Methods used for requirements engineering

Veronika Sládeková

May 3, 2007

Contents

1 Introduction **2**
1.1 Abstract 2
1.2 Foreword 3
1.3 Motivation 3
1.4 Goal 4

2 Theoretic background **5**
2.1 Software life cycle 5
2.2 Software Requirements 7
2.2.1 IEEE Recommendation for SRS 8

3 Informal Methods **13**
3.1 QuARS Tool 14
3.2 Template 17

4 Semiformal Methods **20**
4.1 Unified Modelling Language 20
4.1.1 Requirements Model 21
4.1.2 Structural Models 22
4.1.3 Behavioral Models 22
4.1.4 Extension Mechanism 25
4.2 User Requirements Notation 25
4.2.1 Goal-oriented Requirements Language 25
4.2.2 Use Case Maps 28

5 Formal Methods **33**
5.1 Z notation 34
5.2 KAOS 35
5.3 Usage of Formal Methods 37

6 Choosing the Right Technique **38**
6.1 Considered properties 38

7 Conclusion **42**

Chapter 1

Introduction

1.1 Abstract

The thesis discribes the model of software life cycle and refers to the importance of requirements capture and analysis. The main characteristics of the good requirements specification, taken from IEEE recommendation are described.

The thesis presents two informal methods for capturing software requirements. The first method uses lexical analysis for parsing of written text. The parsed text is compared to the defined dictionary. The dictionary is defined to indicate every use of word or sentence structure that leads to ambiguous description. The second method works as a template to assist in writting a good quality requirements specification. The template forces the writer to write in defined manner. This manner is defined by formal context free grammar.

As an example of semiformal method the thesis decsribes the UML notation. UML is a general modelling method with various kinds of diagrams, each viewing the system from different points of view. It is the most used modelling method. As a contrast to UML the URN notation is described. The URN is esspecially developed for usage in telecomunication systems and services. The diagrams are designed to used for concrete situation. The transformation of URN into other ITU-T languages is supported. The URN notation models also the non-functional requirements, which importance is often underestimated.

For precise specification the mathematics are used. Formal methods are based on set theory and logics. The Z language as an example of formal method is described. Desribed language KAOS is an example of graphical notation with formal description. The occasions of proper and effective use of formal methods are stated.

The thesis describes the main characteristics, notations and examples of

selected techniques representing different approaches. Discuss their possible collaboration and usage in various kinds of software systems.

1.2 Foreword

Supposition of creating a software system with good quality is to understand the process of creation of the software system and to give appropriate attention to every step in the process of creation. With creation of more complicated and large systems the need of good coordination of the work is also neccessary.

The first step in the process of software creation is the elicitation and elaboration of the software requirements followed by analysis. This part of the process of software creation is wrongly considered as easy one and not enough attention is payed to it. The requirements enginnering is slowly finding its place in computer science.

There are many theories about how the requirements should look like, what techniques are the best for requirements elaboration, but in the real project the need of good quality requirements is often underestimated and various available techniques for requirements elaboration are rarely used.

In my thesis I try to remind the necessity of good quality requirements. Introduce various techniques for requirement elaboration and to show their aplicability in the real world systems

1.3 Motivation

The complexity of the software systems made is rapidly raising. The process of software development complicates with the raising complexity of the systems made.

There is a tendency to automatize as many parts of the process of software development as possible. The part of requirements capture is very special. It is due the need of cooperation with the customers - laics. This is the part where informal meets the formal. We cannot eliminate the informalism totally, because there is the customers, who do not understand the formalism at all. But with informal came the problems of ambiguity and misunderstanding.

The semiformal methods were developed to reduce the ambiguity and to raise the readability of the requirements stated. The graphical notations use graphs intuitively representing the system. With very little learning the quality of the requirements capture is rising rapidly. Not all problems are eliminated but the effectivness is very high. This is the reason of the popularity of using graphical notations in the process of software development.

There are other models developed especially for capturing software requirements. Formal methods are based on exact sciences of matematics as logics and set theory.

At this time there exists many different aproaches dealing with software requirements. When creating software developers need to decide which methods are the most suitable for the software being developed. My diploma thesis gives an overview of aproaches dealing with software requirements. An examples of aproaches are given, to help the potential developers to choose a good method for their particular type of system.

1.4 Goal

The goal of my diploma thesis is to introduce techniques for elaboration of software requirements and gives an overview of different aproaches.

The thesis is divided into several chapters. In the second chapter is described the theoretic background, an example of software life cycle, the types of software requirements, and the recommendation of IEEE of the content of Software Requirements Specification to describe a good quality requirements specification.

The third chapter describes informal approaches in requirements elaboration. Two different techniques are presented to give an inspiration of how to deal with natural language.

The fourth chapter describes semiformal methods based on graphical notations. URN and UML notations are introduced and compared.

The fifth chapter introduces formal approach. The basic characterisation of Z notation and KAOS are given.

In the last chapter all techniques are summarized and their aplicability in various kinds of software systems is discussed.

Chapter 2

Theoretic background

2.1 Software life cycle

The model of software development life cycle describes the phases of software development and their order in the development. There exists several models of software development life cycle, many companies developed their own model, adapted to the companys' principles. All these model has similar patterns, which can be put together to create a general model of software development life cycle (Figure 2.1). [1]

The requirements phase is responsible for gathering all needed information from the customers about the wanted functionality of the system. These functionalities are written down and, when accepted by customer, the design phase can begin. In the design phase the details of system are stated. The software and hardware architecture is defined. The implementation is where the code is produced. It is the longest phase of the life cycle. At the end is the test phase, where the funcionality is checked, wheather the software fulfill the customers needs.

Examples of software development life cycle models are given below (Figure 2.2, 2.3, 2.4, 2.5). In each of this models the phase of requirements is at the beginning, so the quality of the requirements has an impact on the whole system development.

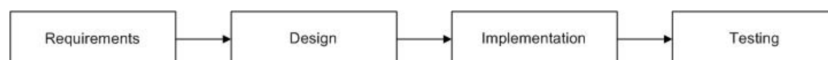

Figure 2.1: The general model of software development life cycle

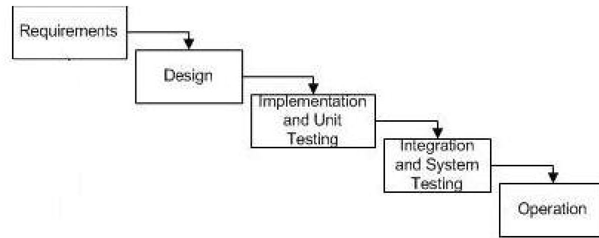

Figure 2.2: The Waterfall model

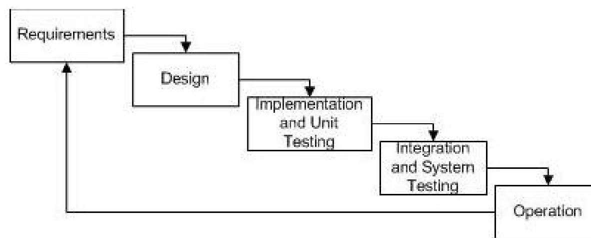

Figure 2.3: The Incremental model

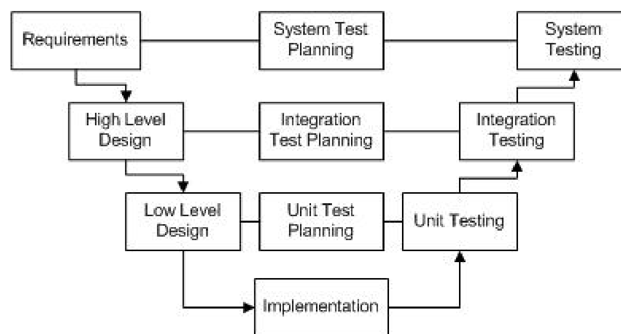

Figure 2.4: The V-shaped model

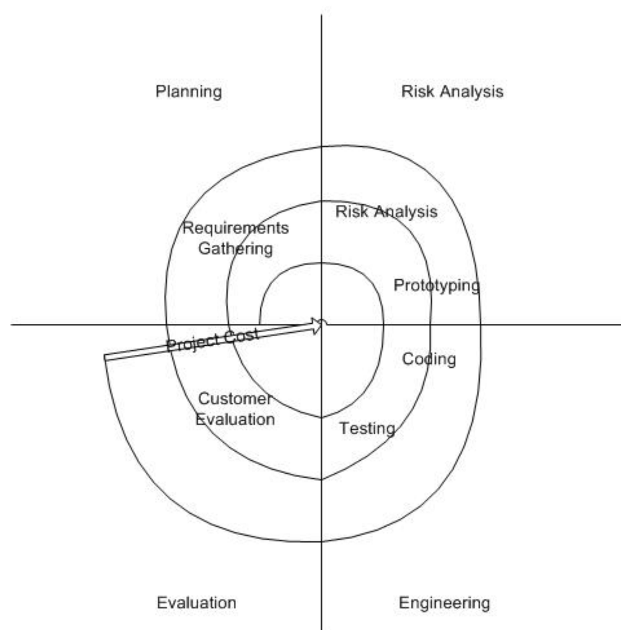

Figure 2.5: The Spiral model

2.2 Software Requirements

The process of software requirements engineering can be divided into four main phases (See figure 2.6).

First is the elicitation of the requirements with the customer. At this phase the meetings with the customer are held, where analysts collect the requirements.

Then the phase of analysis came. Here are the collected requirements analyzed to find out all hidden meanings.

After analysis, requirements are documented. Here is the output Software Requirements Specification (SRS) [2].

The last phase is the verification of SRS document. The document as a part of the contract is verified by the customer. When the verification process ends up with more requirements the process starts from the beginning again.

The process is drawn as spiral to show the incremental property. The process ends up when the verificatin passes with closed document.

The requirements itself can be divided into two main groups, functional and nonfunctional requirements. The functional requirements of the system describe the expected functionality of the system. Functions are described as input output functions. The exceptions and their handlings are stated.

The non-functional requirements do not care about what the system should do, but how. Non-functional requirements state the limitations for the

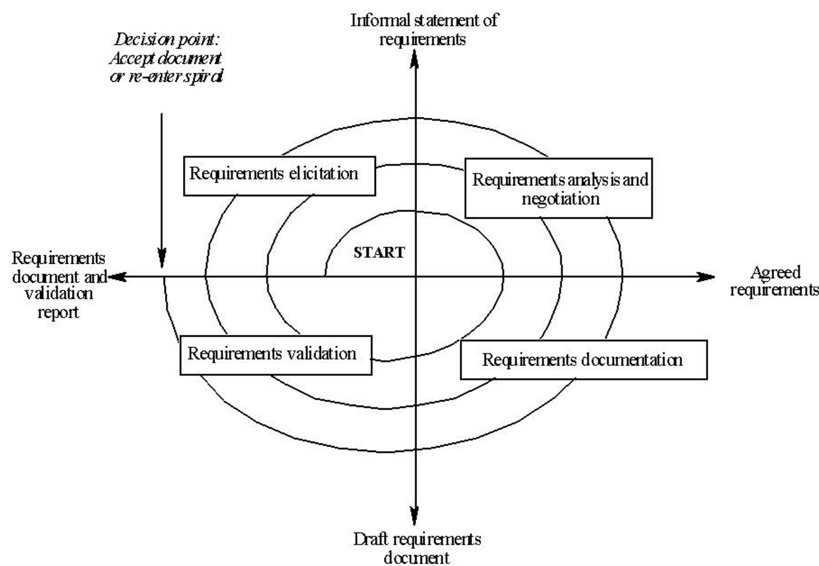

Figure 2.6: The model of requirements phase

system. The non-functional requirements include the response time of the system, the hardware limitations, the business constrains, and other things. Sometimes it is really hard to decide wheather is the requirement functional or non-functional. It is due to level of detail of the requirement. Because if a non-functional requirement is stated, when is it deaply analyzed it concludes into several functional requirements. So at some level of detail only functional requirements are considered.

2.2.1 IEEE Recommendation for SRS

The definition of software requirements are the first step in developing software. To reduce the software cost, developers are trying to reduce the amount of mistakes. Most of the misunderstandings are between the developers and the customers. They are from two different professions and got different styles of thinking and expressing.

The result of software requirement specification process is a document named Software Requirement Specification (SRS), which should be unambiguous and complete. The good way to start writing a SRS is to follow IEEE Standard 830, Recommended Practice for Software Requirements Specifications [3]. This standard describes what should be in the SRS and which rules should the writer obey to end up with qood quality specification. Appendix of this standard also contains several sample SRS outlines, for better understanding.

The SRS should provide the basis for agreement between the suppliers and

customers. The customers or potential users should be able to determinate if the functions of the software specified in SRS meet their needs.

A good written SRS can be used to estimate the costs and schedules of the development of the software. When the SRS is complete and precise, so no big changes are needed, the estimates are good enough. The advantage is that they are stated at the begining of the development process.

The requirements stated in the SRS are the baseline for validation and verification plans. Because the SRS describes the product not the project, it can serve as a basis for later enhancement or tranformation to other users or machines. In the SRS there should not be design or project requirements. It should only state the constrains on design and project.

The SRS writer should address:

- Functionality. What is the software supposed to do ?
- External intefaces. How does the software interact with people, the system's hardware, and other software?
- Performance. What is the speed, availability, response time, recovery time of various software functions, etc.?
- Attributes. What are the portability, correctness, maintainability, security, etc. considerations?
- Design constrains imposed on an implemantation. Are there any required standards in effect, implementation language, policies for database integrity, resource limits, operating environment(s) etc.?

Attributes of quality of a SRS are:

- Correct
- Unambiguous
- Complete
- Consistent
- Ranked for importance and/or stability
- Verifiable
- Modifiable
- Traceable

SRS Properties

Correct An SRS is correct if, and only if, every requirement reflects the actual needs of customer. There is no tool or procedure to check the correctness, only customers and users can determinate if the SRS is correct.

Unambiguous An SRS is umambiguous if, and only if, every requirement has only one interpretation. The SRS is used by both developers and users of software. The problem is that they are from different environments, that's why they tend to understand things differently, without knowing it. There are techniques used to capture these ambiguities. Mostly used is review by independent people.

Complete An SRS is complete if, and only if, it contains all significant requirements, definitions of the responses of input data, full labels and references to all figures, tables, and diagrams, definitions of all terms and units of measure. Phrase TBD (to be determined) is not allowed in complete SRS. When is the use neccessary there should be a description why the answer is not known, who is responsible to eliminate TBD and when it must be eliminated.

Consistent An SRS is consistent if, and only if, there are no requirements in conflict. There may be real-world objects in a conflict, for example color of a control light. Or there may be a logical conflict, for example different mathematical operations within input variables. Sometimes the same real-world objects are decsribed differently within a group of requirements. The use of standard terminology and definitions removes this problem. If the SRS document is in conflict with another high-level document, then it is not correct.

Ranked for importance and/or stability A SRS is ranked for importance and/or stability if every requirement has a label to indetify it's importance or stability. Not all requirements are equally important. Some of them may be essential, while other may be desirable. It is best to let customers consider each requirement, this can clarify hidden assumptions.

Verifiable An SRS is verifiable if every requirement is verifiable. A requirement is verifiable if there exists finite process that can check that the software meets the requirement. The statement 'the program shall never enter an infinite loop' is nonverifiable, because the testing is teoretically impossible. The statement 'Output of the program shall be produced within 20s of event 60% of the time, and shall be produced

within 30s of event 100% of the time' is na example of verifiable statement, where every property is measurable quantified. If a requirement is not expressible in verifiable terms at the time the SRS is written, there should be stated at which point of software development the requirement must be put into verifiable form.

Modifiable An SRS is modifiable if the structure allows easy changes to the requirements without breaking the completeness and consistency. These include to have a coherent and easy-to-use organization with a table of contents, an index and explicit cross-referencing. The SRS should not be redundant, this means every requirement should appear only once. The redundancy is not an error, but can cause one if the redundand requirement is to be revised. If not explicitly cross-referenced it is revised only once and cause inconsistency.

Traceable An SRS is traceable if the origin of every requirement is known and referenced. There are two types of traceability. Backward traceability is the reference of each requirement in previous documents. Forward traceability is that each requirement has a unique name or reference number, so it can be referenced in future documents. Forward traceability is important for maintenance and operation phase, to assure that the needed change is made on all affected requirements.

SRS Layout

A good SRS should not miss information included in the following draft of the table of contents.

i Introduction

  ii Purpose

  ii Scope

  ii Definitions, acronyms, and abbreviations

  ii References

  ii Overview

i Overall description

  ii Product perspective

  ii Product functions

  ii User characteristics

  ii Costraints

    - ii Assumptions and dependencies
    - ii Apportioning of requirements
- i Specific requirements
- i Appendixes

In the Introduction part the overview of the entire document is given. The organization of the SRS is explained. Definitions and references if larger can be stated as a reference to appropriate appendix.

The Overall description includes the relationships, if any with other systems, specify the interfaces of the system and other constraints made on the system. The summary of the main functions. The qualification of expected users is stated as a reason for requirements. This section also identifies the requirements that may be delayed for future versions of the system.

The biggest and most important is the part where the concrete requirements are stated. The requirements should be uniquely identifiable and the organization should maxize the readability of the document. The level of detail of the requirements should be sufficient to designers and testers.

This is only a recommendation, it can be adapted to the needs of the user. I can only recommend to read this IEEE recomendation to realize all the needed components of the document and properties that is important to be checked.

Chapter 3

Informal Methods

The most natural way of expresing requirements is plain natural language. The main problem with natural language is that it is ambiguous, and the requirements written with it lack the needed quality.

The alternatives to the plainly written text are:

- structured natural language. In structured natural language the writer is limited by the terminology. The writer may have a preddefined template, which helps him to keep the predefinned way of writing.
- graphical notations. The graphical notations by the use of different charts or graphs gives the image of the system from different points of view. These notations are suitable mainly for defining functional requirements.
- mathematical specifications. The precise model of the system, using set and logic theory is made. This model is only a step from the real code.

Requirements written in natural language are the junction points between customers - laymen and developers - profesionals. Natural language is ambigous, that is why are prefered requirements expressed by other more formal methods. But requirements written in natural language are neccessary for layman to understood. Mostly it is the first step in expresing requirements. These requirements are then transformed into other models, mostly graphical. Many tools are developed to simplify the trasformation. Mostly based on lexical and syntactical analysis. I chose two of them. First is tool made for automatical quality evaluation of Natural language requirements. The second is a template made for writing natural language requirements in a stylizied english grammar defined by context-free grammar developed especially for this purpose.

3.1 QuARS Tool

Tool called QuARS (Quality Analyzer of Requirements Specification) [4] was developed at Instituto di Elaborazione dell' Informazione del C.N.R. In Pisa, Italy. This tool analyze natural language requirements and check them againt Quality Model. To make transforming natural language requirements into formal models easier, and more automatized.

This tool is made for detecing and removing defects that could cause problems in the transformation to formal models. The tool uses syntactic analysis for elaboration of document and then checking the words and their position in the sentence against dictionary adjusted for this purpose.

As the first step in development the Quality Model was designed. It is composed of quality properties. The high level properties of the Quality model are :

- Testability - each requirement should by specified in a quantitative or pass/fail test manner
- Completeness - The entities refered by the requirement should be precisely identified
- Understability - The requirement should be fully understand by developers and also by users when reading requirement specification document
- Consistency - The requirements should avoid disagreements

The violation of these properties are signalized by indicators stated in table 3.1. After the analysis of many software requirements specifications taken from industrial projects, the keywords used for detecting the indicators were defined. In table 3.2 are listed examples of 'bad' sentences, to realize the needed writing style. Not every indicator can be shown as a bad sentence example. Indicators such as comment frequency, directives frequency, readability index, under-reference and unexplanation cover the whole document. To catch these indicators the whole document should be analyzed.

The linguistic analysis engine in the QuARS tool defines a basic english Grammar with about 40 production rules and small dictionary. Dictionary consists of grammatical words developed by linguists such as determiners, particle, quantifier, auxialiary verbs, etc, and semantic words automatically generated from morphological analyzer ENGLEX (http://www.sil.org) such as nouns, adjectives, adverbs, verbs. The dictionary contains words defined by AECMA-boeing simplified English Project (http://www.aecma.org/Publications/SEnglish/senglish.htm)

The main logic modules of QuARS tool are : Lexical analyzer (ENGLEX) Syntax analyzer Quality Evaluator Special purpose grammar Dictionaries

Indicator	Description	Notes
Optionality	An Optionality Indicator reveels a requirement sentence containing an optional part (i.e. A part that can or cannot be considered)	Optionality - revealing words : possibly, eventually, if case, if possible, if appropriate, if needed, ...
Subjectivity	A Subjectivity Indicator is pointed out if sentence refers to personal opinions or feeling	Subjectivity - revealing words : similar, better, similarly, worse, having in mind, take into account, take into consideration, as [adjective] as possible
Vagueness	A Vagueness Indicator is pointed out if the sentence includes words holding inherent vagueness, i.e. Words having a non uniquely quantifiable meaning	Vagueness - revealing words : clear, easy, strong, good, bad, efficient, useful, significant, adequate, fast, recent, far, close, in front, ...
Weakness	A Weakness Indicator is pointed out in a sentence when it contains a weak main verb	Weak verbs : can, could, may.
Under-specification	An Under-specification Indicator is pointed out in a sentence when the subject of the sentence contains a word identifying a class of objects without a modifier specifying an instance of this class	This indetificator deals with the syntactic and semantics of the sentence under evaluation
Under-reference	An Under-reference Indicator is pointed out in a NL-SRS document when a sentence contians explicit reference to : -not numbered sentences of the NLSRS document itself -documents not referenced into the NLSRS document itself -entities not defined nor described into the NLSRS document itself	-

Indicator	Description	Notes
Implicity	An Implicity Indicator is pointed out in a sentence when the subject is generic rather than specific	Subject expressed by : Demonstrative adjective (this, these, that, those) or Pronouns (it, they, ..). Subject specified by : Adjective (previous, next, following, last, . . .) or Preposition (above, below, . . .).
Multiplicity	A Multiplicity Indicator is pointed out in a sentence if the sentence has more than one main verb or more than one direct or undirect complement that specifies the subject	Multiplicity - revealing words:and, or, and or, . . .
Comment Frequency	It is the value of the CFI (Comment Frequency Index). [CFI=NC/NR where NC is the total number of requirements having one or more comments, NR is the number of requirements of the NLSRS document]	-
Readability Index	It is the value of ARI (Automated Readability Index) [ARI=WS + 9*SW where WS is the average words per sentence, SW is the avrage letters per word]	-
Directives Frequency	It is the rate between the number of NLSRS ahd the pointers to figures, tables, notes, . . .	-
Unexplanation	An Unexplanation Indicator is pointed out in a NLSRS document when a sentence contain acronyms not explicitly and completly explained within the NLSRS document itself	-

Table 3.1: Quality Properties and their Indicators

Indicators	Negative Examples
Implicity	the **above** requirements shall be verified by test
Optionality	the system shall be such that the mission can be pursued, **possibly** without performance degradation
Subjectivity	**in the largest extend as possible** the system shall be constituted by commercially available software products
Vagueness	the C code shall be **clearly** commented
Weakness	the results of the initialization checks **may be** reported in a special file
Underspecification	the system shall be able to run also in the case of **attack**
Multiplicity	the mean time needed to remove a faulty board **and restore service** shall be less than 30 min.
Under-reference	the software shall be designed **according to the rules of the Object Oriented Design**
Unexplanation	the handling of any recieved valid **TC** packet shall be started in less than 1 **CUT**

Table 3.2: Examples of requirement sentences containing defects

The files in SRS document are analyzed by lexical analyzer to verify the correct english dictionary. Output of lexical analyzer are words each associated with it's lexical category. This is the input for syntactical analyzer. Syntactical analyzer builds the derivation trees. Syntactic nodes are associated with morpho-syntactic and application-specific data. Derivation trees are input for the quality evaluator module. Also the special dictionaries are used. The Evaluator according to the rules of Quality model and the dictionaries evaluate the sentences and provides warning messages. The main purpose of building this tool was to make it easy to use and run independently on the format of SRS document. And also modifiable to run effectively on particular application domains. For easy to use was developed graphical interface. For generality the expected format of SRS document is text format. Every format can be transformed into text format. Sometimes are some information kept in layout lost, but when we assume that the hierarchy of the requirements is established by numeration of requirements and not by formating, this lack of information doesn't compromise the validity. In QuARS it is possible to modify the Dictionaries, to assure a meaningful evaluation in the particular domain.

3.2 Template

Unlike the QuARS tool where the requirements are written and then checked, this approach is bottom-up. The automatization of requirements assessment

requires the understanding of natural language. This linguistic problem can be simplified by limiting the language itself. This is why William Scott and Stephen Cook at University of South Australia developed a context-free grammar for English language to describe software requirements [5]. This grammar limitates the writer to use the predefined writing style to improve the quality of the requirements written. Requirements are written into template that follows the rules of the grammar.

The context-free grammar was chosen because of its ability of parsing for its sufficient coverage of the task. Mathematicaly the grammar is defined as the quadruple $G =< Vn, Vt, P, omega >$ where terminals are the words of Engish language, the starting nonterminal is called <Requirement> and the grammar in Backus Naur From is discribed :

<Requirement> → [<TC Clause> ","] <Independent Clause> [<Restrictive Relative Clause>] | <Independent Clause> [<Restrictive Relative Clause>][<TC Clause>]

<Independent Clause> → <Subject><Auxiliary Verb><Verb><Noun Phrase>

<Restrictive Relative Clause> → [<Preposition>]<Criterion Indicator><Value>

<TC Clause> → <TC Indicator><noun Phrase>[<Verb><Noun Phrase>]

<TC Indicator> → <Preposition>|<Condition>

<Value> → <Number><Units>

<Noun Phrase> → [<determinant>]<Adjective><Noun>[<Preposition><Noun Phrase>][<Conjuction><Noun Phrase>]| [<determinant>]<Adjective><Adjective>[<Preposition><Noun Phrase>][<Conjuction><Noun Phrase>]

<Subject> → [<determinant>]<Adjective><Noun>

<Auxiliary Verb> → "shall"["not"]

<Criterion Indicator> → "no greater than"|"no less than"|"within"

The sentences are divided into phrases. They distinguish two kinds of phrases. Independent one which can stand alone as a full sentence and subordinate phrases dependent on the rest of the sentence. Independent phrases contains verb and a subject. Subordinate phrases are divided into subtypes.

A temporal phrase specifies when the action occurs.

Conditional phrase makes the action dependant on the subordinate phrase. (if, unless)

Relative phrase specifies the noun and give additional detail. In requirements it is recommended to use only restricted relative phrases, which help identify the referent noun. Non-restrictive relative phrases does not help in the identification that's why their use is redundant.

To satisfy the best practice in software requirements there should be placed restrictions on the language defined by the context-free grammar. It is known that the presence of pronouns is the source of ambiguity a that's why the use of pronouns should be omited when writting a requirements specification. Easy identification of relationships within requirements can be achieved by using active voice. The context-free grammar emphasizes the active voice. These restrictions limit the language, but they help in providing a good quality requirements.

To support parsing of manualy written requirements the grammar needs to allow the requirements to be structured in different ways. The grammar was implemented, creating the parse and assess algorithm. The words where checked against Thesaurus dictionary. For the purpose of sotfware requirements the dictionary was extended with abbreviations and acronyms, which are mostly used by software engineers. Because the dictionary consider the synonyms as equivalet, in the process of comparison of requirements the system interacts with writer to consider the minor variations in the meaning of synonyms.

A prototype named BADGER (Built-in Agent using Deterministic Grammar for the Engineering of Requirements) was created as a template for writing requirements.

Both approaches are only university projects, that were applied onto smaller systems, but the results were as expected better than not limited writing. Here you can see that with little checks the quality of the document can be better. Important is that this checking does not spend a lot of time due to automatization. Because of elaboration of natural language, the tools are forced to collaborate with dictionaries. This fact is suitable for other potential users, because with appopriate changes to the vocabulary the tools can be adapted to different conditions.

Chapter 4

Semiformal Methods

By semiformal methods are meant graphical representations of the system, mostly supported with written description. Graphical representation helps the understanding of the system. Detailed description in natural language may not be clearly understandable, when describing larger systems, because of the amount of information.

The most popular from all graphical notations is Unified Modelling Language (UML). This language consists of several types of diagrams to offer a complex model of a system from different points of view. As a second graphical notation I choose User Requirements Notation (URN) that belong to the family of ITU-T Languages created especially for telecomunication systems. URN includes special language just for nonfunctional requirements. There are only few methods concernig nonfunctional requirements, that is why I choose URN as a second method to present.

4.1 Unified Modelling Language

Unified Modelling Language (UML) [6, 7, 8] is modelling language for object-oriented systems. It is the combination of the notations of Rumbaugh, Booch and Jacobson. UML became a standard by the Object Management Group in 1997. The UML is semiformal specification language, trying to find a balance between formalism and readability.

UML diagrams do not have stated the level of abstraction to be used. The UML diagrams can be precise, but also more abstract, it depends on the author. It is cause the need of different level of detail in the system model. While during the presentation there is not need for specific details, after the contract submition the work at the system model get into higher detail.

The UML provides different points of view at the system. Requirements model shows the external funcionality of the system. How the system reacts to the environment. There are also the structural model and the behaviour

model. The behavioral model represents the behavior of the system, how is the system changing throughtout the time and when recieving different inputs as a stimuli. The structural model represents the structure of the system and also the structure of the data that is processed by the system. The structural model gives us an overview of the whole system, to better understand its division of the system and the processes inside the system.

With many types of diagrams, UML became large and complex model of the system. The UML model do not directly support the non-functional requirements. The extension mechanism of UML can be used to deal with some kind of non-functional requirements. There are many different attepmts to capture the non-functional requirements with UML diagrams. This field is not yet standardized.

4.1.1 Requirements Model

The UML requirements model is a black-box view of the system, that hides implementation decisions. The goal of the requirements model is to decsribe the behavior of the system as a whole, not concerning the components. It is a model of external funcionality of the system, the definition of the system at system-level of interaction.

The requirements model of UML is based on use cases. The Use case model consists of actors and use cases. Actors represents objects outside the system (users, others systems, hardware). Use case is a coherent part of functionality visible from the outside of the system. The interaction of actors and use cases is in the mean of exchanging messages. The actors are divided into primar and secondary. The primar actor initiate the use case. Secondary actors supports the use case. Use case have one primar actor and can have many secondary actors. The use case description can be textual, or other more formal techniques can be used.

The description of the Use Case includes a unique identification of the Use Case, primar and secundar actors. The input and output conditions of the Use Case are stated. The output conditions are divided into success end conditions and failed end conditions. Also the triggering event, that initiates the execution of the Use Case should be identified. The description of the Use Case is in form of steps that are supposed to be executed. The extensions from the success execution should be stated. The quality of the requirements written highly depends on the quality of the description. The structured description of the Use Case is used to achieve a good quality specification.

The mapping between the blackbox view of use cases and the whitebox view of structure of the system can be really hard, due to multiple collaboration of classes, that can be in more than one use case.

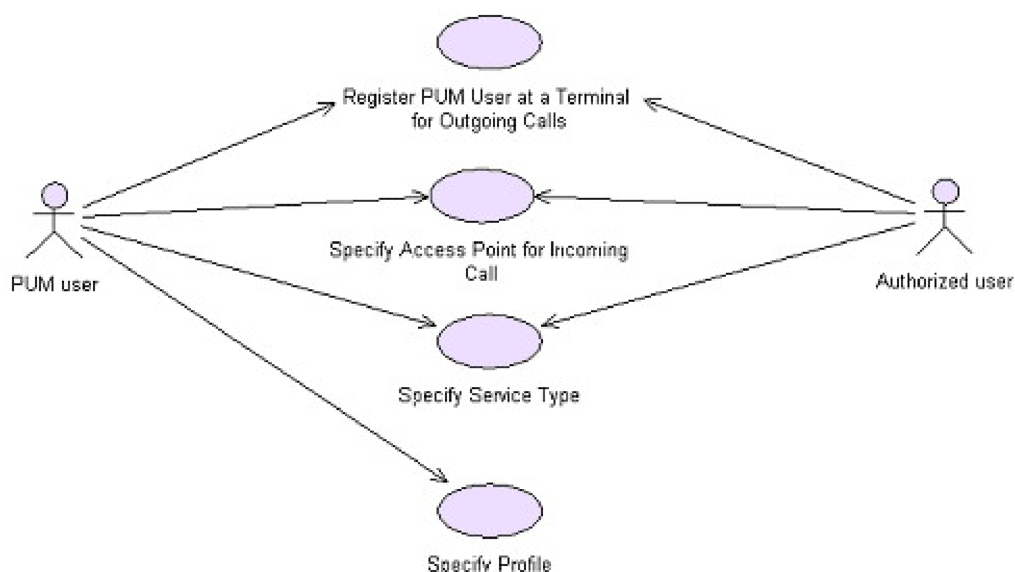

Figure 4.1: An example of UML Use Case diagram

4.1.2 Structural Models

The logical structure of the system, the interfaces of the classes, their relationships are captured in the class diagram and package diagram. The mapping of the physical structure of the system to the logic structure is captured in deployment diagram.

4.1.3 Behavioral Models

The behaviour of the system is modelled by two different views. One is the view of the data processing in the system. The second view is the reaction of the system to the events. This division is done because of the kinds of system being developed. The business systems are primary concerned with data processing, the data flow model is sufficient. On the other hand, real-time systems are event driven, with minimal data processing, there is sufficient state machine model.

Behavior of individual objects is pictured throught the state machines. Interaction of objects, by the means of message interchanging is pictured in collaboration diagram. When the message interchanging is so large that the readability of the collaboration diagram is poor, the time line as other dimension is added. This type of diagram is called sequence diagram.

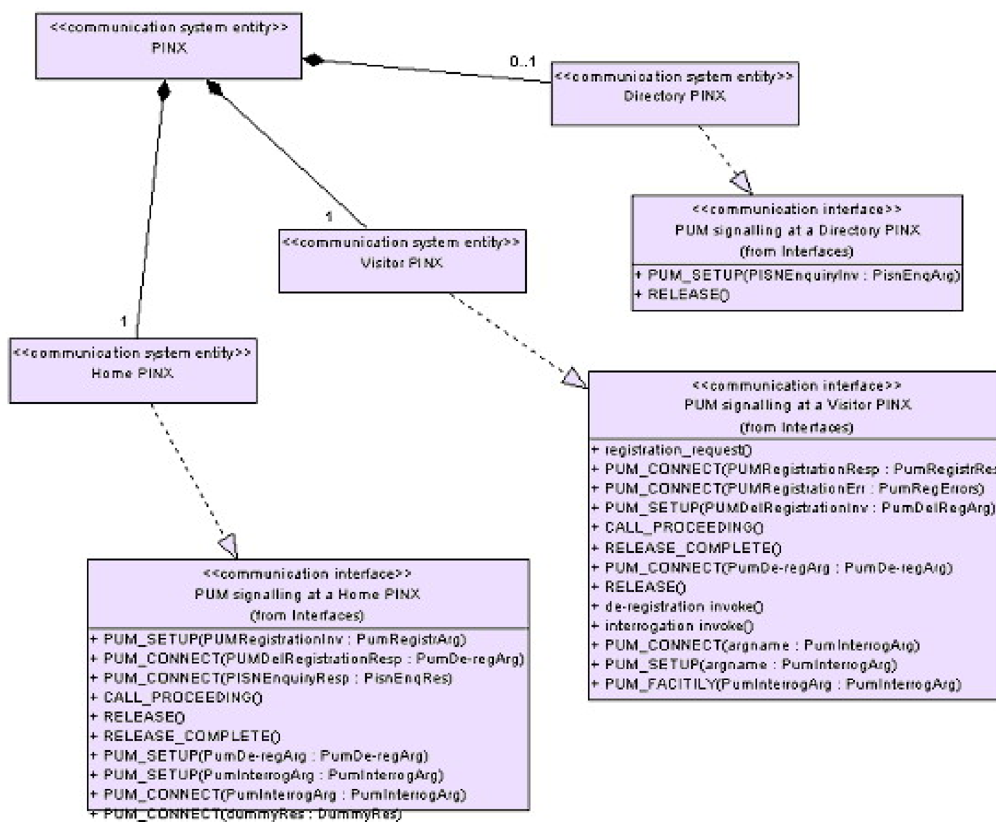

Figure 4.2: An example of UML Class diagram

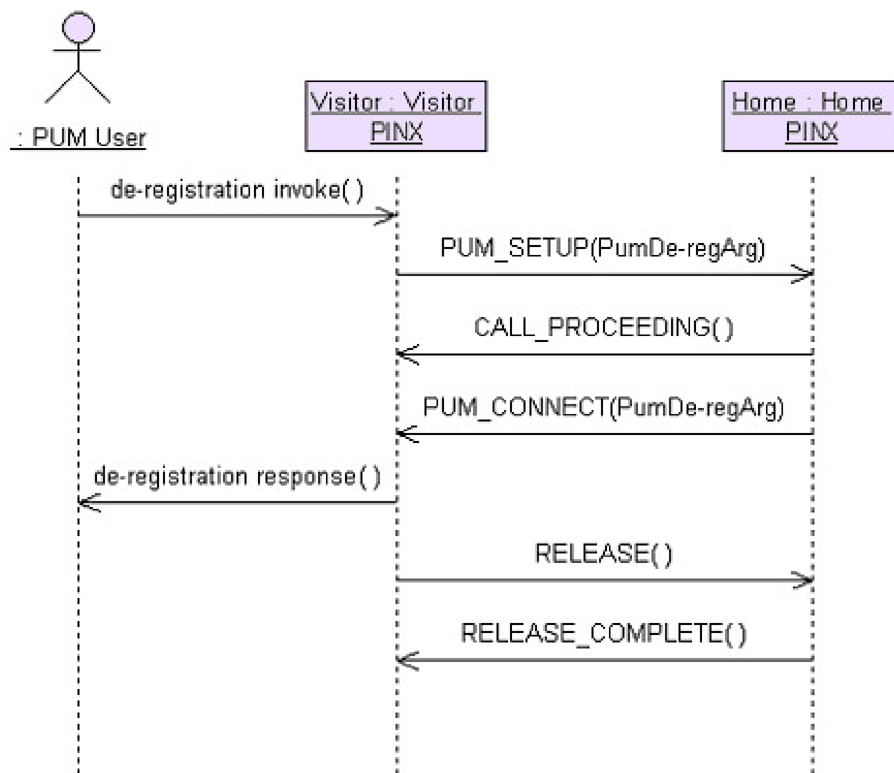

Figure 4.3: An example of UML Sequence diagram

4.1.4 Extension Mechanism

UML is design to be applicable to most of the types of systems. Of course for some special kinds of systems is UML not so suitable, for this purpose UML has an extension mechanism. By defining your own types of notation one can adapt UML to the concrete project. When extending the UML make sure the notation is stated clearly enough to be helpful and not confusing.

UML with its many diagrams can be really helpful when creating a system. Especially when creating a larger system. Graphs alone are not sufficient, the documentation to the graphs is necessary too and cannot be omitted. The quality of the textual documentation has a big impact on the quality of the whole model. UML supports documentation written in formal language OCL to raise the quality of model.

UML has an advantage in simple notation which is mostly intuitite so the graphs are readable also to customers laymen. Graphs included in a document can help to rise the readibility. UML is not hard to learn and the effectivness is very high that is why it is the most used requirements technique.

4.2 User Requirements Notation

In February 2003 the International Telecommunication Union (ITU-T, SG 10) approved standard Z.150 User Requirements Notation (URN) - Language requirements and framework for description of requirements for future telecomunication systems and services [9, 10]. It is standard requirements notation for complex reactive, distributed and dynamic systems and applications.

URN contains two languages Use Case Maps (UCM) [11] for capturing functional requirements and Goal-oriented Requirements Language (GRL) [12, 13, 14, 15] for non-functional requirements. URN is a graphical notation at high level of abstaction, designed for the use at early stages of requirements specification. It does not require to define messages, components and components states. URN supports reusability of scenarios, traceability and transformation to other languages such as Message Sequence Charts (MSC) or Unified Modelling Language (UML). URN deals with non-functional requirements and specifies the relationships between non-functional and functional elements. The prime application domain of URN is telecommunication services, that is the reason why it's application in other domains is not explored.

4.2.1 Goal-oriented Requirements Language

It is the attempt to formalize non-functional requirements. Failures of software projects were often caused by mistakes in informal non-functional re-

quirements. Formalization of requirements capture leads to formal documentation, validation and testability. GRL provide requirements in terms of objects and desired goals. Representing goals in GRL makes it possible to analyze various alternatives. GRL contains three main categories of concepts - intentional elements, links and actors.

Notation

Intentional Elements. The primar concern of intentional elements is "why?". Why particular behaviours and structures were chosen, what are other alternatives, what are the reasons of chosing one alternative among the other, which criteria were taken into account.

There are five kinds of intentional elements :

- Goal : Quantifiable high-level (functional) requirement (illustrated as a rounded-cornered rectangle).
- Softgoal : Qualifiable but unquantifiable requirement, essentially non-functional (illustrated as a cloud).
- Task : Operationalized solution that achieves a goal, or that satisfices a softgoal which can never be fully achieved due to its fuzzy nature (illustrated as a hexagon).
- Resource : Entity whose importance is described in terms of its availability (illustrated as a rectangle).
- Belief : Rationale or argumentation associated to a contribution or a relation (illustrated as an ellipse).

Intentional Relationships. There are five kinds of intentional relations, which connect elements :

- Contribution : Describes how softgoals, tasks, beliefs, and relations contribute to each other. A contribution is an effect that is a primary desire during modelling. Each contribution can be qualified by a degree:
    - AND : The relations between the contributing elements are 'AND'. Each of the sub-components is positive and necessary
    - OR : The relations between the contributing elements are 'OR'. Each of the sub-components is positive and sufficient.
    - MAKE : The contribution of the contributing element is positive and sufficient.
    - BREAK : The contribution of the contributing element is negative and sufficient.

  - HELP : The contribution of the contributing element is positive and but not sufficient.
  - HURT : The contribution of the contributing element is negative but not sufficient.
  - SOME+ : The contribution is positive, but the extent of the contribution is unknown.
  - SOME- : The contribution is negative, but the extent of the contribution is unknown.
  - EQUAL : The equal contribution in both directions.
  - UNKNOWN : There is some contribution, but the extent and the sense (positive or negative) of the contribution is unknown

- Correlation : Contribution that indicates side-effects on other intentional elements (dashed line)
- Means-end : Link for tasks achieving goals. Different alternatives are allowed.
- Decomposition : Defines what is needed for a task to be performed (refinement), always AND.
- Dependency : Link between two actors depending on each other (half-circle).

Actor.

An actor is an active entity that carries out actions to achieve its goals. an actor is represented by a circle. An actor can have a boundary, with intentional elements inside, graphically shown as a grey shadow.

Model

Modelling with GRL notation begins with determination of the high-level goals. These goals can be accompanied with beliefs. The high-level goals are divided into smaller goals, marking the right degree of relationship. Then the positions of actors and their boundaries are stated. Other alternative in modeling is to begin with defining actors and then binding the particular goals to the actors.

The ambition of GRL is to make the writer find all possible alternatives about how can be the high-level goals satisfied, to think about their relationships and interactions wheather they are positive or negative.

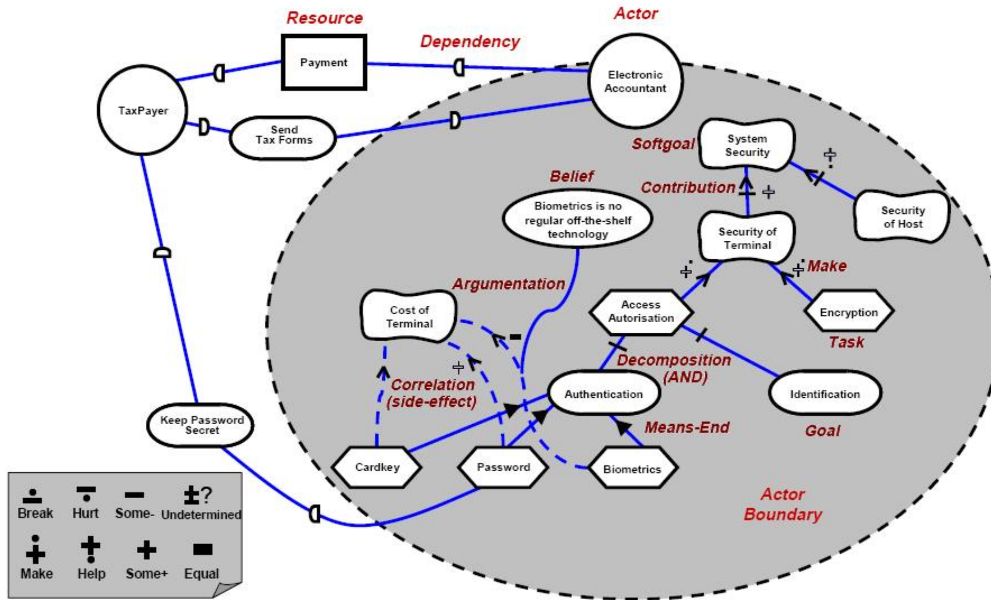

Figure 4.4: An example of GRL model

Evaluation

The first step in evaluation is to decide on the satisfaction of the goals and tasks that represent leaves in the GRL model. Then by the propagation algorithm, using logic of degrees of the relationships, evaluate the model bottom-up. One can try more possible decisions of satisfing goals and tasks at the bottom, to see their impact on the satisfaction of the high-level-goals.

Tool Support

The Organization Modelling Environment (OME) which supports i* and NRF framework was extended also to support GRL framework and GRL export to XML. This goal-oriented and agent-oriented modelling and analysis tool is written in Java language and was developed at the Knowledge Managment Lab at the University of Toronto.

4.2.2 Use Case Maps

Use Case Maps (UCM) as a part of URN is a simple visual notation for capturing functional requirements. It's primary usage is in concurrent and real-time systems. The big advantage of UCM is its independence from internal components and other design details. The use of UCM is at first phases of software development.

Notation

The basis of UCM notation are set of paths along which are defined responsibilities of the system.

Starting point Graphically represented by filled circle. Starting point is the beginnig of the path. It is set by preconditions and triggering event.

Path The line between the starting point and the end bar. Path is devided into path' segments. To each segment a responsibility is joined.

Responsibility In the case of UCM, responsibilities are characterized by short textual description, along with unique identifier. Responsibilities are marked by small letters from the beginning of the alphabet.

End bar Vertical bar, which represents the ending point in the execution of the path. The ending point is set by postconditions, output parameters, and resulting events.

Waiting place Graphically represented as filled circle (the same as starting point). The execution of the path is stopped until the conditions set by the waiting place are satisfied or a triggering event occurs.

Timer is a special kind of waiting place, where a time period is set, after which, if still waiting the execution continues by the timeout path. The timer is used to prevent deadlock. (graphically represented as a clock icon)

Stub Stub represents a path abstraction. Complicated part of path can be described on a separate map. The original part is replaced by a symbol of the stub (diamond). This construction allows moddeling at many level of detail and allows better readability. The function of the stub is extended by dymanic replacing of stubs. The map used in extraction is dynamically chosen from a set of maps at the time of execution.

UCM Model

The modelling UCM starts with main path with it's responsibilities, and then alternative paths are added.

The place where two paths are joining together into a single path is called Join. In the OR-join the execution of single path begins when the execution of one of the starting path comes to the joining point. In AND-join the execution of single path starts when executions of both starting paths comes to the joining point. AND-join is mostly used for synchronization.

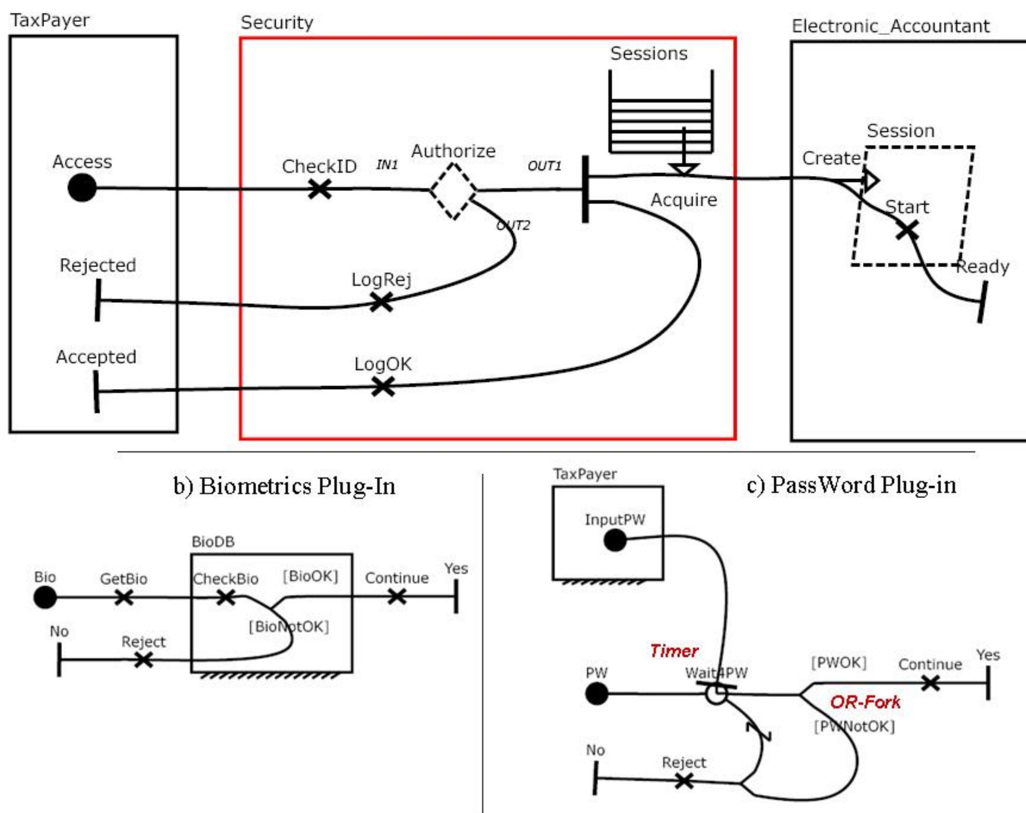

Figure 4.5: An example of UCM model

Fork is when one path is splitting into two different paths. In OR-fork the execution continues only in one of the paths, depending on the conditions stated in the splitting point. In AND-fork the execution continues in both paths simultanously.

The OR's are graphically represented by simple crossing. The AND's are graphically represented by vertical line, to emphasize the need of synchronization.

There also can be presented an interaction of two or more paths, without actual joinning. These are the cases when starting points (waiting points) of one path is dependent form the execution of other paths. It is graphically repesented by drawing the line near the starting (waiting) point. There is also a special sign for abort. In this case the execution of one path interupts the execution of another path. Graphically represented as lightning.

When the UCM model is finished we can bound this model to the model of the system with possible components.The System is drawn as a rectangle, with smaller rectangles inside representing different components. The path segments are drawn inside the components, to show the responsibility of the components in execution of particular path segment.

Tool Support

The tool support of UCM can be found at the www.usecasemaps.org. Here you can find UCM navigator called UCMNav, which is a tool made by the student as his M.Eng thesis in 1998. From this time many other people participated in adding new features to the navigator. The group of people working around this tool have decided to begin working on a new generation of navigator, in a form of Eclipse platform. This platform is called jUCMNav, and is also available for download on UCM web page.

URN as a notation is simple and easy to use. The advantage is in the elaboration of the nonfunctional requirements. The main difference between Use Case Maps of URN and Use Case diagrams of UML is in the level of detail. UCM is designed only for low level of detail, the use cases in UML can be used in various level of detail. Higher level of detail with UCM can be achieved by transformation into Message Sequence Charts (MSC) as an alternative to collaboration diagram in UML. MSC is also an ITU-T language. The transformation can continue into SDL language, which is a combination of graphical technique with formal description. These languages were created as a complements to each other and the transformation is supported by ITU-T products. The advantage in choosing a family of languages is in their collaboration and their big coverage of the requirements analysis process.

The effectiveness of graphical models is very high. Drawing a model can

help to better vizualize the system as a whole. Drawing a grahical representation of whole system can help finding gasps that were not so obvious before. Graphical models with their simplicity are the best way when starting using techniques in requirements analysis.

Chapter 5

Formal Methods

The primar reason of using mathematical formalism in requirements engineering is the quality of the requirements and the possibility of automatisation of some of the processes, especially some test cases. The ambiguity is the biggest problem within requirements engineering and the exact science removes it.

Formalizing of the requirements can also be used in automatic test case generation, and formal proof can replace many test cases. On the other hand the formal specification does not show the correctness of the entire system. The end-to-end tests are still to be done individually. Also good design of the system is needed for system to be correct. The reason why formal techniques are not used so often is in the cost of the formal specification. Many think that the use of formal methods raises the price of the software system, the fact is that the formal specification takes more time and so money, but this time is spared in the phase of software development and maintainance.

The notation used in formal methods is more complex that in graphical models, but because of usage of basic mathematical symbols it is not so hard to learn, and does not require trained mathematicians. The use of formal methods is recommended in every system where the correctness is wanted.

Formal models techniques can be divided into two main categories. Model-based formal methods and property based formal methods.

The model based formal methods creates the model of the system describing the different states of the system and possible operations that can be made on a particular state.

Property based formal methods decribes operations of the system and their relationships. These methods are based on process algebra. Algebraic specification defines syntax of operations and axioms of the system. [16, 17, 18]

5.1 Z notation

Model based formal methods build an abstract model of the system and specify the operations being made. In model based formal specification are used set theory, function theory and logic. The models made are idealized, trying to be free from implementation bias.

The model of the system consists of possible states of the system, accompanied by possible operations of the system and indication of the change of the state.

The most popular model based formal methods are Vienna development method (VDM) and Z notation [19, 20]. They are both based on mathematical set theory, but have different syntaxes. The VDM was the first formal method ever used for large scale project. The Z notation was also used for large scale projects, and it became IBM's main formal specification tool.

Z notation was proposed in 1977 by Jean-Raymond Abrail. Further development continues at Programming Research Groups at Oxford University. Z notation became an ISO standard in 2002. Z is formal specification that is independent from program code and can be completed early in the development. Z notation uses predicate logic. The requirements are decomposed into small pieces called schemas. In Z notation we distinguish two kinds of schemas, static and dymanic.

Static schemas describes all the possible states of the system and invariant relationships. The static schema consist of name of the schema. The top part of the schema contains declarations of variables. At the bottom part the system invariants as relationships between the values of the variables are declared. System invariants are true in every state of the system.

Dynamic schemas represents the operations. The name contains Δ as a symbol of the state change. The top part includes observations before the state change. The input variables names ends up with ? to be clearly recognized. The bottom part contains observations after the state change.

The properties defined in the operations are proved by the invariants using a predicate logic.

With the schemas it is possible to describe the whole system. The description is so specific that it is only a step from real program language. The schemas can be rewritten as procedures. This kind of specification is the bridge between the written code and the specification. It is writting a pseudocode, that is not so exact as code, but has all it's attributes. The most used one is the possibility of automatic test generation and possibility of prove.

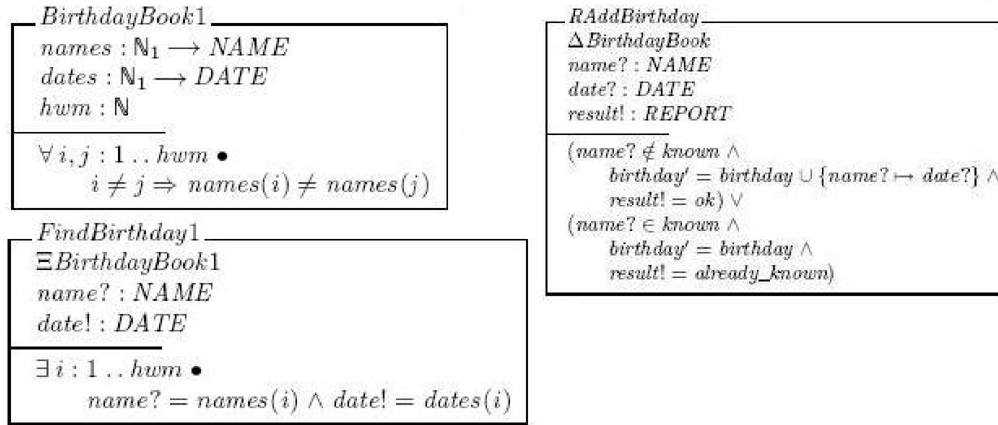

Figure 5.1: An example of Z schema

5.2 KAOS

KAOS is goal-oriented formal language [21, 22, 23]. The KAOS models contains both graphical representations of system and formal mathematical description of the objects.

KAOS contains four types of models. Goal model specifies the goal and its subgoals. The goals with its subgoals are drawn as a derivation trees. The derivation of goal continues while the identification of the agents, responsible for the leafs of the derivation tree, is not explicit.

Agents are responsible for achieving goals. Agents can be users of the system or parts of the software. In the Agent model the agents are assigned to the goals. Operation model specifies the operations made by agents to achieve the goals. The operations are described by the input, output, the state of the system before the operation starts and the state of the system after the oparation ends.

Object model identifies the objects used in KAOS models, such as entities, agents, relationships. A formal specification is used for dascription of the objects in KAOS.

Objectiver is a software used for tool support of KAOS model. This tool supports generation of software requirements specification from the KAOS model, which is heplfull when writting the specification document.

It is relevant to think about KAOS model for projects where the requirements analysis takes form 4 to 8 man months. The time spent using KAOS is around 3 man months and the cost is 10

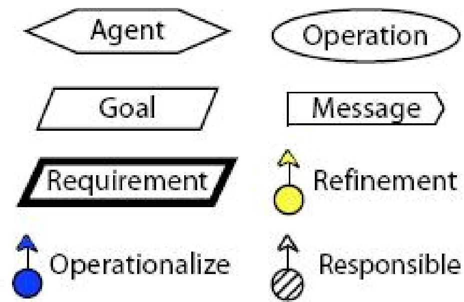

Figure 5.2: KAOS notation definition

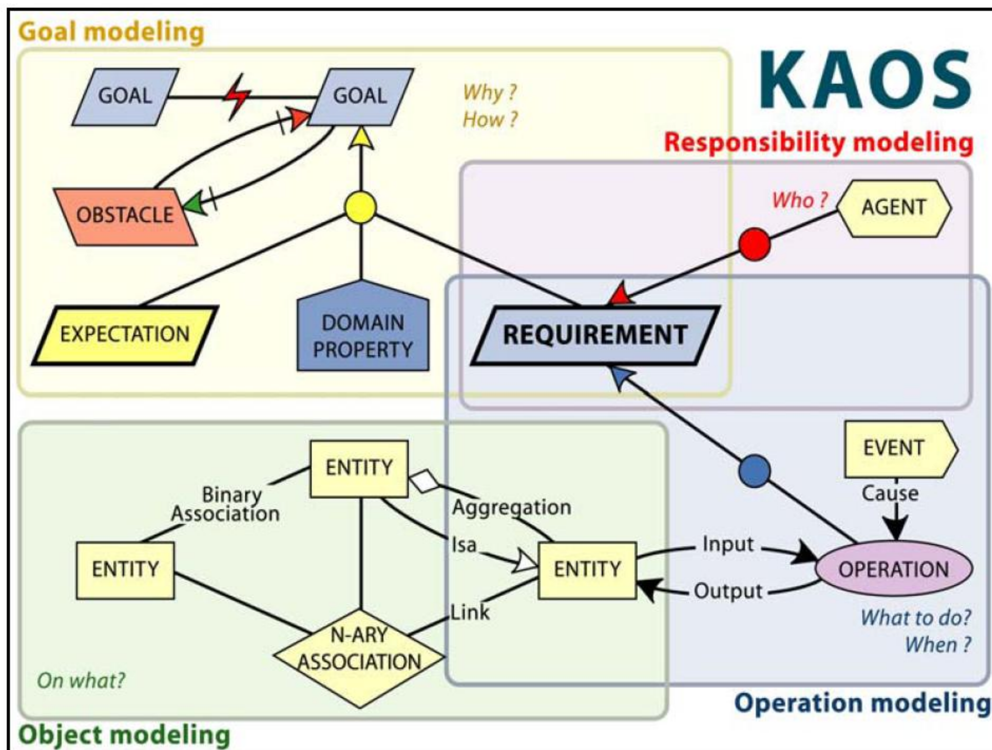

Figure 5.3: KAOS modelling example

Requirement In_DES_64_State
Concerns S_{SPEC} where $S_{SPEC} = DES64_{SPEC}$
Refines *Achieve*[Adapt_from_DES_64_to_DES_128]
InformalDef The program satisfies S_{SPEC}.
FormalDef $S_{SPEC} = DES64_{SPEC}$

Goal *Achieve*[Adapt_from_DES_64_to_DES_128]
Concerns $S_{SPEC}, T_{SPEC}, A_{REQ}$ where
$S_{SPEC} = DES64_{SPEC}$ $T_{SPEC} = DES128_{SPEC}$
$A_{REQ} = Request_64_to_128_Onepoint_{REQ}$
RefinedTo In_DES_64_State ; In_DES_128_State
InformalDef The program initially satisfies: S_{SPEC}.
When it reaches a safe state all obligations generated by
S_{SPEC} are satisfied.
FormalDef $S_{SPEC} \wedge \Diamond A_{REQ} \overset{\Omega}{\rightharpoonup} T_{SPEC}$

Operation Request64to128Onepoint
Input The adaptive system status = status of MetaSocket
Output A_{REQ}
DomPre S_{SPEC}
DomPost $S_{SPEC} \wedge A_{REQ}$

Figure 5.4: An example of formal description of entities in KAOS

5.3 Usage of Formal Methods

The use of mathematical model eliminates the ambiguity from the requirements specification. With the help of mathematical proving methods many expected properties of the system can be proved. This helps to find mistakes made in the analysis. It is important to realize that not everything can be proved and also that there can be mistakes made in the proofs. By testing the presence of the mistakes is proved not the absence. In the formal proofs the correctness of the system is prooved, and if the mistakes are made they are easier to find.

The formal methods are used more often in large systems, where the collaboration of many programers is needed. The use of formal methods is rising. Nowadays for building most of critical systems the use of some kind of formal method is required. The formal specification is used whenever the correctness is important. The problem with formal methods is that it takes time to make a good specification. Even if this is returned in the maintainance phase, the managers don't take this into account.

Due to lack of experience the estimations made are not so precise as expected. This fact often leads to disillusion from the use of formal methods.

Formal specification can be used also for generating a natural language specification. By automatic generating the specification from the formal model one can avoid the main problems of the natural language written specification. When generated from formal model there is no place for ambiguity. It is complete and consistent.

Chapter 6

Choosing the Right Technique

When finding the right modelling technique for particular software system, one should consider many different aspects.

6.1 Considered properties

- Size and complexity of the system
- The level of abstraction of the model
- The applicability of the model for special kinds of systems
- Support of the non functional requirements analysis
- The possibility of modelling technique to capture the most important parts of software
- The features of the tools available
- Traceability
- The experience of the analysits, programers
- The experience of the customer

The expected complexity and size of the system is the main aspect to be considered. Smaller systems are mostly simplier and the work around is not so big. Complex modelling techniques are not necessary to be used here, and often spare more time than clasical analysis.

For more complex systems it is recommended to use more complex modelling techniques. The use of structural language, when writting down the basic user requirements to set out the main functionality of the created software, is important. Structural language can truncate the amount of text

needed. When good writting styles are used the understandability of the text is even higher than writting with classic sentences.

For deeper analysis the usage of graphical models is recomended. Our brain works with images that is why we better and faster understand the images than paragraphs of text describing the image. The graphical representation can be heplful when comunicating with customers - laicks.

Graphical models do not consist of images alone, written description is also needed. The quality of written description has impact on the quality of the whole model. The deeper is analysis the more time is spend, but the implementation is easier than. Graphical models have different kinds of the depth of the analysis. The wanted depth of the analysis of the requirements should be also considered when chosing a right modeling technique.

UML model contains many kinds of diagrams, which can be used with different depth of analysis. On the other hand the URN language supports only the upper level of the requirements analysis. To support deeper analysis URN has an option of transformation into MSC language, which goes into more detail. The family of ITU-T languages ends up with graphical model using formal methods SDL. On the ITU-T family is best the support and coordination of the languages.

I consider as an advantage the GRL model, which as one of a few is esspecially made for capturing non functional requirements. There are not many languages considering non functional requirements. Non functional requirements are important, because are mostly crucial for software acceptance by the customer. When using a model not considering non functional requirements at all, there is a high chance that this can be forgotten and the created software is out of scope of the requirements. As a disadvantage of ITU-T languages I consider their applicability. They are designed only for telecomunication systems and services. This can be considered as an advantage when choosing a model for telecomunication systems and services. It is always better to choose a specific model for the wanted kind of system, than the general one.

The notation available in modelling technique should be also considered. In every kind of software one should concentrate on the most complex part of the software, and find out if the notation of the modelling technique decsribes this part enough to make it clear. Not everyhing can be modelled. One have to realise that it is only a model, the real system is more complicated. The model should emphasize the most complex parts of the system.

With choosing a modelling technique hand by hand one need to choose also a software that supports the technique. The rules are the same as choosing any kind of software. One should find references on this software, if there are not any hidden troubles that could affect the usage for a specific kind of software.

The traceability of the modelling technique in other development is also important. For example the eclipse platform which supports the UML modelling is able to generate a headers of the needed procedures in Java language. This feature of the tool supports the programers. Many of the more complex techniques supports the test case generation. When using the formal methods unit test can be even automated. The ability of the tool to generate a readable documentation is also important. When writing of the documentation is automated, lot of time is spared. In more complex systems the documentation has more than hundreds pages, which is hard to maintain.

The experience of the people working with the model is also decisive. When the analysits are used to model using one kind of technique, it is propable that when the experience is good, they will continue in using this technique in future projects. When considering a new technique the analysits should be trained to be able to use it properly. This also spends time and money, which is important for the managers to consider also. Not only analysits creating model should be trained. There are also programers, who work with the model. The programers opinion should be considered to.

The model of the system is a basis for the documentation needed by the customer. This documentation is often a part of contract, so it must be written in a style that is understandable for the customer. If the customer is used to more detailed specification it is an advantage for the analysts. They can discuss all details of the proposed system and so can find many mistakes. If the customer needs to approve complex analysis document, which he is not able to understand, he is not able to consider whether it has the expected functionality.

Technique	scope	domain	abstraction level	notes
QuARS	natural language description	general	low	university project. not widely used
BADGER	natural language description	general	low	university project. not widely used
Circe	natural language description	general	low	used for validation of requirements
UML	graphical model, supported by textual description	general	not defined	the most used graphical model
URN	graphical model, supported by textual description	telecomunication services	low	supported by ITU-T, dealing especially with non-functional requirements
MSC	graphical model, supported by textual description	telecomunication services	medium	supported by ITU-T
KAOS	graphical model, supported by formal description	general	high	can be used from the beginning of the requirements analysis, automatic validation of the model supported
VDM	formal model	general	high	first used in large scale project, very similar to Z notation
Z	formal model	general	high	the most used formal method
SDL	graphical model, supported by formal decsription	telecomunication services	high	supported by ITU-T
OBJ	formal model	general	high	family of languages based on algebra and logic

Table 6.1: Overviw of basic properties of modelling techniques

Chapter 7

Conclusion

In my diploma thesis I outline the life cycle model of software system to show the importance of the requirements engineering as a part of software engineering. I presented an IEEE Recomendation for a Software Requirements Specification to show a reader all the properties of a good quality requirements documentation. These properties are emphasized to help reader to realize the aspects of natural language written documentation and its problems.

I presented modelling techniques, that are used for capturing software requirements. I show different approaches to give reader an overall overview of specification techniques, to realize all different possibilities in choosing a modelling technique. I presented techniques that works with the natural language description to show what can be done only by proper formulation of the sentences.

The techniques using graphical notation were presented as an example of most used techniques, that are also heplfull for customers for better understanding of software description.

At the end were presented techniques using formal description, which are slowly finding their users due to the need of mathematical notation. The formal methods are forcing analysits to model the system to the deepest details, which spends a lot of time. I tried to show that formal methods are not just a teory, and the obstacles are not so big as they are presented.

Bibliography

[1] http://codebetter.com/blogs/raymond.lewallen/archive/2005/07/13/129114.aspx.

[2] Ian Sommerville. *Software Engineering, 6th Edition.* Addison-Wesley, 2002.

[3] Software Engineering Standards Committee of the IEEE Computer Society. Std 830-1998 ieee recommended practice for software requirements specifications. pages 1–20, 1998.

[4] G. Lami M.Fusani, S.Gnesi. An automatic quality evaluation for natural language requirements.

[5] Stephen Cook William Scott. A context-free requirements grammar to facilitate automatic assessment.

[6] Bruce Powel Douglas. The uml for systems engineering. *http://www.nohau.se/articles_ e*, pages 1–12.

[7] http://www 306.ibm.com/software/rational/uml/.

[8] http://portal.etsi.org/mbs/Languages/UML.

[9] Daniel Amyot. Introduction to the user requirements notation: Learning by example. *Computer Networks: The International Journal of Computer and Telecomunications Networking*, 42:285–301, 2003.

[10] Daniel Amyot. Requirements engineering & user requirements notation. *http://www.UseCaseMaps.org/urn/*, pages 1–46, 2002.

[11] http://www.usecasemaps.org.

[12] K. Saleh A. Al-Zarouni. Capturing non-functional software requirements using the user requirements notation. *The 2004 International Research Conference on Innovations in Information Technology*, pages 1–9, 2004.

[13] http://www.cs.toronto.edu/km/GRL.

[14] Gunter Mussbacher Daniel Amyot. Urn: Towards a new standard for the visual description of requirements. *www.usecasemaps.org/pub*, pages 1–17, 2002.

[15] Study Group 10. Draft specification of the user requirements notation. *ITU - Telecomunication Standardization Sector*, 2000.

[16] Ian Sommerville. *Requirements Engineering: Processes and Techniques.* John Wiley & Son Ltd, 1998.

[17] Anthony Hall. Seven myths of formal methods. *Software, IEEE, Volume 7, Issue 5*, pages 11–19, 1990.

[18] Michael G.Hinchey Jonathan P.Bowen. Seven more myths of formal methods. *Fifth European Software Engineering Conference*, pages 34–41, 1995.

[19] J. M. Spivey. The z notation:a reference manual. *J. M. Spivey, Oriel College 1998*, pages 1–168.

[20] http://en.wikipedia.org.

[21] A kaos tutorial. *http://www.objectiver.com/en/documentation/kaos/*, pages 1–42.

[22] Denis Ballant. Modeling project requirements with objectiver. *http://www.objectiver.com/en/documentation/kaos/*, pages 1–12.

[23] Heather Goldsby Ji Zhang Greg Brown, Betty H.C.Cheng. Goal-oriented specification of adaptation requirements engineering in adaptive systems. *http://www.objectiver.com/en/documentation/kaos/*, pages 1–7.

Abstrakt.

Proces tvorby softvérového systému sa so zväčujúcou komplexitou systému stáva komplikovanejším. Vzniká potreba riadenia činností súvisiacich s tvorbou softvéru. Vytvoril sa model životného cyklu softvéru. Všeobecne by sa dal cyklus rozdeliť na päť základných fáz : zber a analýza požiadaviek, design, implementácia, testovanie a údržba.

Každá z fáz je pre vývoj softvéru dôležitá. Chyby vytvorené v tej ktorej fáze ovplyvňujú fázy nasledujúce. Snahou je zachytiť chyby čo najskôr, preto je kladená dôležitosť na kvalitu požiadaviek, ktoré ovplyvňujú celý proces vývoja softvéru. Chyby vytvorené vo fáze zberu a analýzy požiadaviek často vedú k vývoju softvéru nepoužiteľného zákazníkom. Ak v zbere požiadaviek dôjde k nedorozumeniu so zákazníkom,a v popise špecifikácie požiadaviek sa nedostatočne popíše. Takéto nedorozumenie sa odhalí až v procese testovania zákazníkom. Čo je však dosť neskoro, kedže boli už vynaložené nemalé prostriedky na implementáciu. Takýmito zbytočnými chybami sa minú peniaze a hlavne aj kopa času. Často sa stáva že tieto náklady v plnej miere nesie vývojová firma. Aby sa predišlo zbytočným chybám je potreba poriadne popísať vyvýjaný softvér ešte pred začiatkom implementácie.

S definovanm požiadaviek bol vždy najväčší problém, kedže je potreba komunikácie so zákazníkom-laikom, ktorý často nevie správne formulovať požiadavky. Preto je potreba spísať požiadavky vo forme prijateľnej pre zákazníka. Požiadavky treba popísať spôsobom, aby sa vyhlo nedorozumeniam. Problém pri použití hovorového jazyka je z jeho nejednoznačnosťou. Použítie štruktúrovaného jazyka a jednoduchých viet pomáha v opise. Pri písaní je potreba použiť formulácie, ktoré čo možno najjednoznačnejšie opisujú systém. Treba sa vyvarovať slovných spojení ktoré vedú k nejednoznačnosti a čo možno najkonkrétnejšie popísať systém.

Vyvinulo sa niekoľko rôznych systémov ktoré pomáhajú pri používaní hovorového jazyka. Napríklad taký QuARS je postavený na lexikálnej analýze písaného textu. Text je rozdelený na lexémy, a následne kontrolovaný oproti nadefinovanému slovníku. Systém upozorňuje na slovné formulácie ktoré vedú k nejednoznačnosti písaného textu. Tento nástroj je tak dobrý ako dobre je nadefinovaný slovník. Aj keď sa systém konkrétne nepoužije, je poučné oboznámiť sa s formuláciami vedúcimi k nejednoznačnosti a pokúsiť sa ich v písanom texte nepoužívať.

Ďalší spôsob kontroly písaného textu je kontrola priamo pri písaní. Vety sú tvorené priamo podľa predlohy. V systéme BADGER bola predloha vyvinutá na základe správnych formulácií a zaznamenaná pomocou formálneho jazyka použitím bezkontextovej gramatiky. Písnie podľa takejto predlohy je obmedzujúce a teda aj náročnejšie ale výsledok je viditeľne lepší. Použitá formulácia je oveľa presnejšia a nie príliš všeobecná. Nejednoznačnosť je oveľa menšia a tak aj možné chyby sú eliminované. Použítie nástrojov na

úpravu písaného textu je jednoduché a nie veľmi časovo náročné. Je všk potreba si uvedomiť, že nejednoznačnosť sa nedá celkom odstániť.

Ďaľším spôsobom ako zlepšiť jednoznačné pochopenie popisovaného systému je použitie grafického modelu. Grafické modely majú veľkú výhodu, sú oveľa priehľadnejšie ako písaný text. Ľudský mozog pracuje s obrázkami a preto je použitie obrázkov na vysvetlenie problematiky oveľa lepšie ako kopa písaného textu. Grafické modely sú jednoduché a ich notácia je intuitívna, takže nie je potreba dlhodobého školenia. Notácia je porozumiteľná aj pre laikov - zákazníkov.

Najčastejšie používaným grafickým modelom je UML Model. Grafický model systému pozostáva z viacerých diagramov, každý z iným uhlom pohľadu. Tento model obsahuje diagramy popisujúce vonkajšiu, hrubú funkcionalitu systému, dátový model systému, štrukturálny model zachycujúci vzťahy medzi jednotlivými softvérovými časťami systému ako aj previazanie softvérových častí na hardvérové komponenty. UML obsahuje aj diagramy zachycujúce správanie systému a to pomocou popisu výmeny dát medzi jednotlivými objektami, alebo popisom reakcií časti systému na vonkajšie impulzy. Pre každý typ softvéru je vhodné použitie iných diagramov. Diagramy UML však nie sú dostačujúce. Každý objekt v modeli je doplnený popisom. Tento popis má predpísanú štruktúru, aby sa zvýšila kvalita popisovaných počiadaviek. UML model podporuje aj popisovanie formálnym jazykom OCL. Kvalita popísaných požiadaviek závisí aj od hĺbky modelovania systému, kedže UML nepredpisuje do akej hĺbky ten ktorý model zasahuje.

Pre porovnanie som si vybrala modelovací jazyk URN. Jedná sa o jazyk vyvíjaný špeciálne pre telekomunikačné systémy. URN pozostáva z dvoch modelov GRL model zachytáva nefunkčné požiadavky na systém, UCM zachytáva funkčné požiadavky. Nefunkčné požiadavky sú pre vývoj veľmi dôležité, majú vplyv na výber použitých technologických riešení. UCM model zachytáva základnú funkcionalitu systému. Je to ekvivalent k Use Case diagramu v UML modeli. Pre hĺbšiu analýzu je potreba transformácie na ďaľší z ITU-T jazykov MSC. MSC jazyk je obdobou sekvenčných diagramov UML. Popisuje výmenu dát medzi jednotlivými komponentami systému. ITU-T jazyky podporujú ešte transformáciu MSC na grafický model s formálnym popisom SDL jazyk.

Výhodou ITU-T jazykov je ich priama nadväznosť a jednoznačná hĺbka zachytenej analýzy. Tieto jazyky sú však jednostranne orientované na analýzu telekomunikačných systémov, čo obmedzuje ich použitie. UML model je všeobecný vhodný pre väčšinu systémov, preto je aj najpoužívanejší spomedzi grafických modelov. Ak sa však firma jednostranne orientuje na vývoj špecializovaných systémov určite je vhodnejšie vybrať si modelovací jazyk ušitý na mieru.

Jednoznačné eliminovanie nejednoznačnosti je možné len použitím matematických formálnych modelov. Na popis požiadaviek sa používa množinová teória, logika, ale aj pseudoprogramovacie jazyky. Vzhľadom na komlikovanú notáciu je potreba viac času na naštudovanie notácie. Použitie formálnych metód je veľmi zriedkavé. Najčastejšie používaným formálnym jazykom je Z jazyk. Z je používaný IBM ako základný formálny jazyk. Z používa na modelovanie systému schémy. Schémami popisuje jednotlivé stavy systému ako aj prechody medzi týmito stavmi. Modelovací jazyk KAOS je príkladom spojenia grafického modelu a formálneho popisu. KAOS vytvára model systému pomocou cieľov. K cieľom sú priradené časti systému, ktoré zodpovedajú za vyplnenie jednotlivých cieľov. Jednotlivé časti systému sú popísané formálným jazykom. KAOS podporuje generovanie dokumentácie s vytvoreného modelu, čím šetrí čas potrebný pre spísanie dokumentu potrebného pre zákazníka. Formálne jazyky sa používajú hlavne kôli možnosti automatického modulového testovania.

Pri výbere modelu na zachytenie požiadaviek je potrebné vziať do úvahy viacero faktorov. V prvom rade je to veľkosť a komplikovanosť systému na určenie potrebnej hĺbky analýzy. Pre menej komplikované systémy nie je potreba hĺbkovej analýzy. Komplikovaná analýza by zbytočne predražila jednoduchý systém. Naopak pri väčšom systéme investície do analýzy sú nutnosťou. Model by mal čo najlepšie zachytiť hlavne najzložitejšie časti systému. Schopnosť modelu zachytiť podrobne zložité časti systému by malo byť to najdôležitejšie kritérium pri výbere modelu. Popri výbere modelu je potreba vybrať si aj nástroj na modelovanie. Tu treba vziať do úvahy možné technické nároky na počítač, dostatočná podpora tímovej práce, a náročnosť na ovládanie.

Vývoj softvéru je komplexná vec. Každá fáza vývoja je dôležitá. Zber a analýza požiadaviek je často podceňovaná. Popis požiadaviek je podkladom pre programátorov a uľahčuje im prácu ak je kvalitný. Požitie modelov na zachytenie požiadaviek pomáha pri analýze uvodomiť si potrebné súvislosti a tak predísť zbytočným chybám.