

Diploma thesis

Developing formal technique for capturing requirements using the ForTheL language and the SAD system (Initial Draft)

Document History

Date	Version	Author	Change
16 march 2006	0.001	Petro Protsyk	Created, Introduction
17 march 2006		Petro Protsyk	Introduction
20 march 2006		Petro Protsyk	Requirements Engineering
21 march 2006		- / -	- / -
22 march 2006		Petro Protsyk	Goals and Tasks of Thesis, Requirements Specification
23 March 2006		- / -	Requirements Specification process , Validation , Management
24 March 2006			Corrections / NL tools
25 March 2006			ForTheL in OpenSpec => FTL approach
26 March 2006			Quality Requirements Analysis, Linguistic Analysis (notes 3.01.2006)
27 March			Linguistic Analysis
28 March			Linguistic Analysis (Finish), Correction
30 March			Corrections
4 April			Описание системы
11 April			Описание системы
17 April			Проектирование системы, программирование
18 April			Codding & Designing in Poseidon
23 April			ForTheL interface codding
16 May 2006		PPP (Zaporizhzhja)	System's design Translation
18 May 2006			Correction
28 May 2006			Formal Methods
28 May 2006			First Release
1 June 2006			Corrections

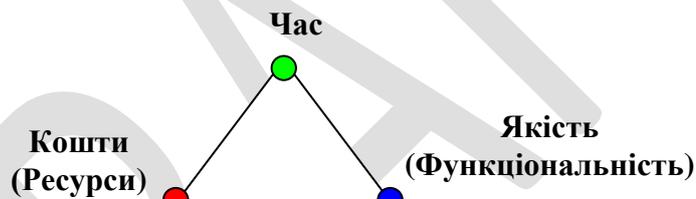
Contents

CONTENTS	2
BCTYII	3
INTRODUCTION	6
GOALS AND TASKS	10
THE REQUIREMENTS ENGINEERING	11
OVERVIEW	11
THE REQUIREMENTS ENGINEERING PROCESS.....	12
REQUIREMENTS ELICITATION.....	14
REQUIREMENTS ANALYSIS	15
REQUIREMENTS SPECIFICATION PROCESS.....	16
REQUIREMENTS VALIDATION.....	21
REQUIREMENTS MANAGEMENT.....	25
TOOLS FOR NATURAL LANGUAGE REQUIREMENTS ANALYSIS	26
TEMPLATE-BASED APPROACH	26
THE MODEL DRIVEN ARCHITECTURE METHODOLOGY	31
PROCEDURES OF LINGUISTIC ANALYSIS	32
FORMAL METHODS IN REQUIREMENTS ENGINEERING	39
DEFINITION	39
CLASSIFICATION.....	39
APPLICATION.....	41
FORMAL METHODS AND THEIR NOTATIONS.....	46
CONCLUSIONS	51
FORMAL THEORY LANGUAGE (FORTHTEL)	53
A BRIEF INTRODUCTION INTO FORTHTEL LANGUAGE	53
FORTHTEL-BASED APPROACH FOR REPRESENTING FORMAL SPECIFICATIONS.....	58
FORTHTEL IN REQUIREMENTS ENGINEERING.....	60
THE DESIGN OF SYSTEM FOR MANAGING REQUIREMENTS.....	67
CONCLUSIONS	73
DEFINITIONS	74
ABBREVIATIONS AND ACRONYMS	75
FORTHTEL AND SAD SYSTEM EXAMPLES	76
EXAMPLE 1.....	76
EXAMPLE 2.....	82
REFERENCES	83

Вступ

Сучасні програмні системи в більшості випадків - масштабні та складні у реалізації проекти. Існують різні моделі організації процесу їхньої розробки. Найбільш відомі - лінійна послідовна модель, каскадна (або водоспадна), спіральна, ітеративна (Rational Unified Process) моделі і їхні комбінації (Microsoft Solution Framework). Кінцева мета кожної з моделей - одержання якісного продукту (програмної системи), що задовольняє всі вимоги замовника, у встановлений час при оптимальних витратах ресурсів.

У дисципліні програмної інженерії цю проблему часто зображують «трикутником обмежень»(Малюнок 1: Трикутник обмежень):



Малюнок 1: Трикутник обмежень

Зміна кожного із зазначених на малюнку факторів неминуче спричиняє зміни у двох інших. Наприклад, при зменшенні ціни, тобто при зменшенні кількості розробників зростає час розробки та/або зменшується якість одержуваної системи.

Основне завдання менеджера програмної системи - правильно розподілити ресурси, з огляду на часові обмеження та вимоги замовника. Тому, першочерговою проблемою, не залежно від моделі процесу розробки є добування (збір) початкових вимог замовника до системи і до її програмної складової. На підставі первісних вимог визначається архітектура, засоби розробки, компоненти та структура майбутньої системи. Далі формулюються докладні вимоги до компонентів системи. Вимоги до компонентів повинні

узгоджуватись з вимогами замовника і уточнюватися по мірі їх деталізації. Процедури збору, уточнення, керування вимогами, як правило, визначаються моделлю процесу розробки. Продукт цього етапу - документи з детальним описом майбутньої системи, усіх вимог, специфікацій та інші документи. Як правило, існують різні версії документів для менеджерів, розробників, замовників та інших учасників розроблюваної системи. Кожна з версій документів повинна відображати інформацію найбільш важливу для відповідної групи задіяних у проекті. Для менеджерів – інформація, що дозволяє приймати правильні рішення, по керуванню проектом, для розробників - необхідна при розробці, для замовників - надаючи реальне уявлення про майбутню систему.

Описаний етап у розробці програмних систем одержав назву Requirements Engineering. Мета цього етапу - виявити, проаналізувати, верифікувати та задокументувати вимоги до майбутньої системи. Ключ до успішного виконання проекту багато в чому залежить від успішності виконання цього початкового етапу розробки. Він дозволяє розробникам і замовникам дійти згоди про те, які результати повинні бути досягнуті, а також уникнути помилок в майбутньому. Це твердження базується на припущенні, що чим раніше помилка буде знайдена, тим дешевше її виправити. А також тому, що можна побудувати стабільну модель системи (на підставі вимог) до початку процесу проектування та розробки.

Мета цієї роботи полягає дослідженні деякі з існуючих підходів обробки, формалізації та специфікації вимог до програмного забезпечення. Запропонувати підхід, заснований на мові ForTheL([3],[4]).

Формалізація вимог виконується для того, щоб виключити з них неоднозначність, неузгодженість та некоректність. Багато напівформальних та формальних мов були розроблені з надією забезпечити таку формалізацію. Загальний недолік існуючих підходів полягає в тому, що вони є складними для розуміння і досить специфічними. Це суттєво обмежує їх практичне

використання. Для того, щоб покращити рівень розуміння мови для неспеціаліста, в роботі пропонується використовувати мову ForTheL. Головні відмінності ForTheL – в тому, що вона є формальною мовою, та її синтаксис близький до звичайної англійської мови.

З моменту створення мова ForTheL була призначена для написання математичних текстів. Саме тому, інша важлива ціль цієї роботи полягає в тому, щоб дослідити можливості мови для збереження та запису вимог до програмного забезпечення. Далі їх можна буде обробляти з використанням системи автоматизації дедукції (САД). САД – це процесор текстів, написаних мовою ForTheL (<http://ea.unicyb.kiev.ua/sad.en.html>) та інших формальних мовах.

Систему САД можна розглядати, як сучасне бачення програми Алгоритму Очевидності, запропонованої академіком В.М. Глушковим. Він пропонував одночасно досліджувати використання формалізованих мов, для подання формальних текстів в формі, найбільш зручній для користувача, формалізації та еволюційного розвитку кроку доведення зробленого комп'ютером, вплив інформаційного середовища на крок доведення, та інтерактивної допомоги людини в пошуці доведення. Поточна реалізація системи САД може вирішувати різноманітні задачі по доведенню теорем, верифікації окремих ForTheL текстів, пошуку дерева виводу та інші.

В роботі розглядається використання цих функцій для перевірки специфікацій вимог. Приклади в роботі показують такі можливості системи.

Introduction

Modern software systems in most cases are large projects and difficult for implementation. There are various models of their development process organization. The most known are Linear Sequential Model, Cascade (or waterfall), Spiral, Iterative (Rational Unified Process) models and their combinations (Microsoft Solution Framework). The ultimate goal of each development process model is the developing of a qualitative product (program system) that is satisfying all customer requirements, during established time at optimum expenses of costs (resources).

The process of software systems development is studied by Software Engineering (SE) discipline [12]. The IEEE Computer Society defines software engineering as:

(1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.

(2) The study of approaches as in (1)

In the Software Engineering discipline the problem of process organization is frequently represented as «the constraint triangle» (Figure 1: The Constraint Triangle):

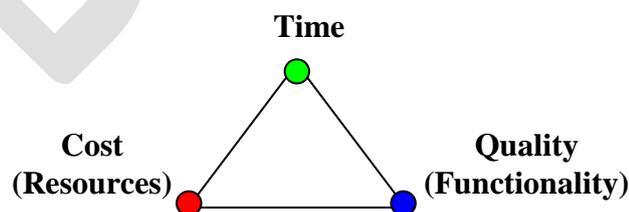


Figure 1: The Constraint Triangle

Any change to one of the goals specified on the figure inevitably entails changes to others two. For example, with the reduction of the price i.e. the

reduction of the developers' number the time of development grows and/or quality of received system is decreasing.

The main task of the software system manager is the correct planning of the resources, taking into account time restrictions and requirements of the customer. Therefore, a prime problem, that is not dependent on development process model, is an eliciting of the initial customer's requirements of the system and to its software components. On the basis of initial requirements the architecture, means of development, components and structure of system may be defined. Further detailed requirements for the system's components will be formulated. Requirements for components should be coordinated with requirements of the customer and should be specified in the process of their detailed elaboration. Procedures of the requirements capturing, specifying, managing, as a rule, are defined by the development process model. A product of this stage is the documents with the detailed description of the future system, of all requirements, specifications and other documents. There can be various versions of documents for managers, developers, customers and others stakeholders of the system under development. Each of document's versions should reflect the information the most important for each group participating in the project. The information allowing to make correct decisions on the project for managers, for developers - necessary for development, for customers - giving real representation on the future system.

The described stage in system's development lifecycle is called the Requirements Engineering. The most general goal of the Requirements Engineering process is to identify, analyze, verify and document requirements for the system to be developed. The pledge of the project's successful implementation in many respects depends on the successful performing of this initial development stage. It allows developers and customers to come to the agreement on what results should be achieved, and also avoid mistakes in consequence. This statement is based on the assumptions, that "the earlier the mistake will be found, the more cheaply it may be corrected", and that "it is possible to construct stable model of

the future system (on the basis of requirements) prior to the beginning of designing and development process”.

The goal of this work is to investigate some of the existed approaches for refinement, formalization and specification of natural-language (NL) requirements and to propose a new approach based on the ForTheL language [3][4].

The formalization of NL requirements is performed in order to reduce their ambiguity, inconsistency and incorrectness. Many semiformal and formal languages have been developed in an attempt to provide such formalization. A common drawback to these languages is that they are difficult for non-experts to understand, which limits their practical application. In order to improve the level of comprehension for non-expert the ForTheL (Formal Theory Language, pronounced “FORTEL”) was developed. The main peculiarities of ForTheL are that it is a formal language and designed to be close to natural English language.

But, initially ForTheL language was designed for writing mathematical texts. That’s why another important goal of this work is to investigate the possibilities of the language for capturing the requirements and processing them using the System for Automated Deduction (SAD). SAD is a processor of texts written in ForTheL (<http://ea.unicyb.kiev.ua/sad.en.html>) and other formal languages.

The SAD system may be considered as the modern vision of the Evidence Algorithm program advanced by Academician V. Glushkov. He proposed to simultaneously investigate the use of formalized languages for presenting formal texts in the form most appropriate for a user, the formalization and evolutionary development of computer made proof steps, the influence of the information environment on the evidence of a proof step, and man assisted search for a proof. The system SAD can be used to solve large variety of theorem proving problems including: establishing the deducibility of sequents in first-order classical logic, theorem proving in ForTheL-environment, verifying correctness of self-contained ForTheL-texts [3][4].

It also may be applicable for tasks concerned with verification of formal requirements specifications. The examples in this work will show such capabilities of the system.

DRAFT

Goals and Tasks

The goal of this study is to investigate the existed approaches for refinement and formalization of natural-language (NL) requirements specifications.

As a result of the investigation the approach based on the ForTheL language is presented. It will use the general approach and structure of Requirements Engineering stage in the System Engineering Process as a basis.

The software requirements that follow the restricted syntax of English language may be interactively translated into ForTheL statements or completely written using it. The interactive translation would be based on special templates.

After translation, requirements will be processed by the SAD system. The system will act as a formal checker in categorical analysis of requirements.

In Appendix some examples are presented to demonstrate the abilities of ForTheL language and the SAD system.

As a practical result the prototype of system for capturing and processing software requirements using abilities of SAD system will be developed. The system must operate in collaborative, distributed environment and guarantee high security level of the information.

The Requirements Engineering

Overview

Requirements engineering (RE), in software engineering, is a term used to describe all the tasks that go into the instigation, scoping and definition of a new or altered computer system. Requirements engineering is an important part of the software engineering process; whereby business analysts or software developers identify the needs or requirements of a client; having identified these requirements they are then in a position to design a solution [5].

Earlier it was considered that Requirements Engineering is relatively easy part of the process. However, soon, it became clear, that RE is a very important stage. As, many failures at system developing have been caused by mistakes admitted during the Requirements Engineering and it was too expensive or even impossible to correct them and to satisfy client's requirements in the given time.

At RE stage it is possible to construct a stable model of the future system on the basis of requirements prior to the beginning of designing and development process to prevent failures in the future. Complete requirements represent a declarative description of the future system. That's why Software Engineering (SE) researchers emphasized on the fact that: requirements describe what is to be done, but not how they are implemented [13].

In the following, the requirement engineering process will be examined and the main techniques developed for it will be discussed.

The Requirements engineering process

The whole process of requirements engineering is a web of sub processes, and it is very difficult to make a clear distinction between them [27].

At a different level of consideration the researchers divide the RE process into various stages. Nevertheless, it is accepted the RE process consists of five main activities:

- Requirements Elicitation
- Requirements Analysis
- Requirements Specification
- Requirements Validation
- Requirements Management

The Requirements Engineering process in general may be illustrated by the following scheme (Figure 2: RE process schema):

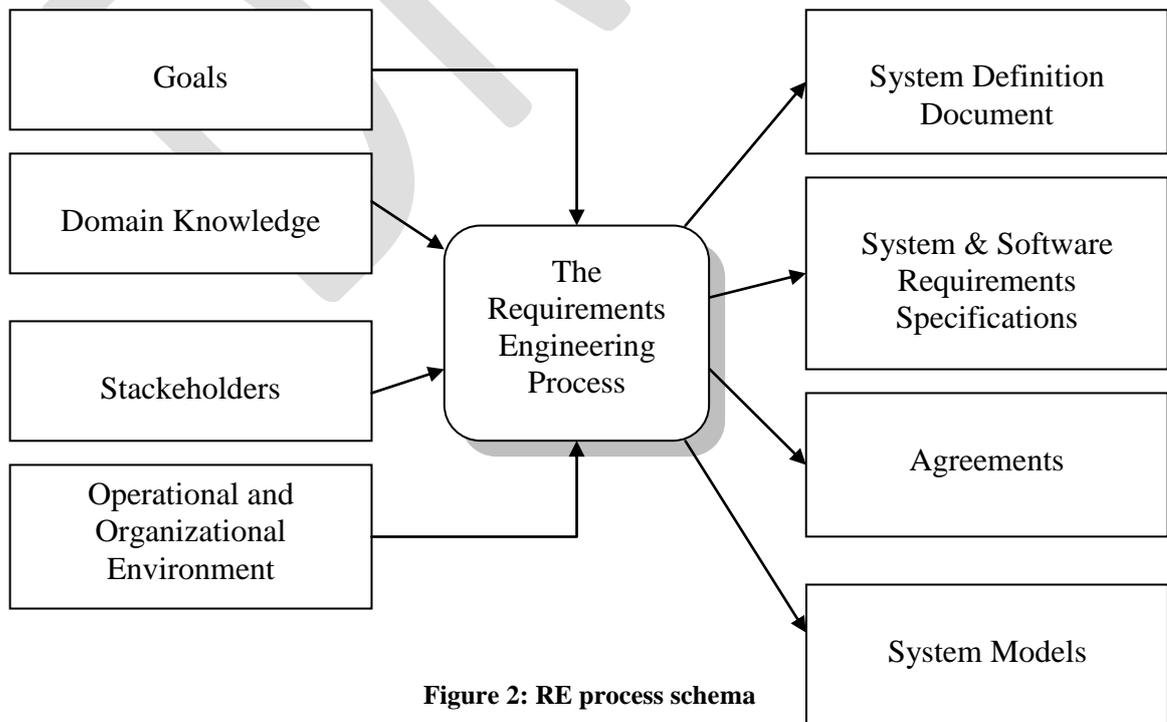


Figure 2: RE process schema

The scheme shows which information is involved into the process. In other words what are the inputs and outputs of it. The RE process is fundamentally interdisciplinary, and the requirements specialist needs to mediate between the domain of the stakeholder and that of software engineering [1]. Typical examples of software stakeholders include: users, customers, domain experts, etc. They represent a very important source of the system's requirements.

The experience of successful projects reveals that resources and technical risks can be reduced through rigorous and thorough requirements engineering.

Successfully completing a requirements engineering stage is a challenge. In the first place, it is not easy to identify all the stakeholders, give them all an appropriate form of input, and document all their input in a clear and concise format. And there are constraints. The requirements engineer is expected to determine whether or not the new system is: feasible, schedulable, affordable, legal, ethical [5].

The general difficulties involved with requirements analysis are increasingly well known:

- the right people with adequate experience, technical expertise, and language skills may not be available to lead the requirements engineering activities;
- the initial ideas about what is needed are often incomplete, wildly optimistic, and firmly entrenched in the minds of the people leading the acquisition process;
- the difficulty of using the complex tools and diverse methods associated with requirements gathering may negate the hoped for benefits of a complete and detailed approach.

The requirements engineering process usually have the iterative nature. Requirements typically iterate with a level and detail of the system's design. In some projects, all requirements can not be understood before the system design or

they can be clarified during the development process. Such kind of projects is usually of a high risk.

Now, the main activities of Requirements Engineering will be discussed in more details.

Requirements Elicitation

Requirements elicitation is the first stage in considering the problem the software system should be able to solve. This process is following after system engineering process which defines the context and goal of the software. The task of requirements elicitation is the establishing of boundaries and requirements for the software system. It is carried out by interaction with stakeholders and detailed studying of corresponding knowledge domains. Requirements elicitation is always be a human activity. There exist various techniques and methods of requirements elicitation. The most widespread techniques are interviews, scenarios, prototypes, facilitated meetings, observation, etc. The detailed description of which may be found in 0.

At this stage the relationships between developers of the system and the customer should be established.

The requirements elicitation is variously termed “requirements capture”, “requirements discovery” and “requirements acquisition”.

Gathered during these stage requirements may be checked for quality using different methods and tools ([7], [25], [26]). The quality of requirements is important feature for the following stages. Several rules for writing quality requirements are discussed in [28] and some examples are given.

Requirements Analysis

After requirements were discovered they must be analysed to:

- Necessity (need for the requirement);
- Detect and resolve drawbacks in them (for example, consistency, conflicts, ambiguity situations, completeness, etc);
- Improve their quality;
- They must be structured and refined;
- Discover the bounds and properties of the system;
- Discover how system will interact with the environment;
- Another necessary analysis.

After this stage the requirements have to be described clear enough to enable their specification and validation.

For providing more convenient analysis procedures, requirements may be structured by different characteristics. For example, whether the requirement is functional or non-functional [1]:

Functional requirements describe the functions that the software is to execute (formatting some text, modulating a signal). They are sometimes known as capabilities.

Non-functional requirements are the ones that act to constraint the solution. Non-functional requirements are sometimes known as constraints or quality requirements. They can be further classified according to whether they are performance requirements, maintainability requirements, safety requirements, reliability requirements, or one of many other types of software requirements.

One of the most common procedures in requirements analysis is negotiation. It may be used to resolve problems with requirements where conflicts occur between two stakeholders requiring mutually incompatible features (conflicting

requirements). Requirements that seem problematic are discussed and the stakeholders involved present their views about the requirements. After negotiation the requirements are prioritised and a compromise set of requirements may be agreed upon. Generally, this will involve making changes to some of the requirements, what also may cause new problems appearing.

Other techniques used for requirements analysis are: requirements classification, conceptual modelling, requirements negotiation, prioritization, architectural design and requirements allocation. They are widely presented in the corresponding literature.

Requirements Specification process

First of all, let's start with definition of the Software Requirements Specification process. There may be different definitions more or less complete. The definitions in the work are commonly based on an excellent source for definitions in Software Engineering discipline - the IEEE Computer Society (<http://www.ieee.org>, [1][2]). There, the Software Requirements Specification process is defined as “a process results of which are unambiguous and complete specification documents”.

It is accepted to consider the Software Requirements Specifications (SRS) is a complete description of the behavior of the system to be developed. It includes a set of use cases that describe all of the interactions that users will have with the software, numerical values, limits and measurable attributes which may be checked on the working system.

The SRS is a document (paper or electronic), which defines (specifies) the Software System.

The complete description of the functions to be performed by the software specified in the SRS will assist the potential users to determine if the software specified meets their needs or how the software must be modified to meet their needs [6].

From the IEEE standard [2]:

The basic issues that the SRS writer(s) shall address are the following:

- **Functionality.** What is the software supposed to do?
- **External interfaces.** How does the software interact with people, the system's hardware, other 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 constraints** imposed on an implementation. Are there any required standards in effect, implementation language, policies for database integrity, resource limits, operating environment(s) etc.?

An SRS should be:

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

An SRS is **correct** if, and only if, every requirement stated therein is one that the software shall meet.

There is no tool or procedure that ensures correctness. The SRS should be compared with any applicable superior specification, such as a system requirements specification, with other project documentation, and with other applicable standards, to ensure that it agrees. Alternatively the customer or user can determine if the SRS correctly reflects the actual needs. Traceability makes this procedure easier and less prone to error.

An SRS is **unambiguous** if, and only if, every requirement stated therein has only one interpretation. As a minimum, this requires that each characteristic of the final product be described using a single unique term. In cases where a term used in a particular context could have multiple meanings, the term should be included in a glossary where its meaning is made more specific.

An SRS is **complete** if, and only if, it includes the following elements:

a) All significant requirements, whether relating to functionality, performance, design constraints, attributes, or external interfaces. In particular any external requirements imposed by a system specification should be acknowledged and treated.

b) Definition of the responses of the software to all realizable classes of input data in all realizable classes of situations. Note that it is important to specify the responses to both valid and invalid input values.

c) Full labels and references to all figures, tables, and diagrams in the SRS and definition of all terms and units of measure.

Consistency refers to internal consistency. If an SRS does not agree with some higher-level document, such as a system requirements specification, then it is not correct.

An SRS is internally consistent if, and only if, no subset of individual requirements described in it conflict. The three types of likely conflicts in an SRS are as follows:

- a) The specified characteristics of real-world objects may conflict.
- b) There may be logical or temporal conflict between two specified actions.
- c) Two or more requirements may describe the same real-world object but use different terms for that object. The use of standard terminology and definitions promotes consistency.

An SRS is **ranked for importance and/or stability** if each requirement in it has an identifier to indicate either the importance or stability of that particular requirement. Typically, all of the requirements that relate to a software product are not equally important. Some requirements may be essential, especially for life-critical applications, while others may be desirable.

An SRS is **verifiable** if, and only if, every requirement stated therein is verifiable. A requirement is verifiable if, and only if, there exists some finite cost-effective process with which a person or machine can check that the software product meets the requirement. In general any ambiguous requirement is not verifiable.

An SRS is **modifiable** if, and only if, its structure and style are such that any changes to the requirements can be made easily, completely, and consistently while retaining the structure and style. Modifiability generally requires an SRS to:

- a) Have a coherent and easy-to-use organization with a table of contents, an index, and explicit cross referencing;
- b) Not be redundant (i.e., the same requirement should not appear in more than one place in the SRS);

c) Express each requirement separately, rather than intermixed with other requirements.

An SRS is **traceable** if the origin of each of its requirements is clear and if it facilitates the referencing of each requirement in future development or enhancement documentation.

To achieve the above mentioned features a number of techniques for specifying software requirements are developed. These techniques may be divided into two separate major classes – Natural Language (NL) based and based on special languages. Further more special language approaches also may be divided into several types. For example, universal approaches (Unified Modeling Language) and domain dependent (Z Notation, B-Method). But a common drawback of these languages, however, is that they are difficult for non-experts to understand. This fact limits their practical application. And the Natural Language continues to be the most appropriate way to express software requirements. However, Specifications expressed in the Natural Language are difficult for automated processing. But there exist a tools and linguistic methods that may be applicable for such tasks [7]. Unfortunately these methods are not as effective as it would be desirable.

The current state of Software Requirements Specification process was described in the article [8] by Donald Firesmith.

Later in this work the ForTheL language based approach will be presented. ForTheL is a formal natural like language. The ForTheL text may be translated into corresponding first-order logic formulas. The goal of the developing new approach is to remove the drawbacks of the formal specification languages with special syntax and to allow non-experts to take the advantages of formal methods in Requirements Specification process.

Requirements Validation

Proceeding from the various reasons in practice concepts of verification and validation are frequently confused. It must be emphasized that the verification and validation (V&V) may be performed at different levels (stages). That is the verification and validation procedure may be performed for the whole System or only for the System Requirements or for the Software System Requirements. There are also a difference between System Requirements and Software System Requirements (see for example, [1],[9]).

For clearing distinction between concepts of verification and validation let's begin from the dictionary definitions¹:

valid

- (of a reason, objection, etc.) sound or defensible; well-grounded.
- executed with the proper formalities (a valid contract).
- legally acceptable (a valid passport).
- not having reached its expiry date.

validate

- make valid; ratify, confirm.
- VV validation

¹"The Concise Oxford Dictionary," Microsoft® Encarta® 97 Encyclopedia. The Concise® Oxford Dictionary, 8th Edition. (c) © Oxford University Press. All rights reserved.

verification

- the process or an instance of establishing the truth or validity of something.
- **Philos.** the establishment of the validity of a proposition empirically.
- the process of verifying procedures laid down in weapons agreements.

sound

- healthy; not diseased or injured.
- (of an opinion or policy etc.) correct, orthodox, well-founded, judicious

correct

- true, right, accurate;
- (of conduct, manners, etc.) proper, right;
- in accordance with good standards of taste etc,

truth

- the quality or a state of being true or truthful (doubted the truth of the statement; there may be some truth in it).
- what is true (tell us the whole truth; the truth is that I forgot).
- what is accepted as true (one of the fundamental truths).

That is the Validation may be considered as truth and Verification as correctness. The truth is thought to be something absolute which cannot be interpreted differently.

Correctness is relative concept. For example, relative to the system under development and system's environment.

The verification of the system ensures that it is designed to deliver all functionality to the customer; it typically involves reviews and meetings to

evaluate documents, plans, code, requirements and specifications; this can be done with checklists, issues lists, and walkthroughs and inspection meetings.

System Validation ensures that functionality, as defined in requirements, is the intended behaviour of the System; validation typically involves actual testing and takes place after verifications are completed.

The more complete and rigorous definitions of System and Requirements Verification and Validation are presented by Terry Bahill and Steven Henderson in their article [9]:

Verifying requirements: Proving that each requirement has been satisfied. Verification can be done by logical argument, inspection, modelling, simulation, analysis, expert review, test or demonstration.

Validating requirements: Ensuring that (1) the set of requirements is correct, complete, and consistent, (2) a model can be created that satisfies the requirements, and (3) a real-world solution can be built and tested to prove that it satisfies the requirements. If Systems Engineering discovers that the customer has requested a perpetual-motion machine, the project should be stopped.

Verifying a system: Building the system right: ensuring that the system complies with the system requirements and conforms to its design.

Validating a system: Building the right system: making sure that the system does what it is supposed to do in its intended environment. Validation determines the correctness and completeness of the end product, and ensures that the system will satisfy the actual needs of the stakeholders.

Again, from their article:

Each requirement must be verified by logical argument, inspection, modeling, simulation, analysis, expert review, test, or demonstration. Here are some brief dictionary definitions for these terms:

- Logical argument: a series of logical deductions;
- Inspection: to examine carefully and critically, especially for flaws;
- Modeling: a simplified representation of some aspect of a system;
- Simulation: execution of a model, usually with a computer program;
- Analysis: a series of logical deductions using mathematics and models;
- Expert review: an examination of the requirements by a panel of experts;
- Test: applying inputs and measuring outputs under controlled conditions (e.g., a laboratory environment);
- Demonstration: to show by experiment or practical application (e. g. a flight or road test). Some sources say demonstration is less quantitative than test.

Modelling can be an independent verification technique, but often modelling results are used to support other techniques.

The requirements validation may be performed to ensure that the software engineer has correctly understood the customer's requirements and needs. It is also important to validate requirements for conformance with company standards and that they are understandable, consistent, and complete. Formal notations offer the important advantage of permitting the last two properties to be proven [1].

It is a good practice to perform requirements verifications and validation several times during system development process. In the iterative development processes this is necessary to perform requirements verifications and validation on every iteration.

Requirements Management

Requirements management is a relatively new branch in Requirements Engineering process. It is the activity concerned with the effective control of information related to system requirements. Requirements management process is carrying out together with other engineering processes.

The beginning of this process should be planned for the same time as process of initial requirements elicitation starts. Directly requirements management process should begin right after the draft version of the requirements specification will be ready.

Nevertheless Requirements Management process (RM) is widely supplied by software [10][11]. The software tools may perform the following RM tasks: store structured software and system requirements in database, manage versions and changes, store additional information related to requirements (attributes, context), communicate with customers, track status, prepare reports, manage user roles (software engineer, developer, customer) and generate information for them.

During the RM process the standards of organization's documents, document's work-flow and archiving must be established. All software engineers and developers must follow these standards. This will reduce ambiguity in future. Using the software tools these tasks are performed more effectively.

Further in the work the existed tools and software for Requirements Management will be discussed in more detail in connection with ForTheL approach for processing requirements. The special tools for this approach will be presented. The development of ForTheL approach will be performed according to the generally accepted standards in Software Engineering discipline.

The topic of the work also concerns with techniques for NL requirements analysis. They will be discussed in the next chapter.

Tools for Natural Language Requirements Analysis

The goal of this section is to give a short overview of the linguistic methods for analysing natural language requirements.

Template-based approach

Semi-formal descriptions represent one of the approaches of reducing inconsistency and ambiguity of natural language documents. In the case of requirement specifications, text documents are structured using text templates. The templates and writing standards must be selected according to the needs of each individual project. It is obvious that one standard can't fit all needs. So, there are templates available for different domains. For example, the Volere requirements specification template (<http://www.volere.co.uk>) is applicable for software projects. In this template it was proposed to use a requirement shell for writing each atomic requirement as given at the picture below:

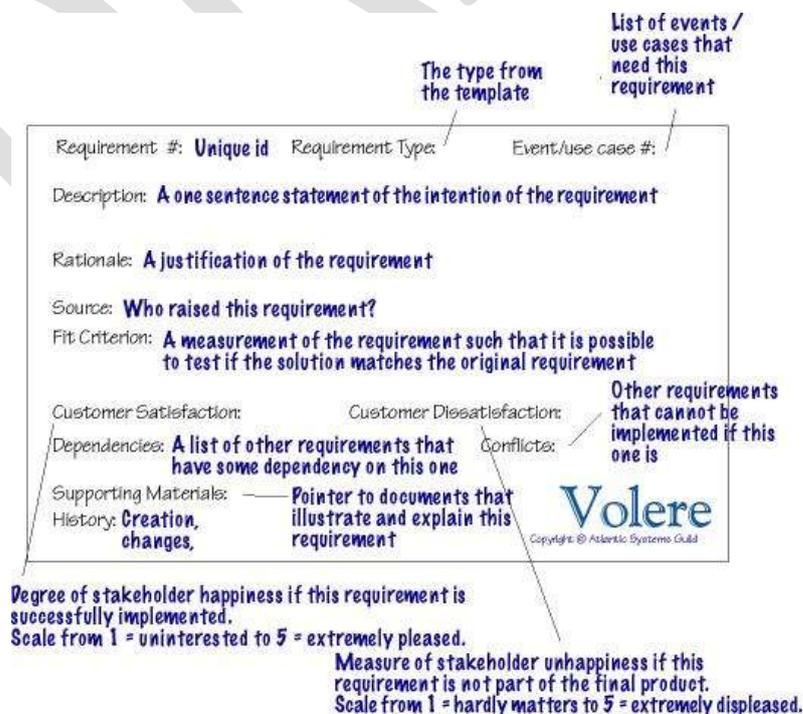


Figure 3: Volere Template

Furthermore, A. Cockburn in his book [13] emphasised that “it is incorrect to publish just one use case template. There must be at least two, a casual one for low-ceremony projects, and a fully dressed one for higher-ceremony projects.”

If object-oriented development methods are applied, requirements are described by use cases. For expressing behavioural information of use cases, text templates are widely accepted. These templates provide typically a semi-formal description called a scenario. The keywords give structure to a natural language expression (see Table: Use Case template). As a result, on average a description will be more complete and less ambiguous than without the structure. The structure mainly acts as guideline for the stakeholder writing down the requirements and specifications, reducing omissions, incompleteness and ambiguously stated expressions.

Table: Use Case template

USE CASE	Name of this Use Case
Service	Benefit of this Use Case for the user
Brief description	Optional description
Actors	List of actors related to this use case
Scope	Extent of the system of concern
Level	Level of Abstraction
Precondition	Conditions for the state of the actors and the system, required for the application of this Use Case
Main success scenario	
Post Condition	(List of) conditions fulfilled after a successful execution
Alternative scenarios	
Failure	Short description of a possible exceptional condition

Each case is mainly a description of a possible scenario and can have pre- and post-conditions. The information is most often expressed in natural language, albeit supplemented with e.g. diagrams or referenced documents. Using a semi-

formal template, the comprehension for users will be improved compared to plain text. However, the information cannot be evaluated by automatic tools – e.g. for automatic verification or for test case generation purposes. The aim is to transform the semi-formal statements into an expression with a formally defined syntax and semantics. The latter can be a diagram or a mathematical type of statements, but having unique semantics.

Ambiguities in structured texts come from expressions without a reference to the particular context or form and the use of words that have more than one meaning within the particular context. Such expressions have to be replaced by ones that define and remove all ambiguities in the given context. In this case, predefined keywords and glossary items provide possible solutions to build up such unambiguous expressions. Further, incompleteness in structured texts is frequently caused by missing objects, subjects, transitions or relations within the expressions. Missing terms result as well in missing refinements. Hence, to remove ambiguity and incompleteness, the use of a set of templates for structuring the requirements and specifications expressions is proposed. For each natural-language part of a semi-formal description a template is applied to formalize it. Every template is defined by a relation to a semantic pattern and a linguistic structure. Based on this relation, a linguistic analysis can be applied to derive suggestions for the application of a template based on a natural language expression. For the evaluation of such expressions the usual procedure of textual analyses is performed:

In a lexical analysis the words of the natural language text are compared to a lexicon, known words are identified and their type is marked. Unknown words have to be classified interactively.

In a syntax analysis a syntax tree is built for every sentence. A linguistic analysis is performed to identify linguistic structures within the syntax tree.

The semantic analysis compares the linguistic structure to semantic patterns. Suggestions for suitable templates are derived.

The templates follow a strict syntax. For each field within a template, a specified type of term can be placed (Tab 2). By defining the terms via a reference to a glossary of predefined terms, the semantics of such a description are formalized to some extent. In this way, the descriptions are transformed from a free vocabulary to a restricted one. As a result, the descriptions can be evaluated by tools.

Scheme of one of the templates with types and an example:

Actor	Activity	Object	Destination
The librarian	selects	the book	in the list of available books.

The templates are based on linguistic structures and on semantic patterns as defined by Chomsky. The template of Tab 2 is based on the communication pattern:

Communication (V) [Agent; Object; Source; Destination]

That has been instantiated in the way:

Communication (select) [Agent: ‘the librarian’; Object: the book’; Source: ‘the librarian’; Destination: ‘in the list of available books’]

As a result of this transformation, the degree of formal definition of syntax and semantics is increased. The step between the syntax analysis and the semantic analysis is relatively large. For common natural language texts there is a wide variety of linguistic structures, demanding for a big maintenance effort. Semi-formal descriptions offer advantages because the number of linguistic structures can be limited.

The above approach was described in the article Refinement and Formalization of Semi-Formal Use Case Description by Matthias Riebisch, Michael Hübner [19].

However, it is important to understand that every procedure of textual analyses can't give hundred percent correct result because of natural language ambiguities. But, there are tools for processing NL requirements that are using textual analyses more or less successfully (for example QuARS [7][14]).

Another approach was described by Kendra Cooper and Mabo Ito in their article: "Formalizing a Structured Natural Language Requirements Specification Notation" [17]. In the abstract they wrote:

«Requirements specification notations are developed by organizations in order to meet their specific needs. For example, the Threads-Capabilities notation, an in house notation at Raytheon Systems Canada, Ltd., has been developed and used for specifying their complex, large scale, air traffic control systems. It is a semi-formal, structured, natural language notation. In this work, we investigate how to make this semi-formal notation more rigorous (i.e., formal) by developing and applying a new formalization process to it. By doing this, we can obtain the advantages of formal methods (precise, unambiguous, automatic generation of test specifications, automated type checking, etc.) while retaining the style and readability of the original notation. We call the formalized notation the Stimulus Response Requirements Specification (SRRS) notation. Our results have been successful for the specific notation. The formalized notation has been demonstrated to reduce the time and improve the quality of the requirements specifications. There is additional training time, however, needed to learn to use the notation and tools».

It was shown that template-based approach for NL requirements strongly depends on procedures of textual analyses. Therefore the effectiveness of this approach is directly defined by effectiveness of textual analyses. For NL

requirements textual analyses may be performed more precisely in comparison with the usual text as they use less rich lexicon.

The Model Driven Architecture methodology

The Model Driven Architecture, known as MDA, is a framework for software development defined by the OMG. Key to MDA is the importance of models and transformations between them in the software development process. MDA defines how models defined in one language can be transformed into models in other languages. An MDA development process generally begins with a Computer Independent Model (CIM) which describes the business system independently of the software system to be implemented.

In this case there exists an open problem: Is it possible to develop the transformation process to define CIM from natural oriented models. This problem was studied in the article "Integrating Natural Language Oriented Requirements Models into MDA" by M.C. Leonardi and M.V. Mauco [18]. They present such a transformation for Language Extended Lexicon and the Scenario Model. A short description of these models from the article:

"Language Extended Lexicon (LEL): It is a structure that allows the representation of significant symbols of the Universe of Discourse. It is composed by a set of symbols which have a name (and a set of synonyms), notions, and behavioural responses. LEL symbols define objects, subjects, verbal phrase and states. When describing LEL symbols two rules must be followed simultaneously: the "circularity principle" and the "minimum vocabulary principle".

Scenario Model: A scenario describes situations in the Universe of Discourse. A scenario is connected to the LEL and it is composed by: a title to identify it, a goal describing its purpose, a context to define geographical and temporal locations and preconditions, actors which are entities actively involved in the scenario generally persons or organizations, a set of resources that identify

passive entities with which actors work, and a set of episodes where each episode represents an action performed by actors using resources. An episode may be explained as a scenario; this enables a scenario to be split into sub-scenarios.”

Procedures of Linguistic Analysis

The purpose of this section is to describe basic procedures of the linguistic analysis and problems which arise with them.

While automating processing of natural language requirements, except of trivial cases, the procedures of the linguistic analysis have to be involved. Unfortunately such procedures sufficiently are not presented as effective software tools.

Therefore at the given stage it is expedient to speak, about some applications of the linguistic analysis procedures to the automated processing of natural language requirements.

It is necessary to take into account, that the Software Requirements, as a rule, do not possess a wide lexicon and complex syntactic structures of their sentences, which is usual for literary texts. Besides it is possible to impose restrictions on language for writing requirements. That sometimes enables to use the methods which are not working effectively in general case. The practical application of linguistic methods is discussed further. Let's consider typical tasks which arise while applying linguistic procedures (Figure 4: Linguistic Analysis).

The figure shows that linguistic analysis is a complex process. On each stage of which the procedures for resolving different ambiguities has be applied (anaphora resolution, proper names extractions, ambiguities while parsing, etc). The result of the analysis strongly depends on the quality of such resolving procedures.

Example, let's consider the following requirement:

The system must react within 10 seconds.

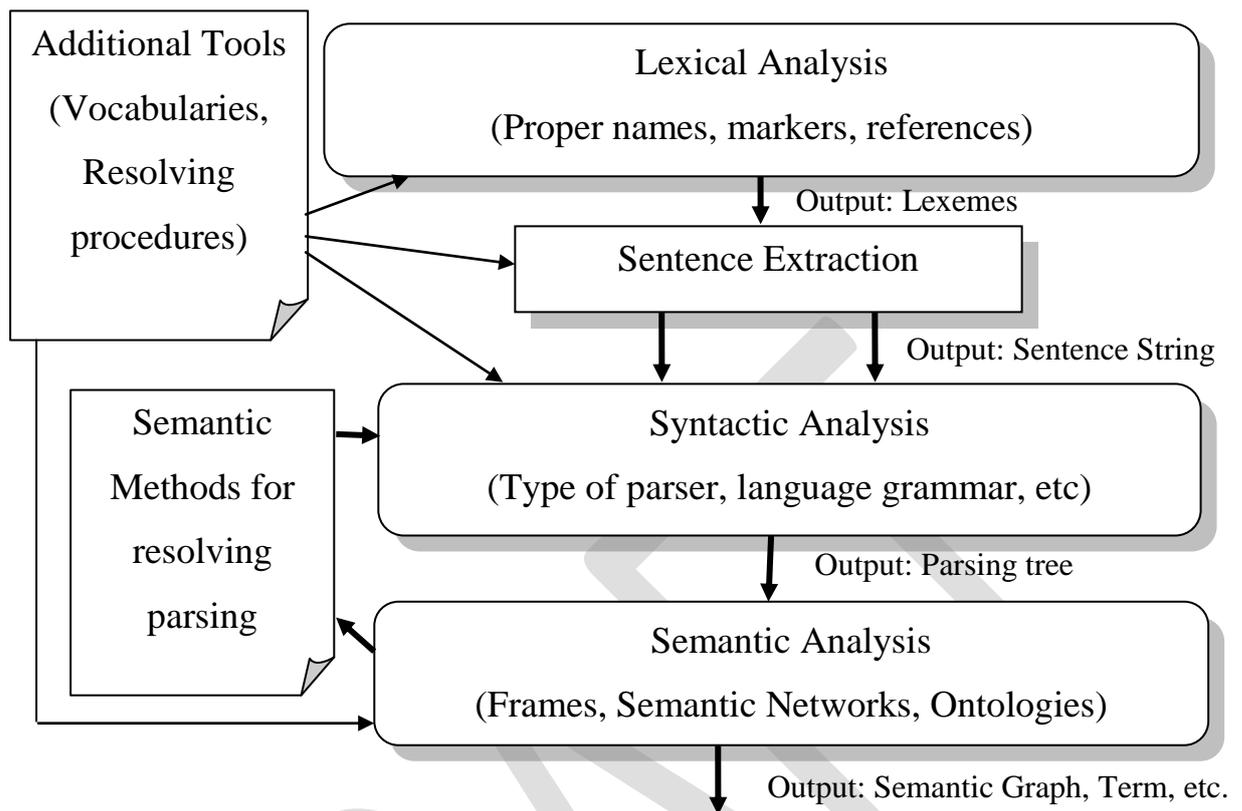


Figure 4: Linguistic Analysis

The first stage is lexical analysis. It is performed together with morphological analysis and other initial processes. The result of this stage is a list of lexemes (words). This list will be used during syntax analysis. Usually lexical analysis did not perform as individual procedure.

A parsing of the English sentences may be performed with Link Grammar Parser [23] or any other available parser. The resulting parsing tree of the above mentioned requirement is shown on the (Figure 5: Parsing Tree). On the stages of lexical and syntax analysis many drawbacks in requirements may be detected and then corrected by requirements analyst.

After successful syntax parsing procedure, the semantical analysis of the sentence will be performed in order to build semantic structure for further analysis. The semantic structure must be chosen by the developer of the system according to tasks which it must perform. The resulting semantic structure may be as follow (Figure 6: Semantical Structure).

This structure then will be analysed to identify undefined entities, contradictions, incompleteness, etc.

Further more, the requirements documents may be checked for readability, because it is more preferable for the customer to have an easy to read documents.

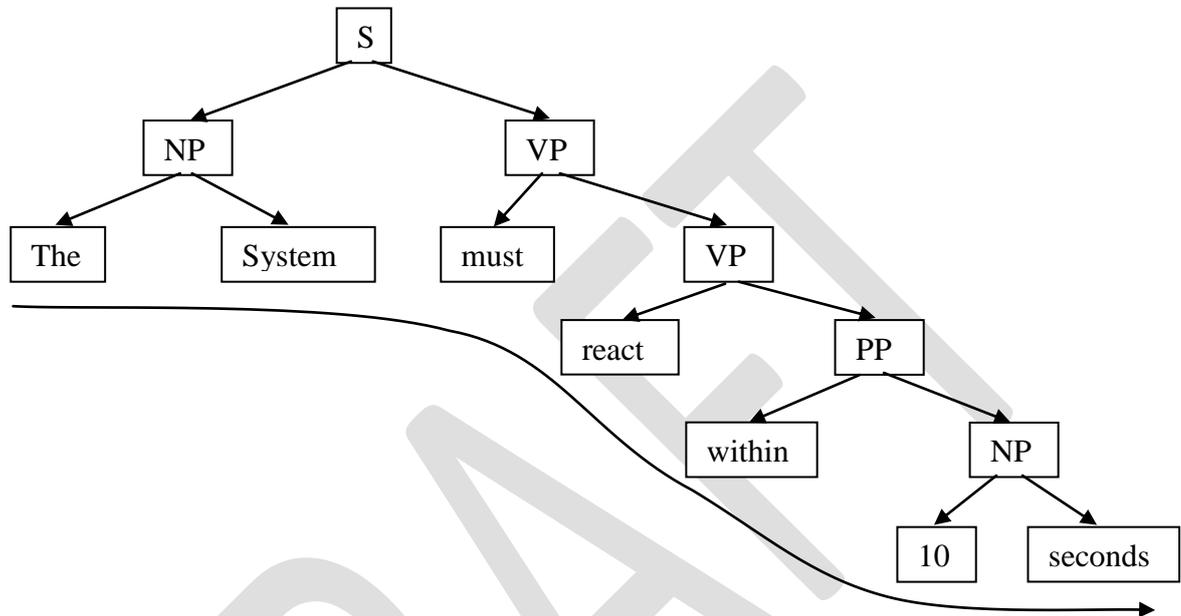


Figure 5: Parsing Tree

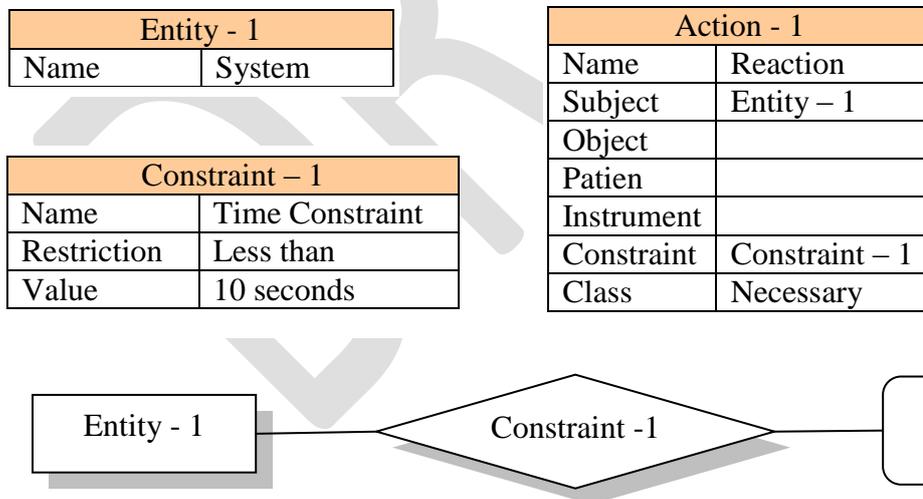


Figure 6: Semantical Structure

The methods of measuring the readability of a text are usually based on multiple correlation analysis. The subjects of analysis may be words per sentence, average number of syllables per word etc. Unfortunately there is no universal metric applicable for any language. The readability index formulas (metrics) only

work for a specific language. For example, such formulas are applicable for English:

Coleman-Liau Formula:

$$\text{Ease} = [5.89 * \text{char/words} - 0.3 * \text{sent}/(100 * \text{words}) - 15.8]$$

The Kincaid formula:

$$\text{Ease} = 11.8 * \text{syllables/words} + 0.39 * \text{words/sent} - 15.59$$

The Automated Readability Index formula:

$$\text{Ease} = 4.71 * \text{char/words} + 0.5 * \text{words/sent} - 21.43$$

The Flesch Reading formula [24]:

$$\text{Flesch Index} = 206.835 - 84.6 * \text{syllables/words} - 1.015 * \text{words/sent}$$

Where:

syllables - number of syllables;

words - the number of words;

sent - the number of sentences;

char - the number of characters in the text.

The Flesch Index can be interpreted by the following table:

Index	Difficulty
90-100	Very Easy
80-90	Easy
70-80	Fairly Easy
60-70	Normal
50-60	Fairly Difficult
30-50	Difficult
0-30	Very Difficult

The Quality Model for Software Requirements is defined in [7]. The quality of requirements is evaluated through so-called linguistic indicators that are directly detectable and measurable. The most valuable indicators are presented in following tables (the description is taken from [7]):

Table: Ambiguity Indicators

Indicator	Description
Vagueness	When parts of the sentence are inherently vague (e.g., contain words with non-unique quantifiable meanings).
Subjectivity	When sentences contain words used to express personal opinions or feelings.
Optionality	When the sentence contains an optional part (i.e., a part that can or cannot be considered).
Implicity	When the subject or object of a sentence is generically expressed.
Weakness	When a sentence contains a weak verb.

Table: Specification Completion Indicators

Indicator	Description
Under-specification	When a sentence contains a word identifying a class of objects without a modifier specifying an instance of this class.

Table: Understandability Indicators

Indicator	Description
-----------	-------------

ator	
Multi plicity	When a sentence has more than one main verb or more than one subject.
Read ability	<p>The readability of sentences is measured by the Coleman-Liau Formula of readability.</p> <p>The reference value of this formula for an easy-to-read technical document is 10. If the value is greater than 15, the document is difficult to read.</p>

Table: Expressiveness Defect Indicators

Indicator	Description
Vagueness	The use of vague words (e.g., easy, strong, good, bad, useful, significant, adequate, recent).
Subjectivity	The use of subjective words (e.g., similar, similarly, having in mind, take into account, as [adjective] as possible).
Optionality	The use of words that convey an option (e.g., possibly, eventually, in case of, if possible, if appropriate, if needed).
Implicitly	<p>The use of sentence subjects or complements expressed by demonstrative adjectives (e.g., this, these, that, those) or pronouns (e.g., it, they).</p> <p>The use of terms that have the determiner expressed by a demonstrative adjective (e.g., this, these, that, those), implicit adjective (e.g., previous, next, following, last), or preposition (e.g., above, below).</p>
Weakness	The use of weak verbs (i.e., could, might, may).

Under-specification	The use of words that need to be instantiated (i.e., flow [data flow, control flow], access [write access, remote access, authorized access], testing [functional testing, structural testing, unit testing]).
Multiplicity	The use of multiple subjects, objects, or verbs, which suggests there are actually multiple requirements.

To detect these indicators in requirements the authors of the tool (QuARS [7]) used only lexical and syntactical analysis. The results of analysis strongly depend on special dictionaries, what makes this approach domain dependent.

As it can be seen from this chapter, the linguistic tools and methods may detect and help to correct a number of drawbacks in requirements and specifications written in natural language. But such analysis is usually domain dependent and current tools have to be adapted for requirement's language of any individual project. Such adaptation is performed using special subject dictionaries, restrictions on language and lexicon, language grammars, etc.

Linguistic analysis can't be considered as a strictly formal analysis technique because of the natural ambiguities existed in the language. That is why it can guarantee the correctness of analysis with some degree of probability which is not acceptable for critical projects.

In the next section the use of Formal Methods for Requirements Engineering will be discussed.

Formal Methods in Requirements Engineering

The information in the section will be presented according to Wikipedia, the free encyclopedia article about formal methods:

http://en.wikipedia.org/wiki/Formal_Methods

Definition

In computer science Formal methods refer to mathematically based techniques for the specification, development and verification of software and hardware systems. The approach is especially important in high-integrity systems, for example where safety or security is important, to help ensure that errors are not introduced into the development process. Formal methods are particularly effective early in development at the requirements and specification levels, but can be used for a completely formal development of an implementation (e.g., a program).

Classification

Formal methods are intended to systematize and introduce rigor into all the phases of software development. This helps the system developers to avoid usual mistakes, provides a standard means to record various assumptions and decisions, and forms a basis for consistency among many related activities. By providing precise and unambiguous description mechanisms, formal methods facilitate the software development of the critical systems.

There are many specific formalisms and notations used by different formal development approaches. The majority of them may be divided into following categories:

- algebraic specification - used for specification and verification;
- predicate logic used for specification and verification;
- state charts - used for specification of reactive systems;
- graphical diagrams (UML) - used primarily for design, and also for requirements analysis.

As it was mentioned above Formal methods can be used at different stages of system's development. For example:

- Formal specification may be prepared and then a program developed from this informally. Then according to the formal specification it may be verified and validated more precisely. This choice may be the most cost-effective option in many cases.
- Formal development and verification technique may be used to produce a program in a more formal manner. Proofs of properties or refinement from the specification to a program may be performed. This may be most appropriate in high-integrity systems involving safety or security.
- Theorem provers may be used to undertake fully formal machine-checked proofs. This can be very expensive and is only practically worthwhile if the cost of mistakes is extremely high (in critical parts of microprocessor design).

By the expressional style formal methods may be roughly classified as follows:

- Denotational semantics, in which the meaning of a system is expressed in the mathematical theory of domains. Proponents of such methods rely on the well-understood nature of domains to give meaning to the system; critics point out that not every system may be intuitively or naturally viewed as a function.
- Operational semantics, in which the meaning of a system is expressed as a sequence of actions of a (presumably) simpler computational model. Proponents of such methods point to the simplicity of their models as a means to expressive clarity; critics counter that the problem of semantics has just been delayed (who defines the semantics of the simpler model?).
- Axiomatic semantics, in which the meaning of the system is expressed in terms of preconditions and postconditions which are true before and after the system performs a task, respectively. Proponents note the connection to classical logic; critics note that such semantics never really describe what a system does (merely what is true before and afterwards).

Application

Formal methods can be applied at various points through the development process. (For convenience, terms common to the waterfall model were used, though any development process could be used.)

Specification

Formal methods may be used to give a description of the system to be developed, at whatever level(s) of detail desired. This formal description can be used to guide further development activities (see following sections); additionally,

it can be used to verify that the requirements for the system being developed have been completely and accurately specified.

The need for formal specification systems has been noted for years. In the ALGOL 60 Report, John Backus presented a formal notation for describing programming language syntax (later named Backus normal form or Backus-Naur form (BNF)); Backus also described the need for a notation for describing programming language semantics. The report promised that a new notation, as definitive as BNF, would appear in the near future; it never appeared.

Development

Once a formal specification has been developed, the specification may be used as a guide while the concrete system is developed (i.e. realized in software and/or hardware). Examples:

If the formal specification is in an operational semantics, the observed behavior of the concrete system can be compared with the behavior of the specification (which itself should be executable or simulateable). Additionally, the operational commands of the specification may be amenable to direct translation into executable code.

If the formal specification is in an axiomatic semantics, the preconditions and post conditions of the specification may become assertions in the executable code.

Verification

Once a formal specification has been developed, the specification may be used as the basis for proving properties of the specification (and hopefully by inference the developed system).

Human-Directed Proof

Sometimes, the motivation for proving the correctness of a system is not the obvious need for re-assurance of the correctness of the system, but a desire to understand the system better. Consequently, some proofs of correctness are produced in the style of mathematical proof: handwritten (or typeset) using natural language, using a level of informality common to such proofs. A "good" proof is one which is readable and understandable by other human readers.

Critics of such approaches point out that the ambiguity inherent in natural language allows errors to be undetected in such proofs; often, subtle errors can be present in the low-level details typically overlooked by such proofs. Additionally, the work involved in producing such a good proof requires a high level of mathematical sophistication and expertise.

Automated Proof

In contrast, there is increasing interest in producing proofs of correctness of such systems by automated means. Automated techniques fall into two general categories:

- Automated theorem proving, in which a system attempts to produce a formal proof from scratch, given a description of the system, a set of logical axioms, and a set of inference rules.
- Model checking, in which a system verifies certain properties by means of an exhaustive search of all possible states that a system could enter during its execution.

Neither of these techniques work without human assistance. Automated theorem provers usually require guidance as to which properties are "interesting"

enough to pursue; model checkers can quickly get bogged down in checking millions of uninteresting states if not given a sufficiently abstract model.

Proponents of such systems argue that the results have greater mathematical certainty than human-produced proofs, since all the tedious details have been algorithmically verified. The training required to use such systems is also less than that required producing good mathematical proofs by hand, making the techniques accessible to a wider variety of practitioners.

Critics note that such systems are like oracles: they make a pronouncement of truth, yet give no explanation of that truth. There is also the problem of "verifying the verifier"; if the program which aids in the verification is itself unproven, there may be reason to doubt the soundness of the produced results.

Examples of theorem proving applications ([32]):

Software generation is an economically important real world application of ATP. Although the use of ATP in software generation is in its infancy, there have already been some interesting results. The KIDS ([33]) system developed at Kestrel Institute has been used to derive scheduling algorithms that have outperformed currently used algorithms. KIDS provide intuitive, high level operations for transformational development of programs from specifications. The AMPHION ([34]) project, sponsored by NASA, is used to determine appropriate subroutines to be combined to produce programs for satellite guidance. By encapsulating usable functionality in software components (e.g. subroutines, object classes), and then reusing those components, AMPHION can develop software of greater functionality in less time than human programmers, with some assurance that the overall system is correct because it is built up from trusted components

Software verification is an obvious and attractive goal, which is slowly being realized using ATP technology. Three examples are given here, while many

more are indexed at the Formal Methods page ([35]). The Karlsruhe Interactive Verifier (KIV, [36]) was designed as an experimental platform for interactive program verification at the University of Karlsruhe, and has since been used to successfully verify a range of software applications. These include some case studies of academic software, e.g., implementation on set functions, tree and graph representation and manipulation, and verification of a Prolog to WAM compiler. KIV is at the threshold of industrial application, with pilot studies undertaken in various domains, including a software controlled railway switch, safe command transfer in a space vehicle, and supervision of neutron flow in a nuclear reactor. PVS ([37]) is a verification system that has been used in various applications, including diagnosis and scheduling algorithms for fault tolerant architectures, and requirements specification for portions of the space shuttle flight control system. NASA uses ATP ([38]) to certify safety properties of aerospace software that has been automatically generated from high-level specifications. Their code generator produces safety obligations that are provable only if the code is safe. An ATP system discharges these obligations, and the output proofs, which can be verified by an independent proof checker, serve as certificates.

Security protocol verification techniques analyze protocols that are used to transmit data over networks in a secure fashion, in an attempt to verify or find flaws in the protocol. The SPASS system has been used to analyze the Neuman-Stubblebine ([39]) protocol, various cryptographic protocols have been analysed by transformation to Horn clauses ([40]), the TAPS system does verification based on invariants, and the Coral ([41]) system has been used to find attacks on faulty security protocols.

Hardware verification is the largest industrial application of ATP. IBM, Intel, and Motorola are among the companies that employ ATP technology for verification. A few good examples of the use of ATP systems for hardware

verification are listed here, while many more are indexed at the Formal Methods page. The ACL2 ([42]) system has been used to obtain a proof of the correctness of the floating point divide code for AMD's PENTIUM-like AMD5K86 microprocessor, while ANALYTICA has been used to verify a division circuit that implements the floating point standard of IEEE. PVS (mentioned above in the context of software verification) has been used to verify a microprocessor for aircraft flight control. The RRL system has verified commercial size adder and multiplier circuits. The HOL ([43]) system has been used at Bell Laboratories for hardware verification. Nqthm ([44]) has been used to produce a mechanical proof of correctness of the FM9001 microprocessor. Safelogic ([45]) is a Swedish company that provides ATP based tools for verification of a system's logical functionality

Formal Methods and their notations

Abstract State Machines (ASMs)

Abstract State Machines ([46],[47]), formerly known as Evolving Algebras, are a formal method for specification and verification. The approach was originally developed by Yuri Gurevich, based around the concept of an abstract state machine, and is also espoused by Egon Börger. ASM theory is the basis for AsmL, the Abstract State Machine Language by Microsoft and XASM, an open source implementation. A number of support tools are available.

Alloy

The Alloy ([48]) specification language is a simple structural modelling tool based on first-order logic. Alloy is targeted at the creation of micro-models of software systems that can then be automatically checked for correctness.

Alloy specifications can be checked using the Alloy Analyzer. The Alloy Analyzer supports the analysis of partial models. As a result, it can perform incremental analysis of models as they are constructed, and provide immediate feedback to users.

B-Method

B ([49][50]) is a tool-supported formal method based around AMN (Abstract Machine Notation), used in the development of computer software. It was originally developed by Jean-Raymond Abrial in France and the UK. B is related to the Z notation (also originated by Abrial) and supports development of programming language code from specifications. B has been used in major safety-critical system applications in Europe (such as the Paris Metro Line 14), and is attracting increasing interest in industry. It has robust, commercially available tool support for specification, design, proving and code generation.

The method of software development based on B is known as the B-Method.

Compared to Z, B is slightly more low-level and more focused on refinement to code rather than just formal specification – hence it is easier to implement a specification written in B correctly than one in Z. In particular, there is good tool support for this.

Process calculi

In computer science, the process calculi (or process algebras) are a diverse family of related approaches to formally modelling concurrent systems. Process calculi provide a tool for the high-level description of interactions, communications, and synchronizations between a collection of independent agents or processes. They also provide algebraic laws that allow process descriptions to be manipulated and analyzed, and permit formal reasoning about equivalences between processes (using bisimulation). Leading examples of process calculi include CSP, CCS, and ACP. More recent additions to the family include the π -calculus, the ambient calculus, PEPA and the fusion calculus

Communication Sequential Processes

Communicating Sequential Processes is a formal language for describing patterns of interaction in concurrent systems. It is a member of the family of mathematical theories of concurrency known as process algebras, or process calculi. CSP was first described in a 1978 paper [51] by C. A. R. Hoare, but has since evolved substantially. CSP has been practically applied in industry as a tool for specifying and verifying the concurrent aspects of a variety of different systems. The theory of CSP is still the subject of active research, including work to increase its range of practical applicability (e.g. increasing the scale of the systems that can be tractably analyzed). CSP was influential in the development of the occam programming language.

π -calculus

In theoretical computer science, the π -calculus ([52]) is a notation originally developed by Robin Milner, Joachim Parrow and David Walker as an advance over

Calculus of Communicating Systems in order to provide mobility in modelling concurrency. The π -calculus is in the family of process calculi that have been used to model concurrent programming languages just as the λ -calculus has been used to model sequential programming languages

Actor model

The Actor model is a mathematical model of concurrent computation that has its origins in a 1973 paper by Carl Hewitt, Peter Bishop, and Richard Steiger. The Actor model treats “Actors” as the universal primitives of concurrent digital computation: in response to a message that it receives, an Actor can make local decisions, create more Actors, send more messages, and determine how to respond to the next message received. The Actor model has been used both as a framework within which to develop a theoretical understanding of concurrency, and as the theoretical basis for several practical implementations of concurrent systems.

Esterel

Esterel is a synchronous programming language for the development of complex reactive systems. The imperative programming style of Esterel allows the simple expression of parallelism. As a consequence, it is very well suited for control-dominated model designs.

The development of the language started in the early 1980s, and was mainly carried out by a team of Ecole des Mines de Paris and INRIA lead by Gérard Berry. Current compilers take Esterel programs and generate C code or hardware (RTL) implementations (VHDL or Verilog).

The language is still under development, with several compilers out. The commercial version of Esterel is the development environment Esterel Studio. The

company that develops it (Esterel Technologies) has initiated a normalization process with the IEEE.

Lustre

Lustre is a formally defined, declarative, and synchronous data-flow programming language, for programming reactive systems. It began as a research project in the early 1980s. In 1993, it progressed to practical, industrial use, in a commercial product, as the core language of the industrial environment SCADE, developed by Esterel Technologies. It is now used for critical control software in aircraft, helicopters, and nuclear power plants.

Petri nets

A Petri net (also known as a place/transition net or P/T net) is one of several mathematical representations of discrete distributed systems. As a modelling language, it graphically depicts the structure of a distributed system as a directed bipartite graph with annotations. As such, a Petri net has place nodes, transition nodes, and directed arcs connecting places with transitions. Petri nets were invented in 1962 by Carl Adam Petri in his PhD thesis.

RAISE

RAISE ([53]) was developed as part of the European ESPRIT II LaCoS project in the 1990s, led by Dines Bjørner. It consists of a set of tools based around a specification language (RSL) for software development. It is especially espoused by UNU-IIST in Macau, who runs training courses on site and around the world, especially in developing countries.

Vienna Development Method

Vienna Development Method ([54], [55]) is a software development method based on formal specification using the VDM specification language (VDM-SL), with tool support. There is an object-oriented extension, VDM++.

Z notation

The Z ([56]) notation (universally pronounced zed, named after Zermelo-Fränkel set theory) is a formal specification language used for describing and modelling computing systems. It is targeted at the clear specification of computer programs and the formulation of proofs about the intended program behaviour.

Z was developed by the Programming Research Group at Oxford University in the late 1970s and is based on the standard mathematical notation used in axiomatic set theory, lambda calculus, and first-order predicate logic. All expressions in Z notation are typed, thereby avoiding some of the paradoxes of naive set theory. Z contains a standardized catalogue (called the mathematical toolkit) of commonly used mathematical functions and predicates.

Although Z notation uses many non-ASCII symbols, the specification includes suggestions for rendering the Z notation symbols in ASCII and in LaTeX.

Conclusions

At the current state of the art, proofs of correctness, whether handwritten or computer-assisted, need significant time (and thus money) to produce, with limited utility other than assuring correctness. This makes formal methods more likely to be used in fields where the benefits of having such proofs, or the danger in having undetected errors, make them worth the resources. Example: in aerospace

engineering, undetected errors may cause death, so formal methods are more popular than in other application areas.

At times, proponents of formal methods have claimed that their techniques would be the “silver bullet” to the software crisis.

In the next section, the formal natural like language will be discussed. The purpose of this language is to make formal methods usable and more comprehensible for non-experts. The examples of its application are shown.

DRAFT

Formal Theory Language (ForTheL)

A brief introduction into ForTheL language

Traditionally it is accepted to use the language of first or higher order logic for writing formal mathematical texts. The usual weakness of this approach is the difficulty for understanding such texts what is caused by linguistic poverty of logic's language. A possible solution of such problem is to construct the languages that simulate natural languages and which allow to translate their sentences into formulas of the first order logic. ForTheL (Formal Theory Language) is a language of such kind. There are two reasons to pursue a verbose "natural" style instead of basing on a terse unifying notation of some traditional language of logic.

First, a text composed with correct English sentences will hopefully be more readable than a collection of formulas built with quantifiers, parentheses, lambdas and so on. It is also more pleasant to write. So, the first reason is to provide a framework with a user-friendly interface.

Second, in a natural language text there is a lot of information that lies beyond logic as such and that usually loosed in translation. In a natural speech, we meet nouns, which denote classes of entities; adjectives and verbs, which act as attributes and restrict classes; adjectives and verbs, which act as predicates and may relate different entities. In a traditional mathematical text, we meet definitions and axioms, important theorems and auxiliary lemmas, we meet various reasoning schemes. Where human language makes distinctions, the language of mathematical logic unifies.

The text written in ForTheL, consist of axioms, definitions, lemmas and theorems. The grammar of the language is constructed so that it simulates the grammar of natural (English) language. Besides, it may be dynamically extended in order to cover various subject domains. Adaptation of language grammar to the

concrete mathematical branch or any other branch that admits formalization is carried out by means of special text constructions - introductors. With their help new syntactic primitives may be added to language. They serve for building more complex syntactic constructions - syntactic units. The sense of syntactic unit can be opened in terms of the first order logic. There are four types of syntactic units:

notion denotes a general, possibly parameterized, class of objects: natural number.

term denotes an object, either by pointing to a concrete value: $(N, X * Y)$; or by quantifying a class denoted with a notion (every human, some natural number).

predicate denotes a property of an object: empty, divides N , is a subset of S . Applied to a term, a predicate forms a statement; applied to a notion as an attribute, it forms a new notion with a restricted class.

statement denotes a logical expression which may be true or false, be atomic or composed from simpler statements: $X > Y$, every divisor of N is a natural number, there exists a countable set.

Let's consider an example. Suppose we have the following facts:

F1: Confucius is human.

F2: Every human is mortal.

The given facts may be expressed by such formulas of the first order language:

F1: $P(\text{Confucius})$.

F2: $(\forall x)(P(x) \rightarrow Q(x))$.

Here $P(x)$ and $Q(x)$ - the predicates representing accordingly « x is human » and « x is mortal ».

Accepting the given facts for axioms, it is uneasy to be convinced, that from them the statement $Q(\text{Confucius})$ is followed or, considering semantic value of predicate Q , «Confucius is mortal».

In ForTheL the specified statements can be written as follows:

Example 1:

1: [the Confucius] [a human/humans] [x is mortal]

2: Axiom. Every human is mortal.

3: Axiom. Confucius is a human.

4: Proposition. Confucius is mortal.

The first line represents the list of three introductors, necessary for translation. Key words of language are allocated by a bold font. The second and third lines contain the representation of statements F1 and F2 accordingly. In the fourth there is a final statement which represents the theorem.

It is obvious, that representation of the given example in ForTheL language is more comprehensible, than similar representation by formulas of the first order language as it allows keeping the semantic structure of the text and is intuitively clearer.

As it was already specified earlier, syntactic primitives (further simply primitives) are used to construct syntactic units. There are six types of base primitives: the class nouns, the definite nouns, adjectives, verbs, functional and predicate symbols. Class nouns define the general probably parametrical class of objects (set, number, an element of set X). They are used to construct notions. The definite nouns generate functions and constants (a zero, the minimal element of set). With their help terms are built. Adjectives and verbs form predicates (less, equal). There are two more base syntactic primitives. They are functional and predicate symbols. They serve as reductions and give a possibility to write statements and terms as formulas. For example $m!$ can serve as the reduced representation of the term: “the factorial of m ”. Besides, there is also a group of auxiliary primitives, formed from base primitives by special rules.

Let's return to the suggested example. In the first line there is a list of three introductors. Syntactically introductor represents the certain syntactic construction made in square brackets. Every introductor generates one or more primitives. Actually primitive will be transformed to a new rule of the ForTheL language grammar which enables to process a corresponding construction. The first introductor [the Confucius] is the definite noun introductor and it generates a new constant Confucius. In an example it is used as a name of the philosopher. It is necessary to notice, that introductors of the definite nouns begin with a keyword "the" (Further in the text keywords of the language will be allocated with a bold italic font). Introductors, beginning with a keyword "a" or "an", serve for class nouns generation. In the given example we use introductor [a human/humans] to define the class of all humans. The following grammatical rule will correspond to the generated primitive: `primClassNoun-> (human|humans) [names]`, where names – non-terminal symbol of the language. (Non-terminal symbols of the language further in the text will be allocated with italic font). Another introductor [x is mortal], submitted in the example, serves for generation of an adjective. Introductor of an adjective should begin with a variable and the word "is" following it. The corresponding rule will have such representation: `primAdjective -> (mortal)`.

Let's pass to strict consideration of introductor's syntactic structure. First of all, it is necessary to specify, that introductors share on two basic types – non-symbolic (textual) and symbolical. The class nouns, the definite nouns, adjectives and verbs are considered textual introductors, to symbolical accordingly introductors of functional and predicate symbols. The type of introductor is defined by its syntax and a pattern. As well as introductors, there exist two types of patterns - textual and symbolical. The textual pattern will consist of sequence of words and variables. This sequence should begin with the list of words, and further necessarily followed by the list of variables divided by words. All words should contain small Latin letters and have length more than one symbol as under the

arrangement the singleton big and small letters of the Latin alphabet are considered as variables. The word can have the second form. It is separated from a word by a symbol “/” and intended for the indication of the plural or past form of a word, for example: man/men. Explaining above told, we shall result an example of a textual pattern: mul/multiplication of A and B. The symbolical pattern will consist of sequence of variables and symbolical words (a symbolical word - sequence of symbols without separators). Such pattern can begin with a variable or a symbolical word, further followed by the list of variables divided by symbolical words. Besides the symbolical pattern can be a name of function with the list of the variables specifying its arity surrounded by brackets, or it may be a simple word with variable after it.

Now let's make a summary of introductors's syntax in ForTheL. The introductors of class nouns begins with a keyword "a" or "an"; the definite nouns - with "the", and then the textual pattern should follow. Introductor of an adjective and a verb begins with a variable, further in the verb introductor the textual pattern simply follows, and in the adjective's introductor before a textual pattern the keyword "is" should be placed. Besides each of these introductors may contain a synonym following after symbol “@”. Symbolical introductor consist only of a symbolical pattern and the target statement separated by a symbol “@”. The type of a symbolical pattern is determined by type of the used statement.

ForTheL-based approach for representing formal specifications

Note: Concepts of "requirement", "specification" and others will be understood in sense of definitions resulted in the “Definitions” section:

In this section will be demonstrated, some possibilities of the ForTheL language for capturing formalized knowledge (formalized requirements). As it was mentioned in previous text, ForTheL is a formal language designed to be close to natural English. The grammar of ForTheL simulates constructions of correct English sentences. It has both a formally defined syntax and semantics. A formal syntax means that it is defined by a context-free grammar. The semantics is defined by first-order logic formulas (FOF). Every ForTheL statement may be finally translated into such formula or set of formulas.

The language may be adapted for writing formal statements in different domains by special constructions, called introductors. For example, suppose we want to write “The weather is fine” in ForTheL. First of all we have to introduce the predicative adjective “fine” and the noun “weather”. The following two introductors do it as follows:

1: [the weather]

2: [x is fine]

Now, we are ready to write the above mentioned statement:

3: Fact. The weather is fine.

The semantics of this statement is represented by such FOF:

isFine(Weather).

This simple example gives the glue to how formal facts may be written in ForTheL. Of course interpretation of predicates is left to the author of the text. But, some of these may be described in ForTheL with Definition construction.

In the requirements and specifications domain ForTheL may be used to write formal requirements and specifications or in semi-formal requirement templates for specifying precondition, scenario and post condition statements.

Then some properties of such requirements may be automatically checked with the help of an SAD prover or any other general purpose prover (SPASS, Vampire, etc). For example requirements may be checked for contradiction (see ForTheL and SAD system examples).

The advantage of such approach is that requirements and specifications are defined formally and their representation is close to natural language, so there are no need in other semi-formal languages and special linguistic procedures to obtain formal semantics and then process it.

«Additionally, it allows aspects of the software to be stated explicitly, without ambiguity, and can make many implicit aspects of the software explicit, thus enabling them to be analysed with greater ease than with some other methods.

The explicit statement of properties in axiomatic form is especially useful in capturing and analysing software requirements, where contradictions, conflicts, errors, and incompleteness's can be more easily trapped and eliminated at an early stage in development»(“Advances in Safety Critical Systems - Results and Achievements from the DTI/EPSRC R&D Programme” edited by Mike Falla).

ForTheL in Requirements Engineering

The software project may be described by such concepts as entity, interaction, attribute, function, the interface (architecture of system), the requirement, the specification (includes normal, test and critical cases), the working plan with its tasks. That's why, first of all, it is necessary to enter these concepts in ForTheL by means of corresponding introductors.

Initially requirements and specifications are expressed by the user in a natural language. Then while analysis and refinement they will be structured with the set of concepts mentioned above.

In this section the opportunity of using the ForTheL language instead of natural for capturing requirements was examined. Such approach will allow making in a consequence various operations with requirements automatically or automated. For example, to check the requirements for consistency. Except that, such requirements will meet a strict certain syntax, and have unambiguous treatment.

Let's consider the natural language requirement taken from the description of some system:

The environment must be stable and be able to recover from system crashes.

This requirement concerns to quite concrete environment, therefore it is necessary for considering it as a definite constant. Therefore for its introduction in language it is necessary to use such introductor:

[the environment]

The adjective specifying property «to be stable» will be entered into language like this:

[x is stable]

Besides it is necessary for us to describe property «to be capable», function «to be restored after» and a class of all of «system failures»:

[x is able to y] [the recover from y]

[a system crash/crashes]

Having such definitions of concepts, the resulted requirement in ForTheL language may be written. It is necessary to notice, however, that each requirement can be written in the different ways depending on the defined concepts. If concepts are fixed, there can be various equivalent forms of requirements representation, but semantics of them will be identical.

It is also important to notice, that the above mentioned requirement in a natural language contains ambiguity so - it is not complete. Ambiguity consists in expression «to recover from system crashes». Because it supposes various treatments:

«to recover from some system crashes» or «to recover from any system crash».

Stronger and more usual treatment will be accepted in the work: «to recover from any system crash».

This ambiguity has been revealed while translating requirements to ForTheL language.

The final requirement in ForTheL is looked as follows:

The environment must be stable and must be able to recover from any system crash.

It shall be noticed, also, that for supporting requirements syntax it was necessary to make some changes in initial ForTheL grammar, therefore it will be impossible to check up the given example in on-line versions of the SAD system (<http://ea.unicyb.kiev.ua>).

In the personal version of the system developed by the author of the work (see Bachelor Thesis) the processing of the given example will look like following:

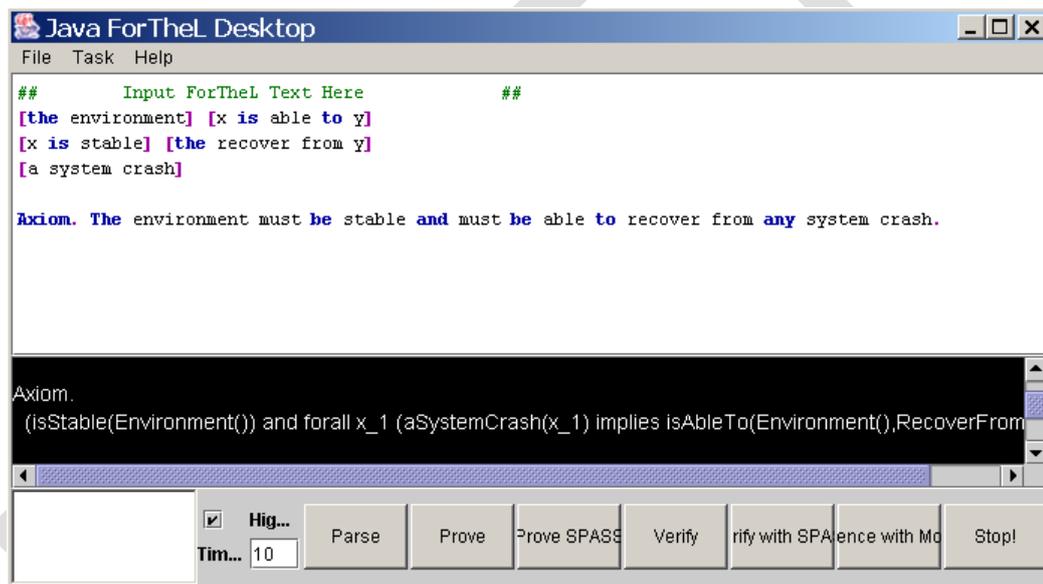


Figure 7: SAD Screenshot

Thus, the given example shows basic capabilities of the ForTheL language for process requirements and to reveal in them potential discrepancies.

Being based on the features of the ForTheL language it may be used for capturing the requirements which can be expressed within the first order of logic.

The procedure of accepting requirement through the input form (a web page) may be follow: The user enters ForTheL representation of requirements. Before accepting changes (modification of the existed requirement, adding a new requirement), ForTheL translator checks a syntactic and semantic correctness of the entered text. In case of a mistake the corresponding message will appear on the

screen. If translation has passed successfully, the ForTheL description together with corresponding first order logic representation will be stored in data base. Thus, the syntactic correctness of the entered requirement will be guaranteed. There will be a formal representation as the formula and further such formal representation can be processed without participation of the ForTheL translator, saving thus computing resources.

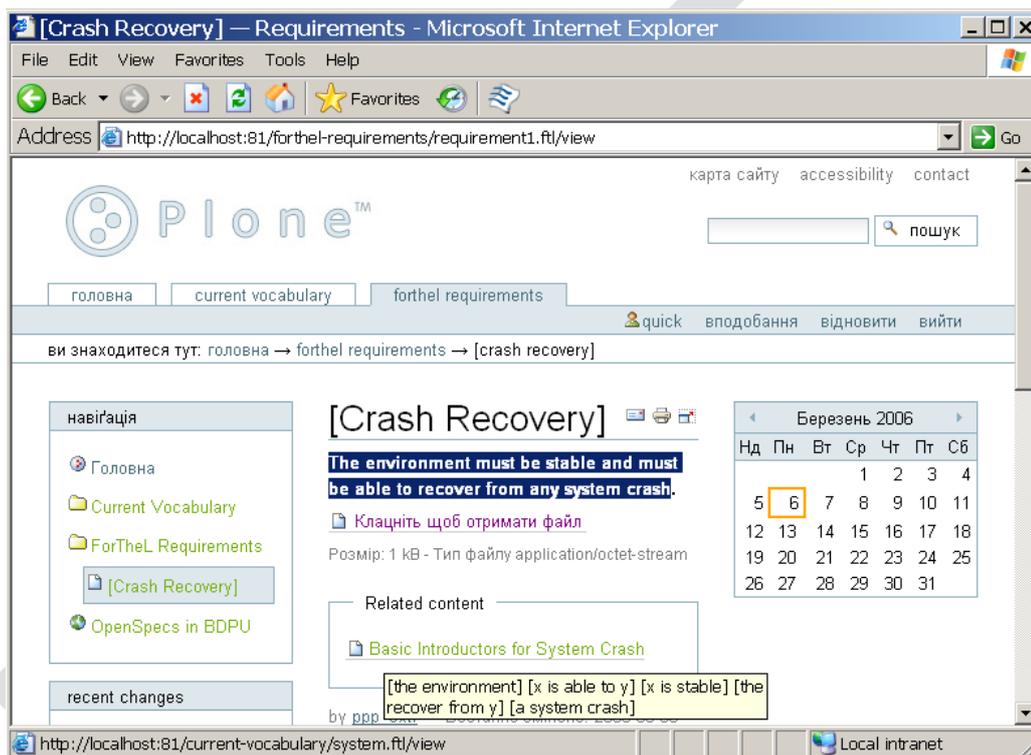


Figure 8: Screenshot

The prototype of the system's interface is shown on the picture above. The prototype just shows how the requirements may be structured. Further, more functional system will be described and the results of development are presented.

This first prototype was built using content-management system (CMS) Plone [20]. A **CMS** is a tool that enables a variety of (centralized) technical and (decentralized) non technical staff to create, edit, manage and finally publish a variety of content (such as text, graphics, video, and so on) whilst being constrained by a centralized set of rules, process, and workflows that ensure a coherent, validated Web site appearance.

The content is stored in the tree like structure:

Requirements :

Current Vocabulary:

General Introductors

Introductors

ForTheL Requirements:

Requirement: Crash Recovery

....

The current vocabulary must contain all necessary introductors for specifying requirements. The ForTheL requirements section is the list of statements and references. Each statement represents requirement. The references refer to other related requirements and sub lists of introductors from the current vocabulary.

After developing such system, the next step may be the introduction of typical patterns of requirements. The user will have an opportunity to choose one of the offered patterns, and then to fill in it the missing parameters. Here the typical pattern is understood as the ForTheL language sentence with non-filled arguments. For example: **The system must execute ...(some_action)... if and only if ...(condition)....**, Here (some_action) and (condition) are non-filled arguments.

Such patterns will allow different system's users to write requirements in a uniform format and grammatic style that is also important component of qualitatively made requirements.

Unfortunately it is necessary to pay attention on the fact that not all requirements may be expressed in ForTheL. The reason is that the ForTheL has an expressiveness of the first order logic. But it is useful when requirements may be expressed in terms of entities, associations (relations) between them, properties,

etc. There exists classes of requirements that may be effectively expressed in ForTheL.

The example of such class is constraint requirements. The importance of this class is that the constraints may be checked for consistency. An example of a constraint might be:

biplane(X) => CardinalityOf(wing(X)) = 2;

Natural language meanings: “The biplane has two wings”.

The ForTheL representation and translation is shown on the picture below:

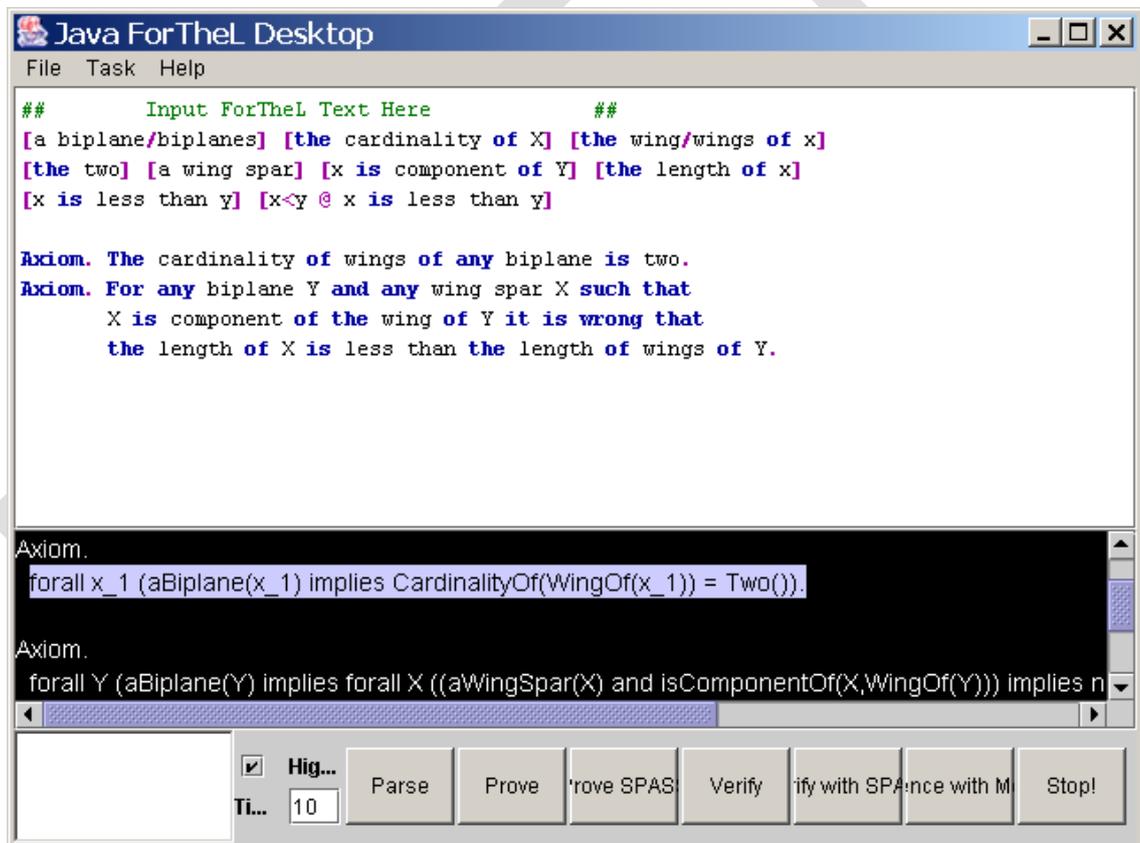


Figure 9: Screenshot

ForTheL listing:

[a biplane/biplanes] [the cardinality of X]
[the wing/wings of x] [the two] [a wing spar]
[x is component of Y] [the length of x]
[x is less than y] [x < y @ x is less than y]

Axiom. The cardinality of wings of any biplane is two.

Axiom. For any biplane Y and any wing spar X such that
X is component of the wing of Y it is wrong that
the length of X is less than the length of wings of Y.

Further in work, the system for processing requirements based on ForTheL
will be discussed in more details.

The Design of System for Managing Requirements

In this section the design of the system for managing software requirements will be described. The system is developed for using during Requirements Engineering in software development lifecycle. As practical result of work the prototype of such system is presented. Its name is Easy Requirements Engineering (ERE).

It is necessary at once to note, that the system should support the collective and collaborative work. The interaction between clients (users) and system is carried out through the network (local or global). All shared data is stored in repository on the server.

As the system should support multiple access to the data, it should include mechanisms for protection from non-authorized access. For this purpose the system has a role based data access. There are four roles of the users - administrator, engineer of the requirements, developer and user. The administrator has the complete control above system and repository. The requirements' engineer has a complete access to his own information. He also can observe the information of other system users. The developer has an opportunity to look through the data in repository and to change the requirement's completeness status, above which he works. The user works with system in a read only mode.

The information in repository is stored in tree-like structure. Each tree node represents a class of the requirements or a requirement instance (certain requirement specification). The requirement's class is a structure composed of a name and a set of attributes. The users of the system (manager, engineer of the requirements) have an opportunity to create requirement's classes and its instances. The attribute is triple (name of attribute, type of attribute, value of attribute), they also are created by the user from the types, presented in the system. In the prototype there are such types of attributes, as number, string, boolean, file

(document), reference, ForTheL text. Besides the class can have properties - "verifiable", "specifiable", "formality", etc.

Let's result an example of the requirements structuring:

Requirements:

System Requirements:

- User's Requirements
- Constraints
- Quality Requirements

Software Requirements:

- functional
- non-functional

Such structured representations of the information will allow system to functioning more effectively. Let's notice, that the classes should support hierarchical inheritance, i.e. inherit all attributes and properties of high level classes. The intersection of attributes and properties of parent and child classes is not supposed.

Now, consider conceptual scheme of the system architecture:

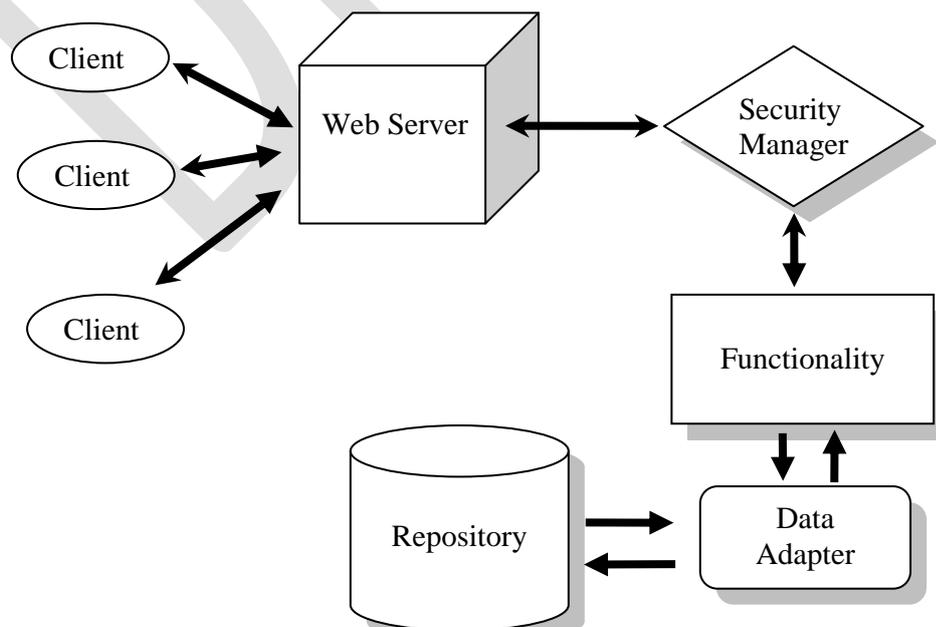


Figure 10: Architecture

It is composed of the web server, with which the users work through a browser. The information is stored in the repository. The information from repository is provided by the data adapter component. Further the information will be subsequently processed. The security manager grants or denies access to the system functionality according to the user rights. As it was already said, the access rights are defined on the basis of roles. The system enables to configure the users' roles, and also to add the new users in system.

The data adapter provides independence of system from a type of repository. I.e. if necessary to modernize or to replace it, it is enough to supply system with the corresponding data adapter component without changing functional part of system.

The entire repository structure is represented by the following ER-diagram (Figure 11: Repository Scheme):

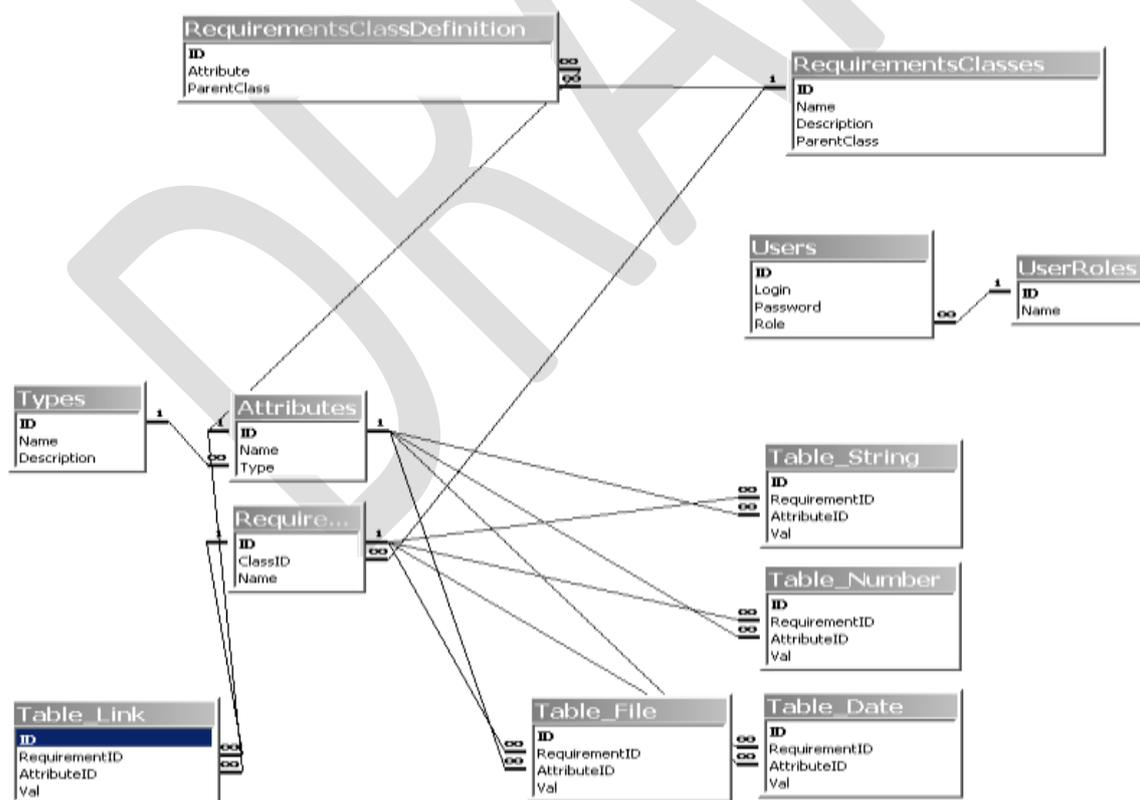


Figure 11: Repository Scheme

Now, let's consider the software tools used for building the prototype of such system. As DBMS MS Access is used, because it is the most widespread mean for small and average level databases. In case of further development of system it is possible to use more powerful DBMS, such as MS SQL Server, Oracle or Informix.

The web application is developed by J2EE methodology with the use of Java programming language and Java Servlets technologies: JSP, EJB. For the web server and Java Servlets container the freely distributed software Tomcat 5 is chosen.

The work in system begins with authorization and reception of access to repository. After granting the access to repository the further interaction with system occurs from the main page of system. Its working area is shown below:

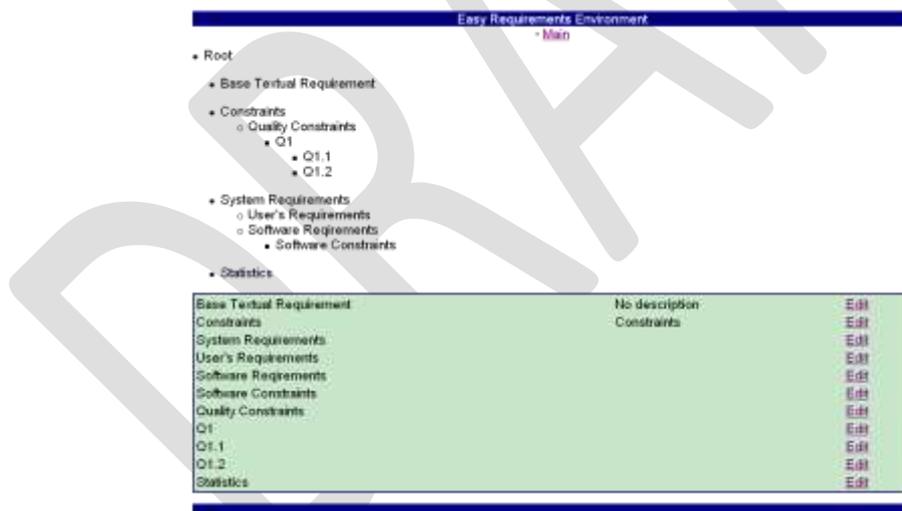


Figure 12 : Work Area

On it the tree of requirements classes is represented. At presence of necessary access rights, the user of the system has an opportunity to edit (to add, delete, change) this requirements storing structure.

For example, dialogue of creation of a new class has such appearance:

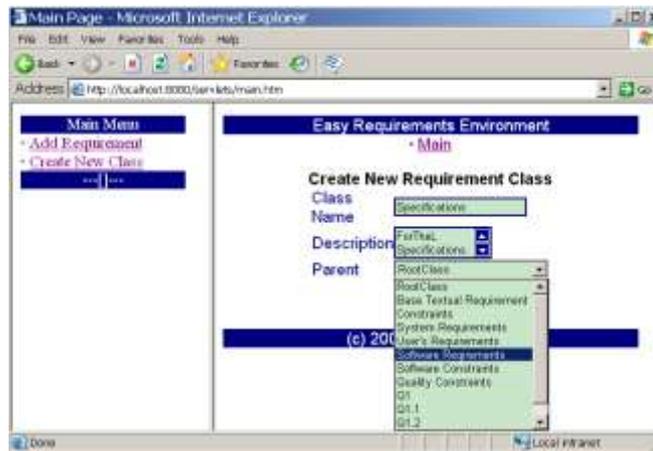


Figure 13: Create Requirement Class Dialog

There is an opportunity to choose an inherited class, and to edit attributes of the created class. The new class inherits all attributes of parental class requirements.

After requirements storage structure creation by the system's administrator or the requirements engineer, other users of system can look through its contents, and add the data (the requirements engineer and the developer).

Let's consider requirements classes which include ForTheL language attributes. For supporting such functionality the system contains built-in type "ForTheL_Text", and also necessary procedures of processing and the analysis. For this purpose in system the personal Windows version of SAD([29]) system is used. After editing ForTheL attribute (change, creation) the system transfers the content to the ForTheL language processor. If the text has been written syntactically correctly the information will be stored in a repository for the subsequent processing. Otherwise the user is notified about the mistake and has an opportunity to correct it. Thus, only correct information is stored in repository.

The general scheme of system states is represented on figure (Figure 14: ERE state diagram).

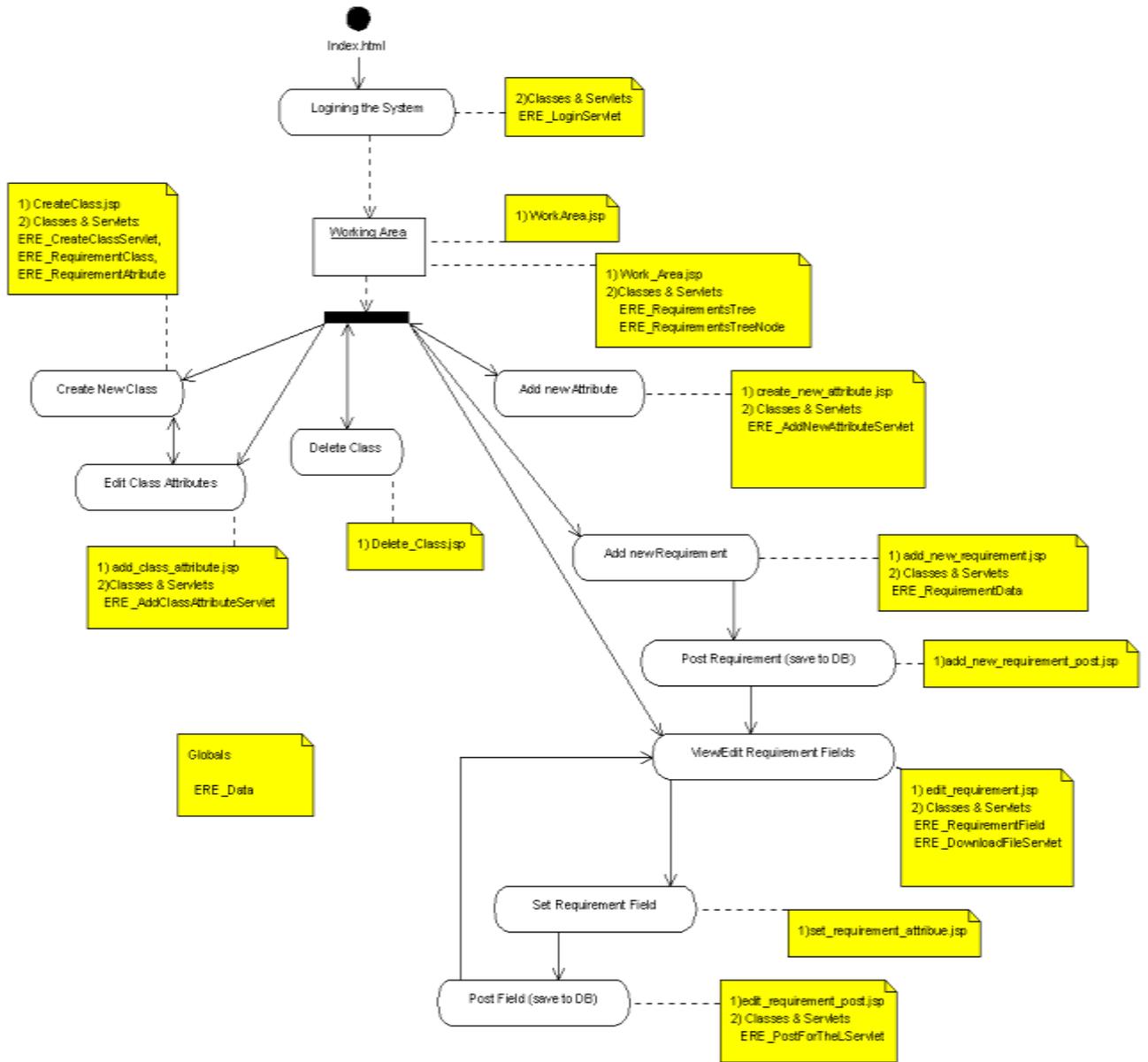


Figure 14: ERE state diagram

Conclusions

The results of the work are divided into two parts theoretical and empirical studies. From theoretical point of view in the thesis the investigation of formal methods in requirements engineering has been performed. Also, a new approach based on ForTheL language has been proposed. The goal of the approach is to increase the level of understandability and usage of formal methods by non specialists in the field of Requirements Engineering. This property is very important especially in the field of software development.

The proposed approach is demonstrated by some examples in the text. The complete example of formalization and verification of a simple security protocol is given in the Appendix.

The approach is also supported by software tools. They are represented by the system for managing software requirements. It is called Easy Requirements Environment. The system uses System for Automated Deduction for processing ForTheL specifications of the requirements. It is able to check specifications for mistakes and prove the given properties of developing system.

So, it can be concluded that stated goals have been achieved in the work. The thesis was participated in the final round of whole Ukraine competition of diploma works in IT sphere – UkrProg'2006 Junior sponsored by Intel corporation. Preliminary results of the work were presented at the student's session during RoCoLi summer school 2005, in Sinaia, Romania (September, 2005).

Definitions

In this section the base terms used in the previous text are described briefly. For some terms the common definitions are given. The terms are organized in the table. In the third column the reference to the source of the term's definition is given. Undefined terms are considered as in [1][12].

What is Use Case?	A use case captures a contract between the stakeholders of a system about its behaviour. The use case describes the system's behaviour under various conditions as it responds to a request from one of the stakeholders, called primary actor.	[13]
Use Case writing technique	Is the moment-to-moment thinking or actions people use while constructing the use cases.	[13]
Use Case standards	Expression of what the people on the project agree to when writing their uses cases.	[13]
Use Case quality	Expression telling whether the use cases that have been written are acceptable for their purpose.	[13]
Stakeholder	Any person or organisation that can be affected by the system (or has a vested interest in system's project).	
Requirement	A text, more or less formal that defines qualities that a system shall or should have. Note: Requirements should avoid implicit decisions.	
Specification	Are the translations of the requirements in measurable attributes of the system. A specification is a document (on paper or in a computer), which defines (specifies) a system. It can contain: Requirements on the system, Test Cases to match the requirements, Manuals (specifies operators roles), Definition of environment (enabling systems).	
The Universe of Discourse	Is the overall context in which the software will be developed. It includes all the sources of information and all the people related to the software. These people are referred to as the actors in this universe of discourse	[27]
Actors	Are the different people involved in the universe of discourse. Basically, actors may be divided into users on the demand side and software engineers on the supply side.	[27]

Abbreviations and Acronyms

Abbreviation	Description
AMN	Abstract Machine Notation
ASM	Abstract State Machines
ATP	Automated Theorem Proving
BCP	Best Common Practice
CIM	Computer Independent Model
CMS	Content Management System
CSP	Communicating Sequential Processes
DB	Database
DBMS	Database Management System
EJB	Enterprise Java Beans
FL	Formal Language
FOF	First-order logic formula
FOL	First-order language
ForTheL	Formal Theory Language
J2EE	Java 2 Enterprise Edition
JSP	Java Server Page
MDA (1)	Model Driven Architecture methodology
MDA (2)	Model Driven Architecture
NL	Natural Language
OMG	Object Management Group
RAISE	Rigorous Approach to Industrial Software Engineering
RE	Requirements Engineering
RM	Requirements Management
SE	Software Engineering
SRS	Software Requirements Specifications
STD	Standard
UML	Unified Modelling Language
V&V	Verification and Validation
VDM	Vienna Development Method

ForTheL and SAD system examples

Example 1

The security protocol

The example is based on the description of the Needham-Schroeder Public-Key Protocol from [22] on page 208.

Initial description of the protocol

This protocol uses public-key cryptography. Each person has a private key, known only to him, and a public key, known to everybody. When Alice wants to send Bob a secret message, she encrypts it using Bob's public key, and sends it to Bob. Only Bob has the matching private key, which is needed in order to decrypt Alice's message. The core of the Needham-Schroeder protocol consists of three messages:

1. $A \rightarrow B : \{Na, A\}_{Kb}$
 2. $B \rightarrow A : \{Na, Nb\}_{Ka}$
 3. $A \rightarrow B : \{Nb\}_{Kb}$
- (1)

First, let's understand the notation. The protocol used a notion of nonce. A typical nonce is a 20-byte random number. Each message that requires a reply incorporates a nonce. The reply must include a copy of that nonce, to prove that it is not a replay of a past message. In the first message, Alice sends Bob a message consisting of a nonce generated by Alice (Na) paired with Alice's name (A) and encrypted using Bob's public key (Kb). In the second message, Bob sends Alice a message consisting of Na paired with a nonce generated by Bob (Nb), encrypted using Alice's public key (Ka). In the last message, Alice returns Nb to Bob,

encrypted using his public key. When Alice receives Message 2, she knows that Bob has acted on her message, since only he could have decrypted $\{Na, A\}_{Kb}$ and extracted Na. That is precisely what nonces are for. Similarly, message 3 assures Bob that Alice is active.

Property checking in the SAD system

The protocol was widely believed to satisfy the following property: Na and Nb were secrets shared only by Alice and Bob. But, this property is not true.

The following scheme shows that there exists a possibility for adversary (C) to get private secrets of the user (B), hiding from him (as A).

- | | |
|--|--|
| 1. $A \rightarrow C : \{Na, A\}_{Kc}$ | 1'. $C \rightarrow B : \{Na, A\}_{Kb}$ |
| 2. $B \rightarrow A : \{Na, Nb\}_{Ka}$ | |
| 3. $A \rightarrow C : \{Nb\}_{Kc}$ | 3'. $C \rightarrow B : \{Nb\}_{Kb}$ |

In order to check this property we will use SAD system and its input language ForTheL for formalizing protocol specification.

First, let's define agents as protocol users and action of sending message from one agent to another:

[an agent/agents][x send y to z]

For security purposes we need to describe the notion of key, encryption and decryption functions:

[a key] [the encrypt of m with k]

[the decrypt of m with k] [the public key of X]

The protocol consists of three types of messages for each step of information exchanging:

**[the messagea of X and Y] [the messageb of X and Y]
[the messagec of X]**

Besides, we need the notion of the agent nonce and possibility for describing which nonces each agent knows:

[the nonce of x] [x know/knows y]

Now we are ready to formalize the protocol core, but before, let's introduce function symbols for more convenient notation:

[{N,A} @ messagea of N and A]

[-{N,M} @ messageb of N and M]

[{N} @ the messagec of N][nonce(X) @ the nonce of X]

[Pub(X) @ the public key of X][X-(K) @ encrypt of X by K]

The core of the protocol and the adversary behavior are presented by following four axioms:

Axiom First. For any agent A,B

A send {nonce(A), A}-(Pub(B)) to B.

Axiom Second. For any agent B,A

if A send {nonce(A), A}-(Pub(B)) to B

then B send (-{nonce(A),nonce(B)})-(Pub(A)) to A.

Axiom Third. For any agent A,B

if B send (-{nonce(A),nonce(B)})-(Pub(A)) to A

then A send {nonce(B)}-(Pub(B)) to B.

Axiom Adversary. For any agents A,B,C

if C knows nonce(A) then

C send {nonce(A),A}-(Pub(B)) to B.

Having such a description of protocol we are able to formulate and then verify the following protocol property using the SAD system:

Proposition. Let A,C be agents.

if A send {nonce(A), A}-(Pub(C)) to C then

for any agent B such that B != A

if (C does not send {nonce(C),C}-(Pub(B)) to B) and

(C send {nonce(A),A}-(Pub(B)) to B)

then B does not know nonce(C) and B know nonce(A).

This property tells that protocol does not satisfy security requirements.

Entire Protocol specification in ForTheL:

[an agent/agents][x send y to z] [a key]

[the encrypt of m with k] [the decrypt of m with k]

[the public key of X]

[the messagea of X and Y] [the messageb of X and Y]

[the messagec of X]

[the nonce of x] [x know/knows y]

[{N,A} @ messagea of N and A]

[-{N,M} @ messageb of N and M][{N} @ the messagec of N]

[nonce(X) @ the nonce of X]

[Pub(X) @ the public key of X]

[X-(K) @ encrypt of X with K]

Axiom. For any agent A the public key of A is a key.

Axiom. For any agent A A knows nonce(A).

**Axiom. For any agents A,B if $A \neq B$ then A knows nonce(B)
iff (A send {nonce(A), A}-(Pub(B)) to B) or
(A send (-{nonce(B),nonce(A)})-(Pub(B)) to B).**

**Axiom. For any agent A,B
A send {nonce(A), A}-(Pub(B)) to B.**

**Axiom. For any agent B,A
if A send {nonce(A), A}-(Pub(B)) to B
then B send (-{nonce(A),nonce(B)})-(Pub(A)) to A.**

**Axiom. For any agent A,B
if B send (-{nonce(A),nonce(B)})-(Pub(A)) to A
then A send {nonce(B)}-(Pub(B)) to B.**

**Axiom. For any agents A,B,C if C knows nonce(A)
then C send {nonce(A),A}-(Pub(B)) to B.**

#(Property)

**Proposition. Let A,C be agents.
if A send {nonce(A), A}-(Pub(C)) to C then
for any agent B such that $B \neq A$
if (C does not send {nonce(C),C}-(Pub(B)) to B)
and (C send {nonce(A),A}-(Pub(B)) to B)
then B does not know nonce(C) and B know nonce(A).**

The results of text processing in SAD system:

[ForTheL] parsing successful

[Reason] verification started

[Reason] line 48: goal: if A send {nonce(A), A}-(Pub(B)) to B then A send
{nonce(B)}-(Pub(B)) to B

and B knows nonce(A).

[SPASS] Haigha started

[SPASS] parsing prover output

[SPASS] SPASS V 2.1

.....

[SPASS] SPASS V 2.1

[SPASS] SPASS beiseite: Proof found.

[SPASS] Problem: Read from stdin.

[SPASS] SPASS derived 1 clauses, backtracked 0 clauses and kept 19 clauses.

[SPASS] SPASS allocated 593 KBytes.

[SPASS] SPASS spent 0:00:00.05 on the problem.

[SPASS] 0:00:00.01 for the input.

[SPASS] 0:00:00.03 for the FLOTTER CNF translation.

[SPASS] 0:00:00.00 for inferences.

[SPASS] 0:00:00.00 for the backtracking.

[SPASS] 0:00:00.01 for the reduction.

[SPASS] Haigha ended

[SPASS] Command line is:spass.exe -Stdin -DocProof=0 -PProblem=0 -PGiven=0 -CNFOptSkolem=0

[Reason] verification successful

[Main] session finished in 00:02.40

[Main] 00:00.25 in [ForTheL] -- 00:01.05 in [Reason] -- 00:01.10 in

[SPASS]

Example 2

Reasoning in natural-like language

(Taken from original SAD examples):

[an animal] [a wolf] [a fox] [a bird] [a worm] [a snail]

[a plant] [a grain] [x eats/eat y] [x is smaller than y]

Every animal A that does not eat some plant eats

(every animal that eats some plant and is smaller than A).

Every wolf is an animal. Every fox is an animal.

Every bird is an animal. Every worm is an animal.

Every snail is an animal. Every grain is a plant.

There exist a wolf and a fox and a bird

and a worm and a snail and a grain.

Every worm is smaller than every bird.

Every snail is smaller than every bird.

Every bird is smaller than every fox.

Every fox is smaller than every wolf.

Every worm eats some grain. Every snail eats some grain.

Every bird eats every worm. Every bird eats no snail.

Every wolf eats no grain. Every wolf eats no fox.

Proposition.

There exists an animal A and an animal B

such that A eats B and B eats every grain.

References

- [1]. IEEE Computer Society, Guide to the Software Engineering Body of Knowledge (SWEBOK), version 2004, (chapter 2).
- [2]. IEEE Std 830-1998, IEEE Recommended Practice for Software Requirements Specifications, IEEE 1998.
- [3]. Alexander Lyaletski, Konstantin Verchinine, Anatoli Degtyarev, and Andrey Paskevich. System of Automated Deduction (SAD): Linguistic and Deductive Peculiarities. In M.A. Klopotek, S.T. Wierzchon, and M. Michaliwicz, editors, Advances in Soft Computing: Intelligent Information Systems 2002, Physica-Verlag, Springer, pages 413-422, 2002.
- [4]. Lyaletski A., Verchinine K., and Paskevich A. On verification tools implemented in the System for Automated Deduction. Proceedings of the 2nd Workshop on Implementation Technology for Computational Logic Systems (ITCLS), Pisa, Italy, 2003, 25—36
- [5]. Wikipedia, the free encyclopedia, article:
http://en.wikipedia.org/wiki/Software_requirements_specification
- [6]. Robert Japenga, How to write a software requirements specification,
<http://www.microtoolsinc.com/Howsrs.php>
- [7]. Giuseppe Lami, QuARS: A Tool for Analyzing Requirements, Technical Report, September 2005
- [8]. Donald Firesmith, Modern Requirements Specification, Journal of Object Technology, Vol. 2, No. 1, March-April 2003
- [9]. Terry Bahill, Steven Henderson, Requirements Development, Verification, and Validation Exhibited in Famous Failures, System Engineering, vol. 8, No. 1, 2005, Wiley Periodicals, Inc.
- [10]. Anthony Finkelstein, Wolfgang Emmerich, The Future of Requirements Management Tools, (Form Internet).

- [11]. Karl E, Wiegers, Automating Requirements Management, Software Development, July 1999
- [12]. Коммервилл, Иан. Инженерия программного обеспечения, 6-е издание. : Пер. с англ. – М. : Издательский дом «Вильямс», 2002. – 624 с.: ил. ISBN 5-8459-0330-0 (rus), ISBN 0-201-39815-X (eng), Pearson Education Limited, 2001.
- [13]. Alistair Cockburn, Writing Effective Use Cases, Addison-Wesley, 2000.
- [14]. F. Fabbrini, M. Fusani, S. Gnesi, G. Lami, An Automatic Quality Evaluation for Natural Language Requirements, (<http://fmt.isti.cnr.it/WEBPAPER/P11RESFQ01.pdf>)
- [15]. Vershinin K., Paskevich A.: ForTheL - the language of formal theories, International Journal of Information Theories and Applications 7 (3) (2000) 120-126.
- [16]. Volere Requirements Specification Template (<http://www.volere.co.uk>).
- [17]. Kendra Cooper, Mabo Ito, Formalizing a Structured Natural Language Requirements. Specification Notation. http://www.utdallas.edu/~kcooper/research/INCOSE_2002_SRRS.pdf
- [18]. María Carmen Leonardi María Virginia Mauco, Integrating Natural Language Oriented Requirements Models into MDA. (http://wer.inf.puc-rio.br/WERpapers/artigos/artigos_WER04/Maria_Leonardi.pdf)
- [19]. Matthias Riebisch, Michael Hübner, Refinement and Formalization of Semi-Formal Use Case Descriptions, Ilmenau Technical University, 2004. (<http://www.theoinf.tu-ilmenau.de/~riebisch/mbd/riehueb-ECBS-MBD.pdf>)
- [20]. Enfoldsystems, The Definitive Guide to Plone, first edition, 2005 (<http://plone.org>)
- [21]. Verchinine K. and Paskevich A.: ForTheL reference manual, Unpublished draft, hosted at <http://ea.unicyb.kiev.ua/download/forthel.ps.gz>, 2004

- [22]. Tobias Nipkow, Lawrence C. Paulson, Markus Wenzel, Isabelle. A Proof Assistant for Higher-Order Logic, tutorial, 2004
- [23]. Daniel D.K. Sleator, Davy Temperley, Parsing English with a Link Grammar, School of Computer Science, Carnegie Mellon University, CMU-CS-91-196, October, 1991.
- [24]. Flesch-Kincaid Readability Test - Wikipedia, the free encyclopedia, (<http://en.wikipedia.org/wiki/Flesch-Kincaid>)
- [25]. Ivy Hooks, Writing Good Requirements, Published in the Proceedings of the Third International Symposium of the NCOSE – Volume 2, 1993.
- [26]. Donald Firesmith, Specifying Good Requirements, Journal of Object Technology (JOT), Vol. 2, No. 4, July-August 2003
- [27]. Julio Cesar Sampaio do Prado Leite, Peter A. Freeman, Requirements Validation Through Viewpoint Resolution, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 17, NO. 12, DECEMBER 1991.
- [28]. Karl E. Wieggers, Writing Quality Requirements, Software Development magazine, May 1999
- [29]. Процик П.П., науковий керівник Нікітченко М.С. «Мова ForTheL та особливості реалізації системи автоматизації дедукції (САД) на платформі MS Windows», бакалаврська кваліфікаційна робота, Київ, 2005.
- [30]. Eric J. Braude, SOFTWARE ENGINEERING: An Object-Oriented Perspective, Wiley Computer Publishing, John Wiley & Sons, Inc. (Эрик Дж. Брауде, Технологии разработки программного обеспечения, изд. Питер, 2004, ISBN 5-94723-663-X).
- [31]. .Wikipedia, the free encyclopedia (<http://en.wikipedia.org>)
- [32]. Geoff Sutcliffe, Geoff Sutcliffe's Overview of Automated theorem proving: (<http://www.cs.miami.edu/~tptp/OverviewOfATP.html>)
- [33]. <http://www.kestrel.edu/HTML/prototypes/kids.html>
- [34]. <http://ase.arc.nasa.gov/docs/amphion.html>
- [35]. <http://www.comlab.ox.ac.uk/archive/formal-methods.html>

- [36]. <http://i11www.ira.uka.de/~kiv/KIV-KA.html>
- [37]. <http://www.csl.sri.com/sri-csl-pvs.html>
- [38]. <http://ti.arc.nasa.gov/ase/docs/progsyn.html>
- [39]. <http://data.mpi-sb.mpg.de/internet/news.nsf/Spotlight/19991004>
- [40]. <http://www.di.ens.fr/~blanchet/crypto-eng.html>
- [41]. <http://homepages.inf.ed.ac.uk/s9808756/coral/>
- [42]. <http://www.cs.utexas.edu/users/moore/acl2/index.html>
- [43]. <http://www.cl.cam.ac.uk/Research/HVG/HOL/>
- [44]. <http://www.cli.com/software/nqthm/>
- [45]. <http://www.safelogic.se/>
- [46]. Y. Gurevich, P. Kutter, M. Odersky and L. Thiele (eds.), *Abstract State Machines: Theory and Applications*, Lecture Notes in Computer Science, volume 1912, Springer-Verlag, 2000. (ISBN 3-540-67959-6)
- [47]. E. Börger and R. Stärk, *Abstract State Machines: A Method for High-Level System Design and Analysis*, Springer-Verlag, 2003. (ISBN 3-540-00702-4)
- [48]. <http://alloy.mit.edu/>
- [49]. *The B-Book: Assigning Programs to Meanings*, Jean-Raymond Abrial, Cambridge University Press, 1996. ISBN 0-521-49619-5.
- [50]. *The B-Method: An Introduction*, Steve Schneider, Palgrave, Cornerstones of Computing series, 2001. ISBN 0-333-79284-X.
- [51]. Hoare, C. A. R. (1978). "Communicating sequential processes". *Communications of the ACM* 21 (8): 666–677. DOI:10.1145/359576.359585.
- [52]. *Robin Milner: Communicating and Mobile Systems: the Pi-Calculus*, Springer Verlag, ISBN 0521658691
- [53]. <http://www.iist.unu.edu/raise/>
- [54]. John Fitzgerald et al, *Validated Designs for Object-oriented Systems*, Springer Verlag 2005. ISBN 1852338814

[55]. Cliff Jones, Systematic Software Development using VDM, Prentice Hall
1990. ISBN 0138807337. Also available on-line and free of charge:
<http://www.csr.ncl.ac.uk/vdm/ssdvdm.pdf.zip>

[56]. Z ISO Standart

[http://standards.iso.org/ittf/PubliclyAvailableStandards/c021573_ISO_IEC_13568_2002\(E\).zip](http://standards.iso.org/ittf/PubliclyAvailableStandards/c021573_ISO_IEC_13568_2002(E).zip)

DRAFT