event type is denoted with "!" for emitted event, "?" for absorbed event, "#" for internally processed event;
- Acronym of the active entity goes next, followed by a dot. Not present for internally processed events;
- Token of the event goes last. The token is a label, that describes the event, it is similar to a method name.

For example, an atomic event could be written as *?SL.submitItem*, meaning that this event is absorbed by the described entity. The entity with acronym *SL* is the source of this event, and that token of this event is *submitItem*.

Operators can be used to combine expressions into composite expressions. Among the operators are: * for repetition, ? for optionality, ; for sequencing, + for alternative, | for and-parallel execution, || for or-parallel execution (A || B meaning A + B + A|B). Other operators are not supported by Procasor.

A Pro-case, being an expression created in this manner, forms a regular language. There is a verifier tool available [6] for checking consistency of Pro-cases.

2.3. Deriving Pro-cases from Use Cases

In [2] F. Plasil and V. Mencl proposed a manual transformation from textual use cases into Pro-cases. In his subsequent work [1], V. Mencl presented a method for automation of this transformation. Transformation performed in this project is based on both of these proposals.

This transformation builds on two premises (quote from [2]):

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.

(end of quote)

To specify the Premise 2 more precisely, the sentence should follow the SVDPI (subject - verb - direct object - preposition - indirect object) pattern.

These premises guarantee, that we can identify the entity (actor or SuD), that is the subject of the sentence. Optionally, we identify the entity, that is the indirect object of the sentence. From this information, we can conclude, whether the step describes an event emitted, absorbed or internally processed.

Premise 2 alone warrants, that we can locate the main verb of the sentence and its representative object. We get the event token for this sentence by combining the verb and the representative object.

Now that we have the action type, active entity and the event token, we can combine them into an atomic event of a Pro-case. We connect subsequent steps, this connection is reflected by the sequencing operator ";" in the resulting Pro-case.

Extensions and variations (or branches for short) are connected to the main success scenario, this connection is reflected by the alternative operator "+" in the resulting Pro-case. We obtain an event token for the condition triggering the branch. Pro-cases do not support conditions, so we represent the condition with a special internal event. We use this condition event as the first event of the branch. The semantics is, that this condition event (and following events of the branch) can only be processed when the condition is fulfilled.

There's an important exception to the premises above. Special actions do not describe a communication, neither do they describe an internal action. They change the flow of control in the use case and this has to be reflected in the resulting Pro-case. Among others, special actions may be an abort action, or a goto action. Special actions are detected during the transformation, and the Pro-case is adjusted accordingly.

2.4. Linguistic Analysis

Analyzing use cases is the main goal of the Procasor project and because these use cases are written in natural language, our task is to process a natural language sentence in order to acquire information about an action expressed by this sentence.

Because of the complexity of natural language, it would by nearly impossible to analyze every sentence properly. Yet, in a use case, as defined in Section 2.1, every use case step is a simple structured sentence written in natural language describing either a communication action (among entities involved in the use case), or an internal action. This determines goals of linguistic analysis. For every sentence, we need to find action expressed by this sentence and to find all actors involved in this action, as well.

To meet this requirements we employ linguistic tools [Section 4.3] which should help us in the use case step analysis. The information we need for every sentence can be found in its parse tree [Section 4.2] and we use linguistic tools to obtain such parse tree for every use case step.

The parse tree of a natural language sentence captures the phrase structure of the sentence according to the respective natural language. That means, that a parse tree describes roles of all words involved in the sentence and contains basic forms of each word.

The parse tree therefore thoroughly describes each sentence and the object of the use case step analysis [Chapter 5] is to extract from the parse tree the needed information about action expressed by the sentence and actors that appear in this sentence.

Chapter 3. The Procasor project

This chapter introduces all the ways Procasor can be used. Reading of this chapter is essential for understanding what Procasor can offer to the user.

At first we provide a diagram [Figure 3.1], which illustrates all actions, that can be taken either by the user or by the Procasor itself. In following sections we describe these actions in more detail.

3.1. Writing Use Cases

Writing the use cases in the Procasor program has some specifics. Some steps should be done in certain order. In the first place you should understand all the terms about use cases, which were explained in the previous chapters, mainly in Section 2.1. In this section is explained, what you should do. How to do it, is described in Chapter 8.

Before you start writing the use cases, you should do some preparation. First you should make a list of entities, which will appear in the project. And set their hierarchical structure, which entity encloses others. Then it is very useful to write down the list of conceptual objects, which you will use in the project. The conceptual objects are objects which take a part in modeling project and are used when deriving information from sentences. Unlike entities, conceptual objects only act as names or symbols, not described objects. For example, if you are modeling a system for post office, then conceptual objects could be letter, package, telegraph, and others. Now when you have written down all the conceptual objects and entities, you are almost prepared for writing use cases. Use cases are grouped into use case models, every entity may have one or more models. The models are useful in a situation, when you want to describe the entities' behavior from different points of view. Usually you will only need one model, Procasor creates a default use case model for every entity. Once you have selected the model, you can create the use case.

The created use case always belongs to certain entity, this entity is called the System under Design, or SuD for short. Then you need to mark the entity which will act as the primary actor in the use case. Optionally, you can mark entities that will take part as secondary actors. The primary or secondary actor is the entity which is involved in the actions which are described in the use case.

When you have created the use case, you can write sentences as use case steps or as branch steps. A rule applies here, you should keep the sentences simple and unambiguous, so that it is clear what happens in the step. Moreover, readers should understand "where the ball goes" - which entities are involved and in which way the communication goes. In case of branch triggering conditions, you should not describe what happened, but what exactly the designed system detects. You can read more about effective use case writing in [3].

Use case model has a use case expression. This expression uses syntax described in Section 2.2. This expression has to be set, if you want to generate a model Pro-case [6].

If you write use cases following these instructions, then you can get meaningful Pro-cases. But if you will not guard these instructions, using this program could be wasting of your time.

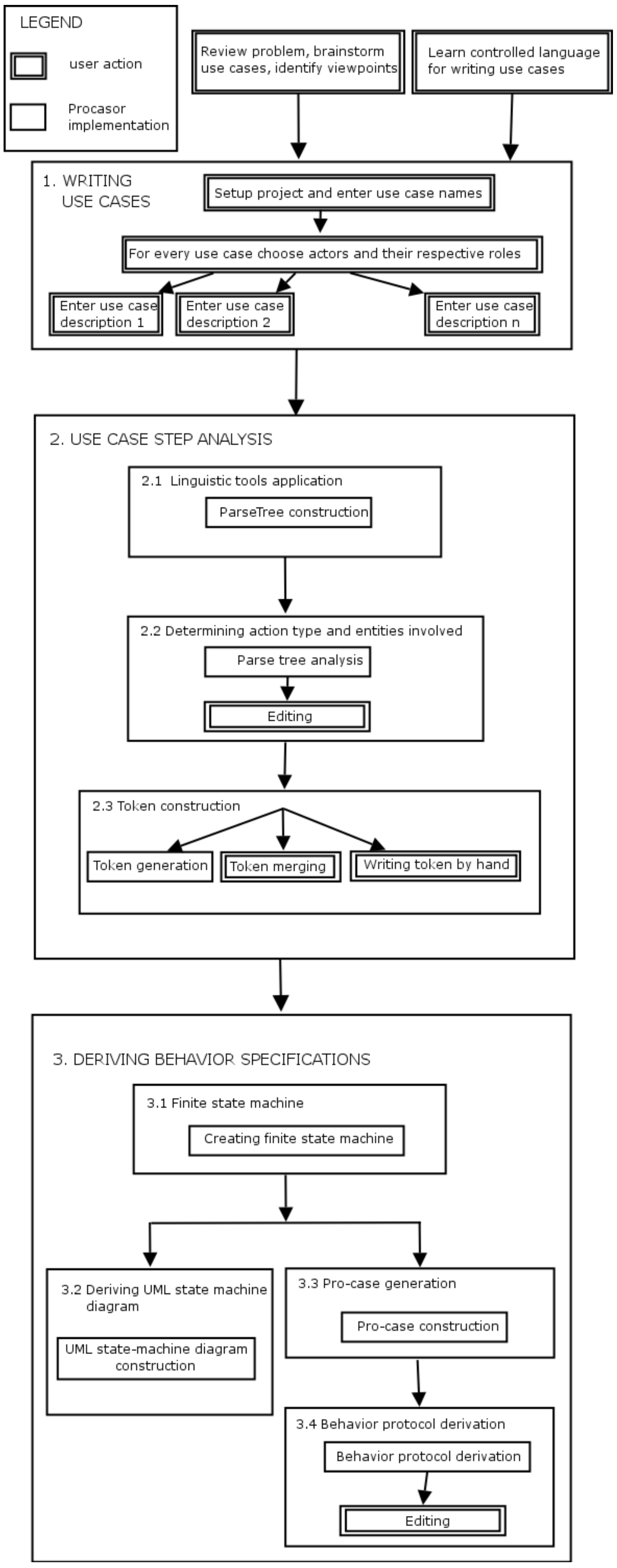
3.2. Use Case step analysis

When a step is entered, it is analysed in order to obtain its semantic information.

3.2.1. Constructing parse tree

Constructing parse tree is the first step of the linguistic analysis employed in Procasor [Section 2.4]. Parse tree [Section 4.2] is constructed by the integrated linguistic tools [Section 4.3]. The parse tree

Figure 3.1. The Procasor project engineering process model


LEGEND
user action
Procasor implementation
Review problem, brainstorm use cases, identify viewpoints
Learn controlled language for writing use cases
1. WRITING USE CASES
Setup project and enter use case names
For every use case choose actors and their respective roles
Enter use case description 1
Enter use case description 2
Enter use case description n
2. USE CASE STEP ANALYSIS
2.1 Linguistic tools application
ParseTree construction
2.2 Determining action type and entities involved
Parse tree analysis
Editing
2.3 Token construction
Token generation
Token merging
Writing token by hand
3. DERIVING BEHAVIOR SPECIFICATIONS
3.1 Finite state machine
Creating finite state machine
3.2 Deriving UML state machine diagram
UML state-machine diagram construction
3.3 Pro-case generation
Pro-case construction
3.4 Behavior protocol derivation
Behavior protocol derivation
Editing

obtained for every use case step is used by following processes of the use case step analysis in order to acquire required information about the action type of the analysed step.

In the process of parse tree construction no user interaction is required, the result of linguistic tools application is shown in the Parsed tree tab [Section 8.4.2].

3.2.2. Parse tree analysis

Based on the parse tree and additional information (actors, conceptual objects), program determines semantic information included in the sentence. This semantic information includes the action type (described in Section 2.2), entities involved in this sentence, the main verb and words that form the indirect object of the sentence. For special actions, semantic information can contain another entries.

For ordinary actions (request receiving or sending, internal actions), involved entities names, main verb and indirect object will be used to create the token of this sentence. For special actions, this semantic information will be used in the Pro-case construction.

Derived semantic information is shown in the Parsed tree tab (see Section 8.4.2) and you may change it there.

3.2.3. Generating tokens

The tokens are part of the action labels which are used instead of the whole sentences in Pro-case generating process, you can learn more about tokens and Pro-cases in Section 2.2. Procasor generates the token from words marked in the parse tree analysis. In addition, the user may choose to use another token, already used elsewhere in the project, or to write the token by hand. To learn more about this topic you can read the Chapter 6, where you will find more information on the token generating.

3.3. Deriving behavior specifications from Use Cases

Procasor provides two kinds of behavior specification - Pro-cases and UML State Machines. You can derive behavior specification on various levels: Pro-cases and UML State Machines for single use cases, model Pro-cases for use case models, or the complete set of UML State Machines for the whole project.

3.3.1. Pro-case generation

Procasor can transform a use case into a Pro-case (defined in Section 2.2). For this transformation to be successful, every step of the use case must have a valid semantic information.

You can create a Pro-case by opening the Pro-case tab in. Pro-case is a part of the text export of a use case (and of the whole project), as well.

3.3.2. Model Pro-case generation.

For a use case model, Procasor can take its use case expression and replace use case acronyms with appropriate Pro-cases, resulting in a model Pro-case.

3.3.3. UML State Machine generation

UML State Machines are another way to specify the behavior of a component. A UML State Machine consists of states and transitions and defines their layout in a diagram.

With Procasor, you can either create a UML State Machine for a single use case, or you can export the whole project to UML with state machines for all use cases. You can display and edit these UML diagrams with your favorite UML modelling tool.

Chapter 4. Linguistic tools application

4.1. Introduction

This chapter describes linguistic tools and their integration into the use case analysis. At first we introduce goals of linguistic analysis in Procasor, after that we define structure for passing results of linguistic analysis - the parse tree, and finally we describe individual tools and their roles in these processes.

Application of linguistic tools is the first step of use case analysis. Each sentence representing one use case step is sequentially processed by these tools in order to acquire information about structure of the sentence and role of each word in the sentence. This information is stored in a structure called parse tree. The parse tree for the sentence is then passed to following processes which take part in use case analysis.

Acquiring a parse tree of a natural language sentence is the main goal of the application of linguistic tools used in Procasor.

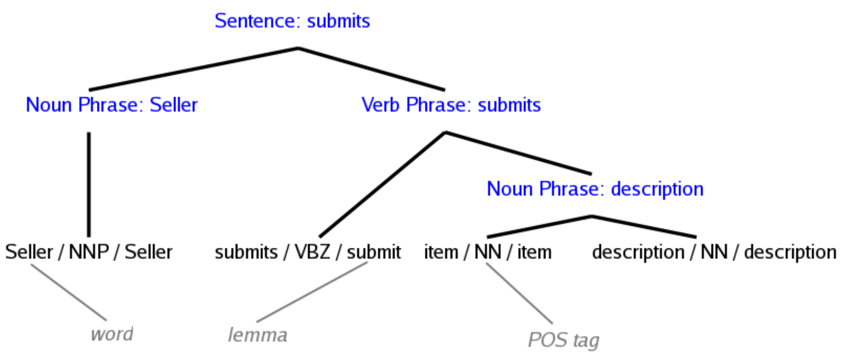
4.2. Parse Tree

Results of linguistic analysis are stored in a structure called parse tree. The parse tree of a natural language sentence captures the structure of the sentence according to the grammar of the respective natural language - English in this case. Leaves of the tree reflect the words of the sentence (in left-right order), while intermediary nodes represent phrases. [Figure 4.1] shows the parse tree obtained for sentence: "Seller submits item description".

Each intermediary node represents a phrase formed by words in its leaves. Information involved in this node consists of type of the phrase and the headword of the represented phrase. E.g. in [Figure 4.1] the nouns "item" and "description" constitute a *noun phrase* (denoted NPB), which together with the verb "submits" form a *verb phrase* (VP). The headword representing the verb phrase is "submits".

While intermediary nodes represents phrases, leaves reflect the words of the sentence. Each leaf responds to a word contained in the sentence. Leaf also includes additional information about the word it represents - *POS*-tag and *lemma*. POS-tag (part-of-speech tag) determines the word type and the role it plays in the phrase structure. Lemma is the base form of the word. E.g. in Fig. 1, the POS-tag of "submits" is VBZ (verb in the "s" form), "item" and "description" are nouns (NN). Lemma of "submits" is "submit".

Figure 4.1. Parse tree for sentence: *Seller submits item description.*



4.3. Linguistic Tools Used

Obtaining the lemma and determining the POS-tag of a word along with acquiring a parse tree from a sentence are key tasks of linguistic analysis employed in Procasor.

To meet these requirements Procasor employs the linguistic tools which cooperate on analyzing the sentence:

- Tokenizer

  Tokenizer is used for preprocessing of the sentence before it is passed to tagger.

- MXPost tagger

  The third tool is **MXPost tagger**, we use the tagger for assigning POS-tags.

- Collins parser

  The key linguistic tool employed in Procasor is the **Collins parser**. It is a statistical parser developed by M. Collins at the University of Pennsylvania and we use this tool for obtaining a parse tree of the sentence. Collins parser was chosen for its high accuracy and for the robustness of its parsing algorithm. However integration of this tool increases hardware requirements and specially initialization process is time-demanding. The user should have at least 200MB of free memory to run the parser.

- Morph-a tool

  Morph-a tool is a morphological tool developed at the University of Sussex to obtain the *lemma* (base form) of a word.

4.4. Using Linguistic Tools

Although the application of linguistic tools is managed automatically, we introduce some basic issues concerning usage of this tools in this chapter. We describe the initialization and parsing processes of linguistic tools visible for the user and provide the information about error situations and their solutions. For more details, please see the Developer's documentation.

4.4.1. Initialization

As mentioned in [Section 4.3], the initialization process of linguistic tools is time-demanding and therefore using of this tools could be uncomfortable. Initialization of the Collins parser is very slow, in particular. We solve this problem by running initialization processes in background, which provides the opportunity to use the application without the need to wait until linguistic tools initialization is completed.

4.4.2. Linguistic Analysis

Linguistic analysis is a part of use case analysis and therefore the user does not have to maintain processes which take part in sentence processing. The results acquired from linguistic tools are automatically collected and passed in the form of a parse tree which is displayed graphically. User is informed about particular steps of linguistic analysis by short messages which are displayed in the Parser Console.

The part of the application which provides the parse tree generation is called the **Parser**. The Parser is represented on the user's screen by the Parser Console [Section 8.2.2.2]. The Parser console shows messages describing the status of the parse tree generation process. Initialization messages from linguistic tools are also printed on this console. Through this console the user can be informed about unexpected and error situations that occur in linguistic tools.

4.4.3. Restarting the Parser

The process of parse tree generation is complicated and is provided by different tools witch have to be in compliance with each other, therefore the application needs to confront and solve unexpected situations and error behavior of the linguistic tools.

The errors that occur in linguistic tools are detected and printed on to the Parser console. Although we can detect an error occurrence, it is very difficult to resolve these errors or to prevent them. Therefore we have chosen to restart the Parser when an error occurs and than we process the sentence again.

In the case an error occurs, the remaining linguistic tools are closed and all linguistic processes are initialized again. The user is notified about the error by the Parser console, which signalizes error occurrence by the red color, the option to restart the parser is immediately offered. We call the situation, when an error occurs a **Broken Pipeline** situation.

Linguistic tools processes are monitored by the Parser and important information and messages are stored in the log file, which can be useful when searching for reasons of error occurrences. In the case an error occurs, please, send email with appended log file to the Procasor team (Section 1). The log messages are appended to the "logs/pipeLine.log" file, detailed information about this log file and its structure is available in the Developer documentation.

4.4.4. Running stand-alone linguistic tools

The distribution of Procasor contains all linguistic tools used by Procasor. In addition, we provide the opportunity to use linguistic tools that are not part of our application. For example the user can run the Collins parser, which has been installed to a detached location and is not part of Procasor distribution. For details about this option, please see the chapter about the Configuration file in the Developer documentation.

Chapter 5. Use Case Step Analysis

Common step analysis is applied on common (non-condition) steps. If common step analysis is not successful, we try to detect a special action. Condition analysis is applied on branch triggering conditions.

5.1. Common Step Analysis

For a common step the semantic information is derived as follows.

First, we detect the subject. A sentence conforming to the SVDPI (subject - verb - direct object - preposition - indirect object) pattern should contain a noun phrase and a verb phrase. This main noun phrase is the subject. As the subject has to be an entity, we match nouns in the subject phrase with names of actors involved in this use case. Moreover, term "System" may refer to SuD and term "User" refers to the primary actor of the use case.

If subject is not found, we try to detect a special action. If the step is not even a special action, the tree analysis fails. Possible reasons are, that the sentence was either badly written, or that the linguistic tools have failed.

If SuD is the subject, we check whether an actor is the indirect object of the sentence. We consider all noun phrase subordinate to the main verb phrase to be indirect object candidates. For every candidate, all included nouns are gathered and matched with actor names. We do not allow the indirect object to be in possessive case.

If SuD is the subject and indirect object is found, the step describes a request sent by SuD to the indirect object actor. If indirect object is not found, the step is an internal action. If an actor is the subject, the step describes receiving a request from this actor (from the view of SuD).

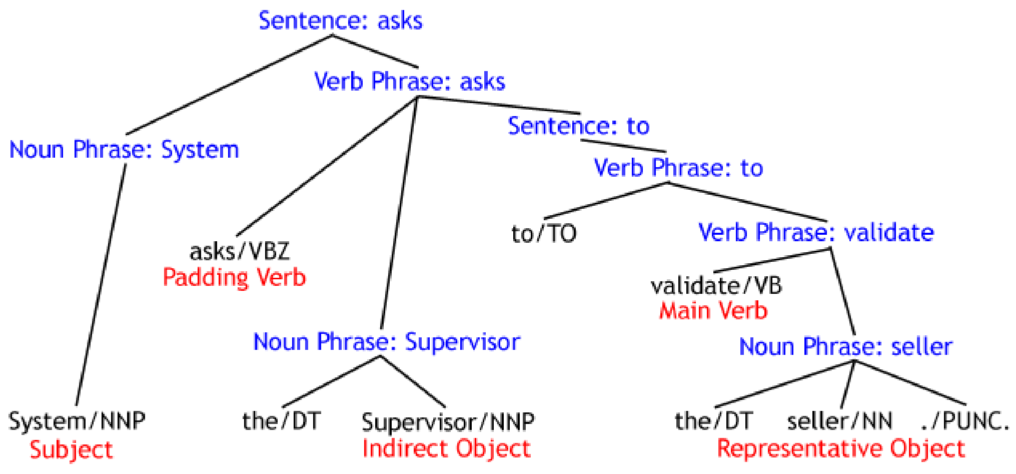
The main verb is the first full verb (verb that carries the meaning) in the main verb phrase. If the first verb is a padding verb (one of "ask", "be", "decide to" etc.) and there's another verb following it, we use the second verb. Otherwise, the first verb is recognized as the principal verb.

Excluding phrases used for subject and indirect object, the first basic noun phrase subordinate to the main verb phrase is identified as the indirect object. All nouns from this phrase are collected and matched with the conceptual object list. If a name of some conceptual object is matched, only the matched words are used as the representative object. Otherwise, all nouns from this phrase are used.

For an example, we will take the sentence "System asks the Supervisor to validate the seller". This sentence is a step of the use case called "Clerk submits an offer on behalf of a Seller" (you can find this use case in the Marketplace project). Computer system is the System under Design, Clerk is the primary actor and Trade Commission, Supervisor and Seller are supporting actors. You can see the parse tree for this sentence in the following figure.

Figure 5.1. Parse Tree Analysis Example

Analysis of the sentence "System asks the Supervisor to validate the seller"



The word "System" is the subject of this sentence. It is a keyword referring to the SuD. Because SuD is the subject of the sentence, we search for an indirect object. We inspect two phrases, "the Supervisor" and "the seller." and detect, that Supervisor is the indirect object of the sentence. From this information, we conclude that this step represents a request sending action.

Afterwards, we identify the main verb. "Asks" is the first verb in the sentence, but it may be a padding verb. Therefore, we search for another verb and find one - "validate". "Validate" is the principal verb, while "asks" is a padding verb. To create the representative object we use the phrase "the seller.", which is the only noun phrase under the main verb. Matching with the conceptual object list (item, offer, price assessment, price, payment method) is not successful, so we mark "seller" as the representative object.

5.2. Detecting Special Actions

A special action does not describe an operation request, instead, it describes a change in the flow of a use case. A special action should only occur in extensions and variations as their last step. We distinguish two basic types of special actions - terminate actions and goto actions. Another special action, including a nested use case, is not supported by the Tree Analyser and must be created manually. To be recognized as a special action, sentence must follow a typical pattern, as described in following paragraphs.

Terminate action may terminate the use case or only the branch (extension or variation). It may also be an aborting action, signaling a failure. For a terminate action to be identified, verb must come from a list of triggering words ("end", "terminate" or "abort", the last identifying an aborting action). At the same time, subject must be one of a few keywords ("Use case", "Extension" etc.) or the sentence must have no subject.

A sentence containing a goto action mostly has no subject, so we do not even try to identify it. Instead, we search the sentence for a verb from a keyword list ("resume", "continue" etc.) and for a step label. A word is identified as a step label if it starts and ends with a digit.

5.3. Condition Analysis

When analysing a condition, we only decide whether each word is important and, thus, should become part of the condition's token. The word is only marked as not important, if it's one of the following:

- a preposition
- punctuation

- a "to" word
- determiner "any" or "the"
- a padding verb

Otherwise, the word is considered important and will be used for creating the token of the condition.

Chapter 6. Generating Tokens

This chapter will give you a deeper look inside the management of the token generating process. In the token generation and management we work with several terms which are crucial to understand the whole topic.

Action label	label containing step's information about • type of the action • the actor • token
selected token	token which will be used for generating the Pro-case

6.1. Preliminary

The tokens and action labels are generated from the information provided by the tree analysis and linguistic analysis. From previous analysis, we already know the type of the action, subject or/and indirect object and the words which should be used for building the token. These are the main verb and the representative object.

The example sentence "Seller submits item description." is analyzed as request receive action and the marked words are:

- Seller - noun - SuD
- submits - verb - main verb
- item - noun - word which is important and should be used for building token
- description - noun - word which is important and should be used for building token

6.2. Main steps of token generating process

The whole token management process consists of three steps.

- Simple generating - first simple token generating, the product of generating process is a simple token
- Learning process - token customization based on other tokens, the product of learning process is a learned token
- Manual changes - performed by user, the product of manual changes is the manual token

Usual scenario should only employ simple generating (described with an example in next paragraph). If there already is a similar token in the active project, we analyze this similar token and try to build the new token according to the analyzed information. The similarity rate can be set by the user in the "Percents of similarity to perform token learning" setting. This newly build token we call the learned token and this process the Learning process. If the information is sufficient for building the learned token, then the learned token is used instead of the simple token. More information about the learning process is in Developer Documentation in the section Token Generation in subsection Learning token.

The simple token generating process concatenate words marked as important for building token. The words are concatenated in the order in which they were used in the sentence. For example, the simple token generating process produce for the sentence "Seller submits item description." following action label.

?	type of action, sentence action is analysed as request receive action
SL	actor, acronym of the entity "Seller", which was marked as subject.
.	the dot which separate the actor and the token
submitItemDescritption	token, concatenated words, which were marked as important for building token

The learning process will be illustrated on another example sentence. Let the action label with token "SL.propertiesSet" be already used in the project and you are writing the sentence "Seller get the properties.". Then you can expect the action label with token "SL.propertiesGet", but the simple generating process produce action label with token "SL.getProperties". If the token "propertiesSet" from the action label already used in the project is similar to the token from the actual sentence "getProperies". Then the process of learning token is applied and produce token "SL.propertiesGet".

6.3. Manual changes

If the user is not satisfied with the selected token, there are three ways of changing the token.

- Merge the token with another token from the project.
- Mix the words in the sentence to compose the token.
- Write the token by hand.

User can write similar sentences for the same action but the slight difference may still lead to creating different tokens. Therefore, the user can change the selected token in one of these ways. Merging the token with another token used in the project let the user keep the tokens in project consistent. By consistence, we mean that in the project the steps that refer to the same action should have the same selected token.

The second way of manually changing the selected token is mixing the token from words contained in the step's sentence. This is useful in the situation when the words used in the selected token are not in the proper order. The selected token may contain some words which the user does not want to be used in selected token, or the selected token does not contain some important word which user wants to use in the selected token.

The third way is to write the token by hand. This should be the least common way of changing the selected token. There is no need to keep tokens short, instead, the token should contain enough information about the action to which it refers to. The use case sentence contains the description of the action and therefore the selected token contains the description of the action, too. But in some unpredictable cases could Procasor's analysis fail and then user can write the token by hand. This option may also come useful, when you can't run the parser for some reason.

6.4. How to manage tokens

This section gives you the meaning how to use token generating process for you purpose. The details how you can do it, you can find in chapter User interface [Chapter 8].

When you have written and parsed the sentence, the selected token is generated and displayed. First is generated the simple token. If the generation of learned token success the learned token is used as selected instead of simple token. In some cases the simple token generation fails. This is due to fact that some required information were unprovided by previous analysis. In this case is better to reformulate the sentence and parse it again and if it does not help then you can change token manually.

If you are not satisfied with the selected token or the Procasor's analysis fails you could change token manually [Section 6.3]. Mainly in the beginning of the project modeling process you may correct the order of the words used in selected token by mixing them together. When you have several use cases written in the project then the amount of the selected token have populated. Then you may use token merging to ensure that the same action should have same selected token in all use cases.

Chapter 7. Constructing Behavior Specifications

Procasor provides derivation of two kinds of behavior specification. You can choose between Pro-cases and UML State Machines. When deriving behavior specification, Procasor uses previously generated semantic information (as described in Chapter 5 and Chapter 6). For both Pro-case and UML State Machine generation, a finite state machine is generated first. From this finite state machine, the behavior specification itself is derived.

7.1. Creating a Finite State Machine

Note

This part is quite technical and difficult. Still, you are encouraged to read it. Knowledge of the rules that direct finite state machine creation may help you to write better use cases, that result in accurate behavior specifications.

We create a finite state machine (FSM). States in this FSM represent the beginnings and ends of events. Transitions with a value represent events (event token being the value), void transitions represent a possibility, that an event follows the previous one.

First, the main success scenario is processed. In addition to its states, the initial state and the final state of the FSM are generated and connected to the first and last event.

Then, extension and variation (we call them branches) steps are processed. The head of each branch is connected to the state they extend. This transition is marked with the condition event of the branch. There is a difference between extensions and variations. Extensions are appended to the end of the parent event (making an optional continuation of the parent event), while variations are appended to the beginning (making an alternative to the parent event). End of the branch is connected to the beginning of the event following the parent event, giving command back to the upper level. Branches may be connected to multiple parents. In that case, a special copy of the branches states and transitions is made for every parent.

Special actions do not represent an event and must be transformed differently. Termination and goto actions are represented as transitions to the relevant state.

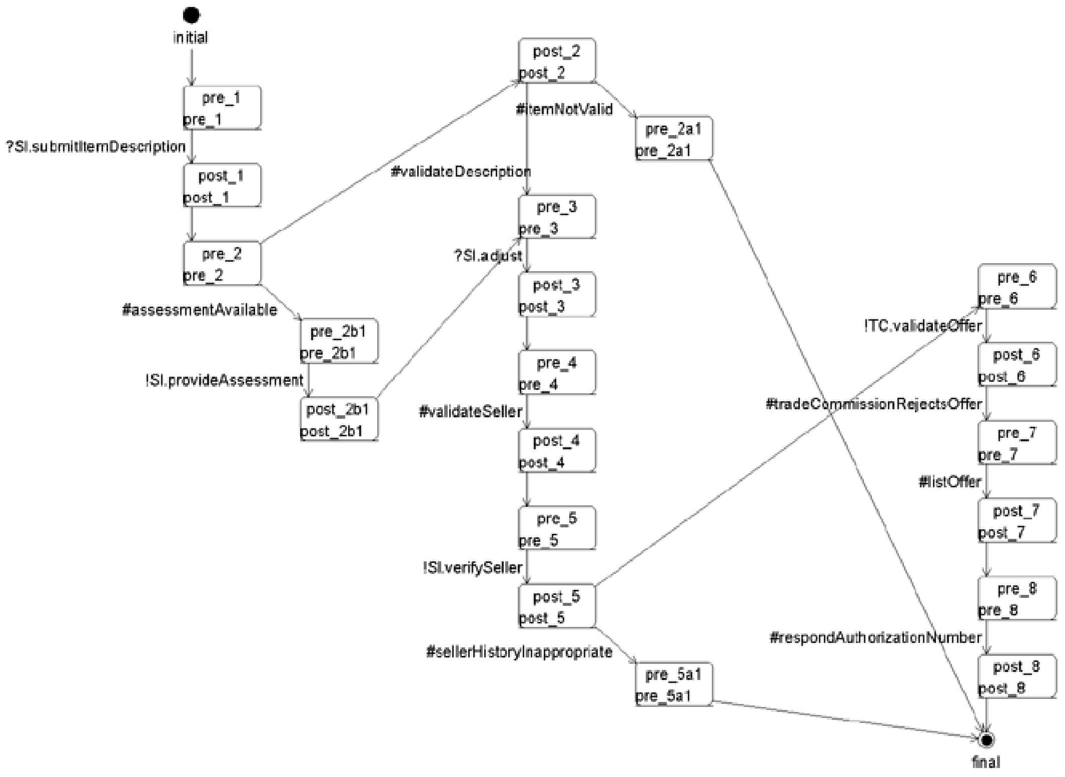
Include use case action results in creating another finite state machine (using the same method) and connecting it to the surrounding states. In addition, if this use case including step has a special variation of type "Handles included use case failure", the included use case remembers this variation as its abort handler.

Abort action works the same as termination action, if not inside an included use case. If there is an abort action inside an included use case, another transition is created. This transition connects the in-cluded use case and its abort handler and bears the value of the condition event, that triggered the abort action. If the included use case has no abort handler, the transition is connected to the parent machine's final state and the same is repeated, until an abort handler is reached or untli we reach the topmost FSM.

Following figure shows a state machine for the use case "Seller submits offer". You can see the original use case in the Figure 2.1.

Figure 7.1. State Machine Example

State machine for the use case "Seller submits offer"



7.2. Creating a Pro-case

To acquire a Pro-case from the previously generated finite state machine, we employ the algorithm for deriving an equivalent regular expression (described in [11]). If there are any included machines, we process them at first, using the same algorithm. After processing them, we replace the FSM by two states and a transition with a value set to the derived expression. Pro-cases derived in this manner only use several expression operators (alternative, repetition, sequence).

7.3. Creating a Model Pro-case

In order to generate the model procase, we parse the use case expression. Operators and brackets are indented to preserve readability. When a use case acronym is encountered, it is replaced by a Pro-case generated from the relevant use case.

7.4. Creating a UML State Machine

For the UML State Machine we compute states positions in the diagram. Generated UML document contains a UML State Machine with its states, transitions, diagram layout information, and UML class entry for the entity that owns the original use case. If you choose to export the whole project to UML, the resulting document will contain classes for all entities and state machines for all use cases.

The UML document is saved in the XMI format, which is an XML notation for writing UML documents. You may open and edit this UML model using your favorite UML modelling tool (we recommend the Poseidon tool [13]).

Figure 7.2. Pro-case Example

Resulting Pro-case for the use case "Seller submits offer"

```
?Sl.submitItemDescription;
#validateDescription;
#itemNotValid
+
(
   ?Sl.submitItemDescription;
   #assessmentAvailable;
   !Sl.provideAssessment
   +
   ?Sl.submitItemDescription;
   #validateDescription
);
?Sl.adjust;
#validateSeller;
!Sl.verifySeller;
(
   !TC.validateOffer;
   (
      NULL
      +
      #tradeCommissionRejectsOffer
   );
   #listOffer;
   #respondAuthorizationNumber
   +
   #sellerHistoryInappropriate
)
```

Chapter 8. User Interface

8.1. Introduction

This chapter provides an overview of Procasor graphical user interface. Especially non-obvious features resulting from generally recommended rules for writing use cases are mentioned. For full understanding, previous chapters are supposed to be read. Particularly Chapter 2 and Chapter 3.

Procasor graphical interface is designed for simple and intuitive creation of the use cases. It provides access to all features described in previous chapters.

To run Procasor, please read the Appendix A carefully and follow its instructions.

8.2. Main Window

When Procasor starts, main window is displayed. Individual windows will take places inside of this main window. The bottom bar consists of linguistic tools information field and of the message area. Main menu is located in the top of the window.

8.2.1. Main Menu

Main menu serves to simplify project editing, parsing tools control and preferences setting. Most important elements from the main menu have their own shortcuts. Menu consists of four items: Project, Parser, Window and About. Description of the individual items follows:

- Project menu.
  - New (Ctrl N) - To create new project. Displays the new project dialog. Project's name has to be filled.
  - Open (Ctrl O) - To open existing project. If the newly opened project contains unanalysed steps and linguistic tools are running or starting, user may choose to analyse the steps.
  - Close (Ctrl L) - To close the active project.
  - Save (Ctrl S) - To save the active project. If project has not been saved yet, the "Save as" dialog appears. Files are saved in folder "projects" by default. Procasor projects use the extension ".pro".
  - Save as - To save the active project into a different file.
  - Rename - To change the name of the project.
  - Export project to txt - To open the exporting window. The exporting window contains the textual export of the project. This textual export can be modified and saved to a file.
  - Export project to UML - To export the project information to UML (Section 7.4) and save it in the chosen file.
  - Close - (Ctrl X) - To close Procasor. Closing Procasor also terminates linguistic tools (if they are running or starting).
- Parser menu

  Parser menu controls states of linguistic tools.

  - Start parser - To start linguistic tools. When starting linguistic tools, all sentences waiting for analysis are removed from queue. Look at Section 8.4.1 for more information about queue.
  - Close parser - To stop linguistic tools.
  - Restart parser - To restart linguistic tools.
  - Show parser console - To show the Parser console [Section 8.2.2.2].
  - Close parser console - To close the Parser console [Section 8.2.2.2].
- Window menu
  - Show hierarchy - (Ctrl H) - To show the Hierarchy panel [Section 8.7].
  - Show token manager - (Ctrl K) - To show the Token manager window [Section 8.6].
  - Properties - (Ctrl T) - To show Properties window [Section 8.5].
- About menu

  To show the basic facts about the Procasor project.

8.2.2. Linguistic Tools Control Centre

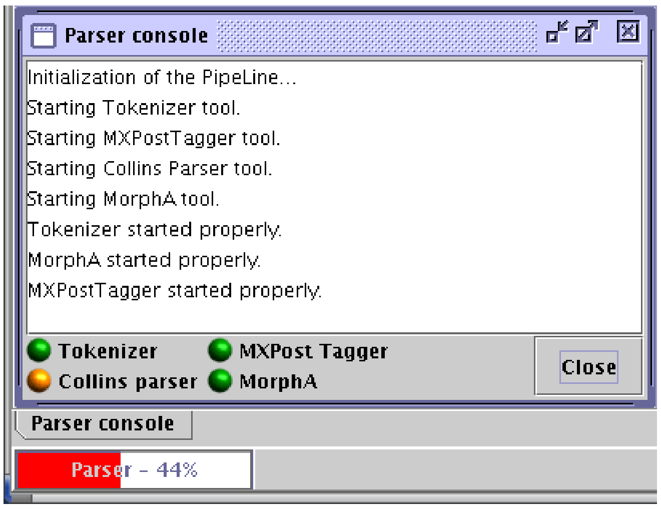
In following text, the group of linguistic tools will be called "Parser". Linguistic Tools Control Centre is shown in Figure 8.1.

In the right lower corner there is the Parser progress bar. Progress bar displays the state of the parser:

- Green bar - Parser is running.
- Red bar - Parser is closed.
- Progress bar with percentage description - Parser is starting.

Clicking the bar with right button brings up parser popup menu, left clicking displays the Parser console.

Figure 8.1. Parser console screenshot



```
Initialization of the PipeLine...
Starting Tokenizer tool.
Starting MXPostTagger tool.
Starting Collins Parser tool.
Starting MorphA tool.
Tokenizer started properly.
MorphA started properly.
MXPostTagger started properly.
```

8.2.2.1. Parser Popup Menu

Parser popup menu allows you to start or close parser and brings Parser console. The parser popup menu has the same items as the Parser menu in the Section 8.2.1.

8.2.2.2. Parser Console

Parser console displays information messages sent by linguistic tools and shows states of each tool separately.

The main part of the console is the section in which linguistic tools messages are displayed. Individual linguistic tool messages are described in Developer documentation in detail. At the bottom of the Parser console there are four icons, that represent the states of linguistic tools: Tokenizer, Collins parser, MXPost tagger and MorphA.

Possible states are:

- 🔴 Tool is closed.
- 🟠 Tool is starting.
- 🟢 Tool is running.

8.3. Project Window

After creating a new project (or opening an existing one) Project window appears. Project window for the Marketplace project example is shown in Figure 8.2.

Most of the window displays entities, their models and use cases and provides means to manage them. The right column contains the list of project's conceptual objects [Chapter 3].

8.3.1. Entity Management

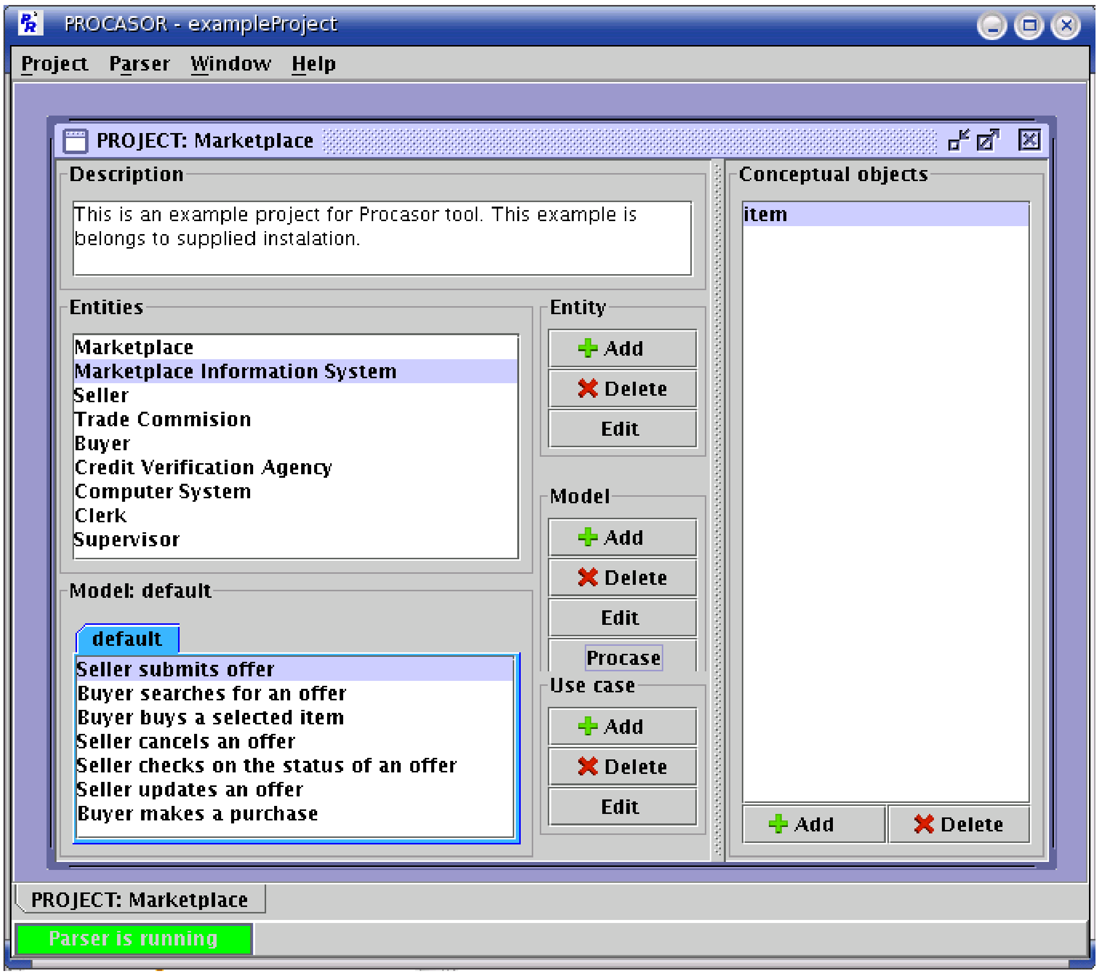
Entities contained in the project are shown in the upper part of the Project window. Three buttons are available: Add, Delete and Edit. Clicking "Add" or "Edit" allow to edit entity properties. Fields "Entity name" and "Acronym" have to be filled and have to be unique in the scope of the project. The acronym has to start with a letter and contain letters of digits only.

Lower part of the window provides access to the entities hierarchy. Parent and/or children entity may be chosen here. Hierarchy of all entities can also be accessed using the Hierarchy window [Section 8.7].

8.3.2. Model Management

Every newly created entity contains one model, named "default". For adding or editing models similar buttons as for entity editing are available. Model editing/adding window is similar to the entity editing window. Instead of the acronym, this window contains the use case expression. The use case expression describes possible use case sequences (see Chapter 3). For easier creating of use case expressions, the Expression editor is provided. Editor allows only allows to insert valid symbols and acronyms, by clicking appropriate buttons [Figure 8.3].

Figure 8.2. Project window screenshot



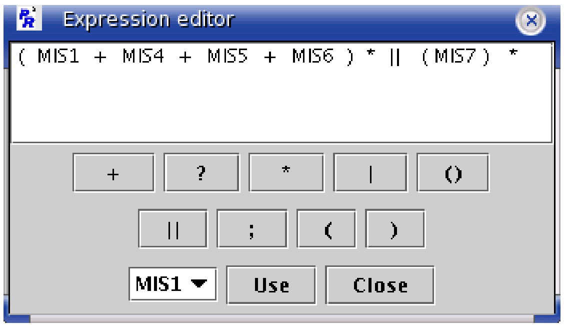
This is an example project for Procasor tool. This example is belongs to supplied instalation.

Marketplace
Marketplace Information System
Seller
Trade Commision
Buyer
Credit Verification Agency
Computer System
Clerk
Supervisor

item

Seller submits offer
Buyer searches for an offer
Buyer buys a selected item
Seller cancels an offer
Seller checks on the status of an offer
Seller updates an offer
Buyer makes a purchase

Figure 8.3. Expression editor



(MIS1 + MIS4 + MIS5 + MIS6) * || (MIS7) *

If model expression for the active use case model is valid, the "Pro-case" button displays a window with generated model Pro-case. The model Pro-case can be modified and saved to a file.

8.3.3. Use Case Management

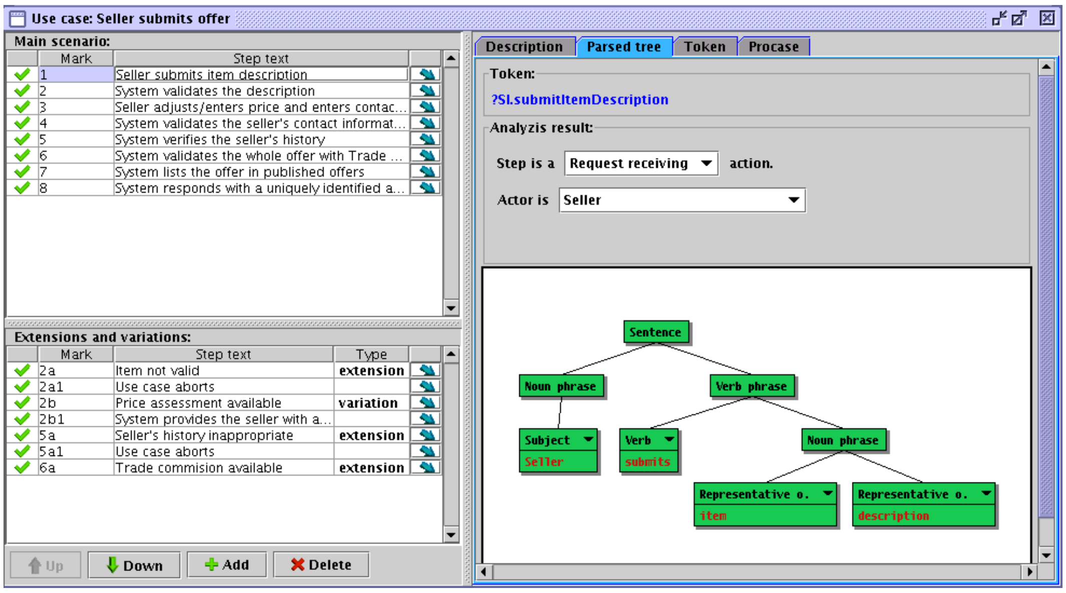
Use cases are displayed in the lower part of the Project window. When adding or editing a use case, name and acronym have to be filled. Acronyms have to fulfil the same conditions as model acronyms. They have to start with a letter and contain letters and digits only.

Entities acting in the use case can be chosen in use case editing window. First entity - System under Design - is the owner of the use case and is added automatically. Other entities may be added as the primary actor or secondary actors. Only one primary actor is allowed. Clicking "Edit" button shows the most important window for use case writing, the Use case window.

8.4. Use Case Window

The window allows writing steps of a use case and analysing them. The window is divided into two parts. The left one contains the steps of the use case, the right one contains tabs that display and manage analysed information. Use case window is shown in Figure 8.4.

Figure 8.4. Use case window



Use case: Seller submits offer

Main scenario:

	Mark	Step text
✓	1	Seller submits item description
✓	2	System validates the description
✓	3	Seller adjusts/enters price and enters contac...
✓	4	System validates the seller's contact informat...
✓	5	System verifies the seller's history
✓	6	System validates the whole offer with Trade ...
✓	7	System lists the offer in published offers
✓	8	System responds with a uniquely identified a...

Extensions and variations:

	Mark	Step text	Type
✓	2a	Item not valid	extension
✓	2a1	Use case aborts	
✓	2b	Price assessment available	variation
✓	2b1	System provides the seller with a...	
✓	5a	Seller's history inappropriate	extension
✓	5a1	Use case aborts	
✓	6a	Trade commision available	extension

Token:
?SI.submitItemDescription

Analyzis result:

Step is a Request receiving action.

Actor is Seller

8.4.1. Use Case Tables

Two tables are situated here. The first table serves for writing steps of the main scenario; second table contains extension and sub-variation steps of the use case. The behavior of these tables depends on the current settings of Procasor. See Section 8.5 for detail.

The first column of the tables displays the state of the analysis for the sentence.

Possible states are:

- ● New sentence
- ✓ Correctly analysed sentence
- ✓ Sentence waiting for analysis. The sentence was sent to the parser.
- ✖ Analysis failed for this sentence
- ❗ Sentence has a conflict. See below.

Star appears next to the symbol, if the text has just been changed.

The second column contains step marks. Red box around the field means, that something with the step mark is wrong. Keep the cursor upon the field to see a tooltip message with detailed explanation.

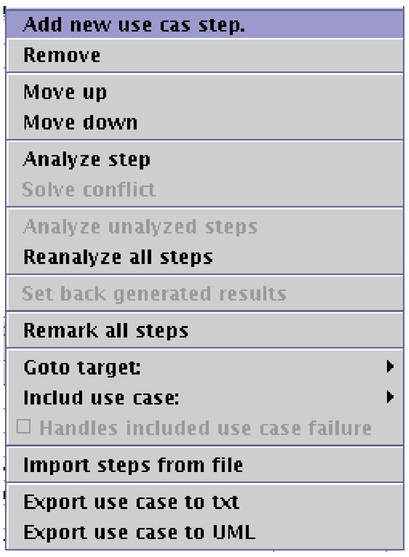
The third column contains the step text. Every confirmed change in text of step triggers analysis of this step. Step is enqueued and waits for analysis. Restarting parser empties this queue. Step, that was manually changed and is newly analysed, comes to state called conflict. Conflict can be solved in Analysis tab [Section 8.4.2.2], where you can choose between the newly generated result and the last manually changed result.

Button located in the fourth column create new steps extending the selected step (e.g. for step "1", new step "1a" is created in the lower table).

You can navigate through these tables by pressing up and down arrow keys. Pressing enter triggers various actions depending on the Batch mode setting [Section 8.5].

Use case popup menu [Figure 8.5] is displayed when right clicking on a step table.

Figure 8.5. Use case popup menu



- Add new use case step - Inserts a new empty step at the selected position.
- Remove - Removes the selected step.
- Move up - Moves the selected step one row above.
- Move down - Moves the selected step one row below.
- Analyse step - Starts analysis of the step.
- Solve conflict - Shows Analysis tab [Section 8.4.2.2], where conflicts are resolved.
- Analyse unanalysed steps - Requests analysis of all unanalysed steps of the use case.
- Reanalyse all steps - Requests analysis of all steps of the use case.
- Discard manual changes - Discards manual changes performed in the result and use the generated result.
- Goto target - Shows the target step, if the sentence type is set to goto action. In the submenu another step may be chosen as the target.
- Included use case - Shows the acronym of the included use case, if the sentence type is set to use case inclusion. In the submenu, you can choose another use case to be included. You will be offered changing the text of the step to a predefined text.
- Handles included use case failure - Sets a special option for this step. Only available for condition steps, that extends a including use case step. See Section 7.1 for effects of this option.

- Import steps from file - Imports steps for this use case from a file. See Section 8.8 for details.
- Export use case to txt - Exports the use case to the selected file in text form.
- Export use case to UML - Exports the use case to the selected file in the UML format.

8.4.2. Analysis Tabs

The right side of the Use case window is composed of four tabs.

8.4.2.1. Description Tab

Consists of two parts. The left one is contains an the same entries as the dialog for creating use cases [Section 8.3.3]. The right part displays a summary of the step analysis result (including the action type, subject, indirect object and action label).

8.4.2.2. Parsed Tree Tab

This panel displays results from the parsing tools and tree analysis.

The biggest part of the tab is filled with the tree of the step's sentence. It combines the result of the linguistic and tree analysis. Nodes in the upper part represent phrases of the sentence, leafs correspond to the words of the sentence. Roles of words can be changed. If you change the role of a word, you will be offered regenerating the token. Grey color of the tree means, that manual changes were made and changing the words' role would have no effect.

Above the tree there is a list box for changing the action type. Some step types (e.g. request sending, goto, ...) offer a second list box for selecting the acting entity or the target step. Action type cannot be changed in steps marked as "Condition" and no other step can be mark as "Condition", unless it is a condition that triggers an extension or variation.

The topmost box shows the action label for the step.

8.4.2.3. Token Tab

Token tab enables the user to change the token. You can read more about token management in Chapter 6. In the upper part, the current action label is displayed. Below, there are three sections, each for a different type of token changing. You should know, that only tokens of Condition, Request receiving, Request sending, Internal action and Unknown steps can be modified.

The Merging section allows you to choose from offered tokens. Button "All tokens" displays a dialog with the list of all tokens used in the project. Click use, to merge the token with the selected one.

The Mixing section allows you to choose which words will be used and in which order. The "Use" button, applies the change.

The Write by hand section allows the user to write the whole token by hand.

8.4.2.4. Pro-case tab

Create and displays the Pro-case for the selected use case. The Pro-case is generated as described in Section 7.2.

8.5. Properties Dialog

From the "Window" menu or by pressing Ctrl+T you can access the properties dialog. The dialog is divided into four sections.

8.5.1. Editing Section

Changes the behavior of the use case step tables.

- Automatic marking - If the property is on, the step mark of every new step is filled automatically and the step text column is automatically activated.
- Remarking - This option can only be switched on if automatic marking is on. Every time a step is inserted, remarking solves possible conflicts among the step marks. E.g., new step with mark "2" is inserted, but there already is a step with the same mark. The remarking algorithm automatically changes the conflicting step marks.
- Batch mode - If batch mode is enabled, every time enter is pressed in the second column of the use case table, a new step following the selected one is created. If batch mode is disabled, first enter triggers the step analysis, the second enter inserts a new step.

8.5.2. Tokens Section

This section allows you to change the properties of token generation. The first option - "Number of tokens in listing window" - sets how many tokens you will see in windows in the Token panel [Section 8.4.2.3]. The second option - "Percents of similarity to perform token learning" - influences token generation (see [Chapter 6] or the Developer documentation for details).

8.5.3. Parser Section

If the "Start parser in start up" option is enabled, linguistic tools are automatically started at application's start-up.

8.5.4. Auto Save Section

Sets how often the opened project is saved. Time is set in minutes. Value '0' means project will not be saved automatically.

8.6. Token Manager Window

This window displays all tokens present in the project. Every token can be manually modified (if the action type is condition, request receiving, request sending, internal or unknown). From a popup menu, you can choose another token to be merged with the active one.

If the modified token is used in multiple steps, you will be asked, whether all tokens with the same token should be modified.

8.7. Hierarchy of Entities Window

This window displays hierarchical relations between entities.

8.8. Importing Use Cases

Procasor provides simple importing into the use cases. Importing is accessible from popup menu in Use case table [Section 8.4.1]. Importing allows to import steps into the use case. Text files, step are imported from, have to be in specified format. Each line corresponds with one step. First word of the line is used as a step mark, the rest of the line is used as a text for the step.

Bibliography

References

[1] Vladimir Mencl. *Deriving Behavior Specifications from Textual Use Cases*. Proceedings of Workshop on Intelligent Technologies for Software Engineering (WITSE04, Sep 21, 2004, part of ASE 2004), Linz, Austria, ISBN 3-85403-180-7, pp. 331-341, Oesterreichische Computer Gesellschaft. Copyright © 2004.

[2] Frantisek Plasil and Vladimir Mencl. *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. Copyright © 2003. 10920617.

[3] Alistair Cockburn. Copyright © 2001 Addison-Wesley. 0201702258. Addison-Wesley Professional; 1st edition (January 15, 2000). *Writing effective use cases*. 270.

[4] http://www.uml.org .

[5] Frantisek Plasil and Stanislav Visnovsky. *Behavior Protocols for Software Components*. IEEE Trans. Software Eng. 28(11). Copyright © 2002.

[6] M. Mach and Frantisek Plasil. *Addressing State Explosion in Behavior Protocol Verification*. Proceedings of SNPD'04, Beijing, China. Copyright © 2004.

[7] *A Maximum Entropy Part-Of-Speech Tagger [http://www.cis.upenn.edu/~adwait/statnlp.html]* . Adwait Ratnaparkhi. In Proceedings of the Empirical Methods in Natural Language Processing Conference, May 17-18, 1996. University of Pennsylvania. Copyright © 1996.

[8] *A New Statistical Parser Based on Bigram Lexical Dependencies [http://www.cis.upenn.edu/~mcollins/]*. Michael Collins. ACL 1996: 184-191. 34th Annual Meetingof the Association for Computational Linguistics, 24-27 June 1996, University of California, Santa Cruz, California, USA, Proceedings. Morgan Kaufmann Publishers. Copyright © 1996.

[9] *Morph-A tool [http://www.informatics.susx.ac.uk/research/nlp/carroll/morph.html]* .

[10] *Tokenizer [http://www.clsp.jhu.edu/ws99/projects/mt/toolkit/]* .

[11] *Theory of Automata, lecture notes [http://kti.mff.cuni.cz/~bartak/automaty/index.html]* . Roman Bartak.

[12] Frantisek Plasil and Vladimir Mencl. *Use Cases: Assembling "Whole Picture Behavior"*. TR 02*/11, Dept. of Computer Science, University ofNew Hampshire, Durham . Copyright © 2002.

[13] *Poseidon UML modelling tool [http://gentleware.com/]* .

Appendix A. Installation Manual

A.1. Requirements and Prerequisites

Procasor is a program developed for Linux platform, written in the Java programming language. Procasor is integrating highly sophisticated linguistic tools, which increase hardware requirements. These requirements for running the application can be summarized as follows:

- Linux OS (kernel 2.2 or above, libc 6 or above),
- Java[tm] runtime 1.4 or above (1.4.1 or above highly recommended),
- Perl version 4 or above (installed as /usr/bin/perl)
- At least 300MB of free memory,
- 350MB of free disk space.

Procasor has been tested and validated on the following platforms: Fedora Core 2-4, Gentoo Linux 2005.1, Mandrake 10. The application was developed on Java 1.4 (mainly on 1.4.2_06). Although we have successfully tested Procasor with Java version 1.5, we recommend Java 1.4 distribution.

A.2. Installing Procasor

Procedure:

1. Make sure that the Java[tm] bin/ directory is referenced in the PATH environment variable and, at the same time, check that the Java[tm] runtime in PATH is the right version:

   ```
   $ java -version
   java version "1.4.2_06"
   Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.2_06-b03)
   Java HotSpot(TM) Client VM (build 1.4.2_06-b03, mixed mode)
   ```

2. Unpack the Procasor distribution to your installation directory:

   ```
   $ gzip -d -c procasor.tar.gz | tar xvf -
   $ ls procasor
   bin/
   config/
   documentation/
   ...
   tools/
   ```

3. Run Procasor. Procasor can be run by the **procasor** script from the **bin/** directory.

   ```
   $ ./procasor/bin/procasor
   ```

A.3. Content of the Installation Directory

bin/	Contains the script for running Procasor.	
	procasor	Script for launching Procasor.

config/	Contains configuration files for linguistic tools used in application and the property file for storing user's settings. **config.xml, properties**
documentation/	Contains Procasor documentation in HTML and PDF (Acrobat) formats. The documentation consists of two books - User documentation and Developer documentation.
documentation/api	Contains Procasor API documentation in HTML format.
exports/	Directory recommended for exporting projects and use cases from Procasor.
imports/	Recommended directory for importing use case steps to Procasor.
lib/	Contains the .jar files used for running the application. **externals/** Contains external .jar used for running Procasor. **procasor.jar** Procasor project .jar file.
logs/	Contains log files of the application.
projects/	Directory used for saving the projects. **marketplace.pro** Example project for demonstrating Procasor functionality.
src/	Contains the source code for Procasor.
tools/	Contains the directories in which the linguistic tools used in Procasor are stored: **CollinsParser/**, **morph/**, **MXPost_tagger/**, **Tokenizer/**

Appendix B. Getting started

B.1. Creating a New Project

To create a new project simply press "Ctrl+N" or select "New" in "Project" menu from the main menu. A window for setting the name and description for the project appears. After filling the name and/or description press the "Use" button. Project window [Section 8.3] appears.

Example: Create a new project called "Simple Marketplace".

B.2. Adding Entities

Start with adding entities. Button "Add" in the box "Entity" displays a window for creation of entities [Section 8.3.1]. Remember the name and the acronym of the entity have to be filled.

Example: To create "Marketplace Information System" simply fill "Marketplace Information System" in the Name field and "MIS" in the Acronym field. Press the "Save" button. Create another entity, Seller (SL) in the same manner.

B.3. Creating a Use Case

To create a new use case, use the "Add" button in the "Use case" box. Use case name and acronym have to be filled. In addition, you have to specify the actors of the use case. After saving the use case, a new use case window [Section 8.4] appears. Here, you may change the information entered in the previous step, if necessary.

Example: Select the MIS entity in the entities table, then press the "Add" button in the Use case section of the Project frame. A dialog for use case creation appears. Fill "Seller submits an offer" into the Name field and "UC1" into the Acronym field. There already is one line in the Actors table, entity MIS is set as the System under Design (SuD). Click next line in the Actors table (marked "...") and choose Seller as the primary actor of the use case. Press the "Save" button.

Before you start entering steps to the use case, make sure linguistic tools are ready. If the Parser progress bar in the left lower corner is green, everything is alright. If the progress bar is red, linguistic tools have to be started. See Section 8.2.2, where manipulation with linguistic tools is described.

B.4. Adding Steps to the Use Case

Now you may start adding steps to the use case. Steps are written into two tables in the left part of the use case window (upper table for the main success scenario, lower one for extensions and variations).

When the text of a step is confirmed (e.g. "Enter" pressing, selection changing), the step is automatically analysed. Result of the analysis consists of the parse tree, step type, acting entities and token. All results are displayed in the tabs in the right side of the window and every one of them can be changed manually. To modify the token, switch to the Token Tab [Section 8.4.2.3]. Chapter 6 gives detailed information on the usage of this tab.

Example: Click the first line in the Step text column of the upper table, write the text of the sentence: "Seller submits item description" and press Enter. Sentence will be analysed. It is recognized as a request receiving action, Seller being the source of the request. Switch to the "Parsed tree" panel. Here you can see the tree of the sentence (result from the linguistic tools), analysed results and token. "submitItemDescription" is the event token for this step, we are satisfied with it and will not change it.

Example: Click the arrow on the right of the previously added line. A new branch is created (in the lower table) and appended to the first step (therefore marked 1a). Enter "Item not valid" as the text of the condition and leave Extension as the type of the branch. Step type is condition, the event token is itemNotValid. This time, we may be unsatisfied with the token and will change it manually. We switched to the token tab, activate the Write by hand pane and enter "itemInvalid" as the new event token.

Example: Again, click the arrow on the right of the previously added line. A step is added to the branch. Enter "Use case aborts" as the text of the step. This time, a special action is detected, as desired.

B.5. Generating Behavior Specification

When all steps of the use case have been entered and analysed and each step has its token, Pro-case can be shown. A Pro-case for the use case is displayed in the Pro-case tab in the Use case window.

Example: The Pro-case for the "Seller submits an offer" use case from the examples above is `?SL.submitItemDescription; ( NULL + #itemInvalid )`.

To receive the global behavior of the described entity, you have to set its model expression. To do this, go back to the Project window and press the Edit button in the Model section. There, you can enter the model expression manually or using the Expression editor.

Example: Select the MIS entity and press the Edit button in the Model section. In the Model expression field, enter "UC1;UC1". This expression describes the behavior of the Marketplace Information System, as two consequent runs of the use case "Seller submits an offer".

Now, you can display the Pro-case describing global behavior of the MIS entity by pressing the Pro-case button in the Model section of the Project window. In addition, you may export the project to UML or to a text file.

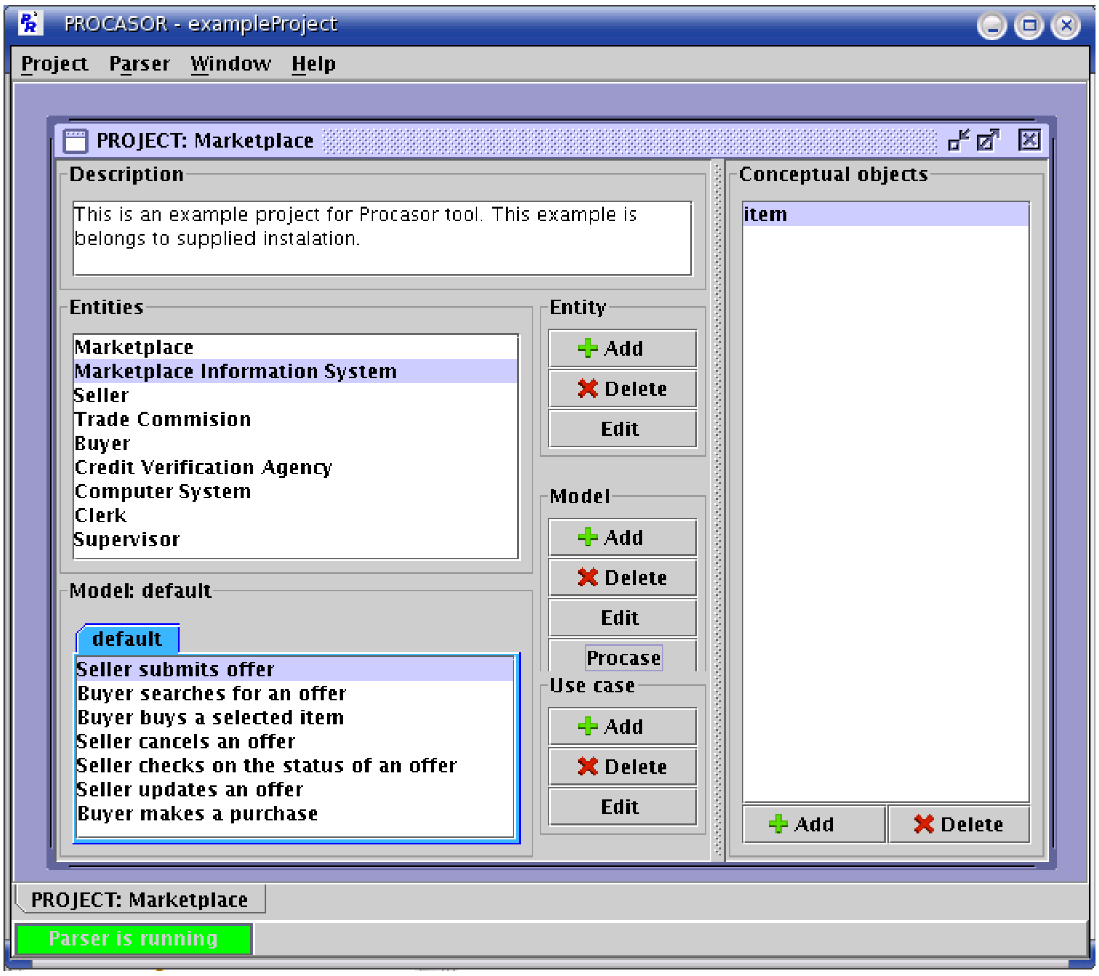
Appendix C. The Marketplace Project

C.1. Opening the Project

This chapter will introduce you to the example project Marketplace, which is supplied with the Procasor installation. Use cases in the market Marketplace project are extracted from [12]

To open the project, simply press Ctrl+O or select "Open" from the "Project" menu. Choose Marketplace.pro (file with the project) in the dialog with file listing and click "Open". Project window will appear (shown in Figure C.1).

Figure C.1. Main window screenshot



This is an example project for Procasor tool. This example is belongs to supplied instalation.

item

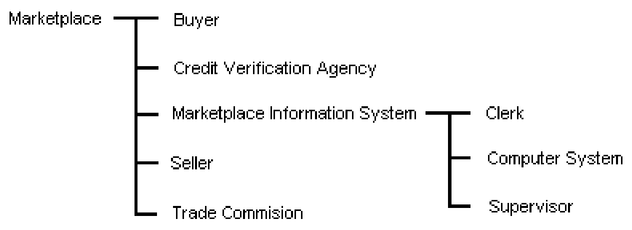
Marketplace
Marketplace Information System
Seller
Trade Commision
Buyer
Credit Verification Agency
Computer System
Clerk
Supervisor

Seller submits offer
Buyer searches for an offer
Buyer buys a selected item
Seller cancels an offer
Seller checks on the status of an offer
Seller updates an offer
Buyer makes a purchase

C.2. Description

Our example describes marketplace with its most important entities and basic actions. In Figure C.2 you can see the diagram of all entities.

Figure C.2. Hierarchy of entities



The main entity is the entire Marketplace, which contains Marketplace Information System (MIS), Seller (SL), Buyer (BU), Trade Commission (TC) and Credit Verification Agency (CVA). Except for MIS (the most important one), none of these entities contain its own use cases. They are only actors.

Marketplace Information System contains several use cases describing cooperation of all mentioned entities. In these use cases, MIS acts as the system under discussion.

Looking closely on MIS, we will see, that MIS is not an atomic entity. MIS is compound of three entities - Computer System (CS), Clerk (CL) and Supervisor (SU). All these entities contain their own use cases.

Let us focus on MIS and its use cases for now. MIS involves seven use cases. We will look at the first one - "Seller submits offer". This use case describes the cooperation of the seller and the MIS primarily. Therefore, Seller is the primary actor of the use case. The Trade Commission is involved as well, it is a supporting actor of the use case. The use case itself describes all actions necessary to submitting an offer.

We will take a closer look on the first step of this use case - "Seller submits item description". The step is analyzed as a request receiving action, Seller being the source of the request. "?Sl.submitItemDescription" was chosen as the token for this step. Chapters 3 through 6 give detailed information on the step analysis, while Section 8.4.2 describes the way analysis results are displayed.

The relation to the rest of the project is significant as well. The same action - seller submits an offer - is described (from other point of view) in the use case "Clerk submits an offer on behalf of a Seller" for the Computer System entity. In this use case, Marketplace Information System no longer is a "black box". Instead, this use case describes the actions taken by the contained entities. For example steps "System validates the description" and "System lists the offer in published offers" refer to the same actions. Their tokens should be the same, as you can verify.

C.3. Use Case List

List of all use cases with actors follows.

- Marketplace information System

  Acronym: MIS

  Use cases:

  - Seller submits offer

    Acronym: MIS1

    Primary actor: Seller

    Supporting actor: Trade Commission

  - Buyer searches for an offer

  Acronym: MIS2

  Primary actor: Buyer

  - Buyer buys a selected item

    Acronym: MIS3

    Primary actor: Buyer

    Supporting actor: Seller, Credit Verification Agency

  - Seller cancels an offer

    Acronym: MIS4

    Primary actor: Seller

  - Seller checks on the status of an offer

    Acronym: MIS5

    Primary actor: Seller

    Supporting actor: Trade Commission

  - Seller updates an offer

    Acronym: MIS6

    Primary actor: Seller

    Supporting actor: Trade Commission

  - Buyer makes a purchase

    Acronym: MIS7

    Primary actor: Buyer

    Supporting actor: Seller, Credit Verification Company

- Computer System

  Acronym: CS

  Use cases:

  - Clerk submits an offer on behalf of a Seller

    Acronym: CS1

    Primary actor: Clerk

    Supporting actor: Trade Commission, Supervisor

  - Buyer searches for an offer

    Acronym: CS2

    Primary actor: Buyer

  - Clerk buys a selected item on behalf of a Buyer

  Acronym: CS3

  Primary actor: Clerk

  Supporting actor: Seller, Credit Verification Agency

  - Seller cancels an offer

    Acronym: CS4

    Primary actor: Seller

  - Seller checks on the status of the offer

    Acronym: MIS5

    Primary actor: Seller

    Supporting actor: Trade Commission

  - Seller updates an offer

    Acronym: CS6

  - Buyer makes a purchase

    Acronym: CS7

    Primary actor: Buyer

    Supporting actor: Seller, Credit Verification Company

  - Supervisor makes an internal audit

    Acronym: CS8

    Primary actor: Supervisor

- Clerk

  Acronym: CL

  Use cases:

  - Seller to Clerk

    Acronym: CL1

    Primary actor: Seller

    Supporting actor: Computer System

  - Buyer to Clerk

    Acronym: CL2

    Primary actor: Buyer

    Supporting actor: Computer System

- Supervisor

  Acronym: SU

Use cases:

- Supervisor validates a seller

  Acronym: SU1

  Primary actor: Computer System

- Supervisor performs internal audit

  Acronym: Su2

  Primary actor: Computer System

Appendix D. License

The Procasor project is copyright(c) 2005 Procasor team [Section 1]

D.1. License information

The Procasor team [Section 1] ("Owner") grants to the individual researcher who install this software ("Licensee") a non-exclusive, non-transferable run-time license to use the Procasor software ("Software"), subject to the restrictions listed below under "Scope of Grant."

D.2. Scope of grant

The Licensee may:

- use the Software for educational or research purposes;
- permit others under the Licensee's supervision at the same site to use the Software for educational or research purposes;
- copy the Software for archival purposes, provided that any such copy contains all of the original proprietary notices.

The Licensee may not:

- use the Software for commercial purposes;
- allow any individual who is not under the direct supervision of the Licensee to use the Software;
- redistribute the Software;
- copy the Software otherwise than as specified above;
- rent, lease, grant a security interest in, or otherwise transfer rights to the Software;
- remove any proprietary notices or labels accompanying the Software;

D.3. Disclaimer

The Owner makes no representations or warranties about the suitability of the Software, either express or implied, including but not limited to the implied warranties of merchantability, fitness for a particular purpose, or non-infringement. The Owner shall not be liable for any damages suffered by Licensee as a result of using, modifying or distributing the Software or its derivatives.

D.4. Consent

By downloading, using or copying the Software, Licensee agrees to abide by the intellectual property laws, and all other applicable laws of the Czech republic, and the terms of this License. Ownership of the Software shall remain solely with the Owner.

D.5. Linguistic Tools Licenses

The distribution of the Procasor project contains linguistic tools which are stand-alone applications with their own licenses. Before using the Procasor, please read carefully licences of linguistic tools, which can be found in the tools/ directory [Section A.3].

D.6. Termination

The Owner shall have the right to terminate this license at any time by a written notice. Licensee shall be liable for any infringement or damages resulting from Licensee's failure to abide by the terms of this License.

Index