Elicitation of functional requirements from the application programming interface documentation for functional testing

1st Evgeny Gerlits
Software Engineering Department
Ivannikov Institute for System Programming of the RAS
Moscow, Russia
gerlits@ispras.ru

2nd Denis Kildishev
Software Engineering Department
Ivannikov Institute for System Programming of the RAS
Moscow, Russia
kildishev@ispras.ru

3d Alexey Khoroshilov
Software Engineering Department
Ivannikov Institute for System Programming of the RAS
Moscow, Russia
khoroshilov@ispras.ru

***Abstract*—We address a common problem in this paper. The only available documentation for a computer program consists of a user API documentation while we need to identify functional requirements and build test suite to test them. We describe a technique for functional requirements elicitation from the user API documentation. Requality software management tool is exploited in this technique. The tool has been used in several industrial software verification projects.**

***Index Terms*—requirements elicitation, requirements extraction, API documentation, Requality, requirements management, functional requirements, requirements markup, requirements catalog**

I. Introduction

Many computer programs provide an application programming interface (API). These are operating systems, software libraries and even social networks, messengers and online-services. So, let we have a computer program providing an API. The task is set to create functional tests for a subset of functions from this API.

API is usually specified in a user API documentation. This kind of documentation commonly includes overall description of the computer program and its subsystems, contains some kind of specifications of classes and should include specifications of functions or methods. Signatures of functions are declared in a programming language. The behavior of functions is usually written in a natural language. An API documentation of high quality should describe reaction of every interface function for all possible values of the arguments and internal states of the computer program. A function may react by:

- returning a particular value;
- modifying the internal state of the program;
- generating an exception or returning an error code.

In this paper, we suppose the only written source of the requirements is the user API documentation. Our experience in the field confirms these assumptions. A functional requirements specification of high quality is almost always available if it is required by a standard like DO-178C [1].

Software test engineers often extract functional requirements from the user API documentation implicitly, i.e. just by reading and analyzing the text without making an explicit requirements catalogue. We believe, this approach can be justified in some cases, for instance, when smoke tests are to be developed.

However functional testing of high quality implies assessment of completeness of a test suite. A common test adequacy criterion for functional testing is the percentage of the requirements verified by the tests. To be able to calculate the test coverage, every test should somehow be traced to the requirements it verifies.

As the behavior of functions is described in the user API documentation in plain text, an additional layer of requirements is needed in which every requirement is isolated explicitly and has a unique identifier [2]. This layer of requirements over the user API documentation is usually called the requirements catalogue.

In addition, standard ISO/IEC/IEEE 29148-2018 [3] defines key characteristics of requirements. Some of these characteristics are difficult to check in plain texts. For instance, these are completeness, verifiability and traceability.

In this paper, we publish a technique to build a catalog of functional requirements. To our mind, a technique is a set of actions aimed at one particular result. The API documentation is the primary source of the requirements in this technique. The functional requirements are not explicitly listed in the text of the API documentation as it is in the functional requirements specification. So, our technique aims at elicitation of the functional requirements from the plain text.

Extraction of functional requirements from the plain text is a challenge. The source of the problems is the focus of the API documentation on the needs of software users but not on the needs of software test engineers and developers. It describes the interface of a computer program. The completeness of a set of functional requirements is not the primary goal of the user API documentation.

The task is additionally complicated because of the ambiguity, inconsistency and overlapping of some statements in the text. Deduction of some statements in the text from some other statements is another problem among many other problems related to the use of natural language.

This paper is structured as follows. We explain our reasons to perform this study in chapter 2. The goal of this study and the reachability criterion for the goal are expressed in chapter 3. In chapter 4, we represent our technique for functional requirements elicitation. This technique has been elaborated during a series of industrial projects on software requirements elicitation. We briefly mention these projects in chapter 5. We overview investigations related to our study and show the novelty of our study in chapter 6. We explain why we reach the goal of this study with our technique and come to some conclusions in chapter 7. Chapter 8 contains a list of references to the cited literature.

II. Motivation

To our mind, challenges associated with elicitation of functional requirements from the API documentation may be overcome by applying:

- a proper requirements elicitation process;
- effective technical solutions for existing issues;
- automation of labor intensive tasks.

In this paper, we present a requirements elicitation process elaborated during a number of industrial projects. We also propose solutions for some issues concearning markup of API documentation text. We automate routine and labor intensive tasks with our requirements management solution Requality.

Industrial requirements management tools [22] usually come along with a requirements management process. These processes do not directly address the problem of requirements elicitation from written sources like API documentations or standards. In this paper, we declare a requirements elicitation process for our Requality requirements management tool.

III. Problem statement

Let a user API documentation is given. One should extract functional requirements from this documentation, build a catalogue of functional requirements and supplement the catalogue with new requirements obtained from other sources. A proper requirements elicitation technique should satisfy to the following requirements:

1) The technique should form a functional requirements catalogue in which every requirement has a unique identifier.
2) The technique should support tracing of requirements to text fragments from the documentation that represent this requirement.
   Such a traceability relation can be used to estimate coverage of the documentation text by the requirements markup.
3) The technique should tolerate possible changes in the text of the API documentation.
   Problems are often discovered in the documentation text during analysis. The author of the documentation fixes them and issues a new version of the documentation. Some text fragments may be changed due to fixes. These changed text fragments might already be traced to existing requirements in the previous version of the documentation. A mechanism is required to transfer the existing mapping of requirements to text fragments onto the next (fixed) version of the documentation.
4) The technique should assist in refinement of functional requirements.
   Requirements refinement is the primary way to improve understanding of the desired functions behavior.
5) The technique should assist in supplementation of the requirements catalogue with new requirements.
   New requirements are those that can not be deduced from the existing requirements. Every new requirement improves completeness of the set of requirements.
6) The technique should not rely on a particular natural language.

IV. Requirements elicitation technique

In this section, we describe different aspects our technique for functional requirements elicitation from the user API documentation.

A. Demo example

All examples in this paper refer to *select* function from POSIX [4] standard. This function waits until an event is registered for one of the file descriptors provided. There are three types of events:

- a file descriptor is ready for reading;
- a file descriptor is ready for writing;
- and error is registered for a file descriptor.

Function *select* is complex. It handles different file types differently. There are several file types:

- regular files;
- sockets;
- terminals and pseudo terminal.

In this paper, we assume for simplicity that *select* function is applied to regular files only.

Automation makes it easier and faster to extract requirements from the documentation. We apply Requality [5] requirements management tool in our industrial projects. Some images in this paper are screenshots of Requality graphical user interface.

B. Requirements catalogue structure

Let us build our requirements catalogue in the form of a *tree* [6] in which:

- verticies are requirements;

- an arc between two incident requirements represents the refinement relation, i.e. the requirement having a higher depth refines the requirement having a lower depth.

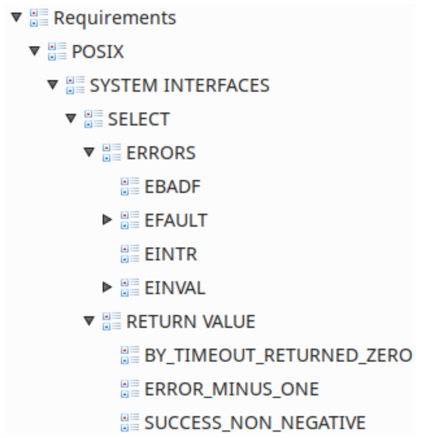
The following conditions should be checked before insertion of a new requirement into the requirements catalogue:

- the new requirement is interpreted unambiguously;
- the new requirement does not contradict the existing requirements;
- the new requirement can not be logically deduced from the existing requirements;
- the new requirement does not semantically intersect with the existing requirements.

In this chapter, we do not specify methods used to check the above conditions. Proving each condition is a challenge but it is often not necessarily. For instance, a requirement may be labeled as unambiguous if all teem members interpret the requirement in the same way. A meeting of all teem members can be appointed to address this issue.

If one of the above conditions is not met, we should fix the new requirement or the requirements catalogue or both. Thus, by adding a new requirement to the requirements catalogue:

- we refine another requirement;
- we improve the completness of the set of requirements.



```
▼ Requirements
  ▼ POSIX
    ▼ SYSTEM INTERFACES
      ▼ SELECT
        ▼ ERRORS
            EBADF
          ▶ EFAULT
            EINTR
          ▶ EINVAL
        ▼ RETURN VALUE
            BY_TIMEOUT_RETURNED_ZERO
            ERROR_MINUS_ONE
            SUCCESS_NON_NEGATIVE
```

Fig. 1. A fragment of a requirements catalogue

A fragment of a requirements catalogue is depicted on figure 1. Let us look at the sub-tree with the root named *SELECT*. The root requirement is refined by two requirements:

- a requirement to the return value of the function;
- a requirement to the value of *errno* variable after the function call.

Requality requirements management tool automatically generates internal identifiers for the requirements. We give mnemonic names to the requirements. All requirements refining one requirement should be given unique names. The path from the root of the requirements catalogue to a requirement may be used as a link to the requirement from the outside. For instance, the unique *link* to *EBADF* requirement is */Requirements/SELECT/ERRORS/EBADF*.

The structure of the API documentation usually determines the structure of the requirements catalogue up to a certain level. For instance:

1) The root requirement can be named after the computer program name. The corresponding requirement formulates the main task, goal or mission of the program.
2) The subsystems or main components of the computer program can reside on the first level of the requirements catalogue. The corresponding requirements formulate the main task or goal of the subsystem or component.
3) Classes and global functions can reside on the second level. The corresponding requirements formulate the main task or goal of a class or function.
4) Methods of classes reside on the third level. Functional requirements for the global functions also start to appear on the third level.
5) Functional requirements for the class methods start to appear on the fourth level.

The text describing functions usually structured in some way. There are two reasons:

- it helps not to lose an important section of information;
- the API documentation is usually generated by a template.

For instance, description of *select* function in POSIX [4] contains the following structure of headings: *NAME, SYNOPSIS, DESCRIPTION, RETURN VALUE, ERRORS*. Sections *NAME* and *SYNOPSIS* do not contain functional requirements. Headings *RETURN VALUE* and *ERRORS* become child nodes (requirements) of requirement *SELECT*. In the next section, we explain how to formulate requirements *RETURN VALUE* and *ERRORS* correctly.

C. *Requirements text structure*

Each requirement should have a textual description:

- a non-empty set of text fragments from the documentation;
- or a manually written text by a template;

Text fragment is a continuous word sequence in a document. Using Requality one can mark text fragments in a document and then link them to a requirement. It is possible to switch between text fragments and corresponding requirements with a single mouse click. There is, for example, the following sentence *If function select ends by time limit then return 0* in the description of function *select*. This text fragment is assigned to the requirement `BY_TIMEOUT_RETURNED_ZERO` and tends to describe it properly. Therefore in this case an additional manual description of the requirement is not mandatory.

In Requality you can supplement requirements by attributes. A manual description is an example of such an attribute. We write manual descriptions for functional requirements according to the following template [7]:

The function/subsystem/system MUST [actions set], [[if CONDITION]], [while CONDITION], [until CONDITION]. An action can be either:

- a modification of the internal state of the computer program;
- or return of a specific value from the function;
- or an exception.

A condition can be a first-order logic relation where predicates are written in a natural language and traditional conjunction, disjunction and negation are used to form more complex logical expressions. A condition is skipped if actions are unconditional. If the subject of the actions is obvious then the phrase *Function/subsystem/system MUST* may also be omitted.

If a set of text fragments from the API documentation is unambiguously argued as a single requirement by stakeholders and this set of text fragments satisfies `NEW_REQ_VERIFICATION` conditions, then this requirement may have no manual description. All requirements without text fragments from the API documentation are considered to come from other sources like an interview with developers or as a result of the analysis.

Let us look at an example. We assign the root requirement *SELECT* to a page header with text *select*. We found it useful for understanding to clarify this text fragment by defining the general purpose of *select* function: Function MUST wait UNTIL one of the provided file descriptors is ready to perform an operation.

Here is another example. A manual description for sub-requirement *ERRORS* can be as follows: Function MUST write the unique error code into variable *errno* if an error happened.

Sometimes it is more precise and simpler for understanding to represent requirements in the form of tables, images and models. Requality supports tables and arbitrary images in manual descriptions of requirements.

Every text fragment may be linked to one and only one requirement. All text fragments should be linked to requirements. This is the necessary condition to finish the requirements elicitation process. This condition contributes to the completeness of the requirements set.

D. Heuristic technique

We formulate a set of heuristics in this section that make a requirements elicitation technique effective. We tried to implement them in our technique.

1) Attempts to achieve the best possible requirements quality characteristics are often irrational. The quality of requirements should be enough to achieve the planned testing quality defined in the completeness criterion. The API documentation is not supposed to contain a complete set of functional requirements. Therefore the authors of the documentation, designers, developers and test engineers become an important source of functional requirements. They help to elicit and improve the requirements catalog [8].
2) The use of special requirements management tools like Requality greatly facilitates and simplifies the requirements elicitation process and the maintenance of large requirements catalogs.
3) Improvement of the API documentation is required to improve the quality of the requirements extracted from that documentation.
4) Representation of some requirements in a more suitable form rather than textual one and visual modeling of unclear requirements help to improve the requirements quality [7]. Effective non-textual requirements representations include tables and formulas. Data flow diagrams [9], UML action diagrams, UML state diagrams, decision tables [10] and other models [11] can be used to model different aspects of requirements.
5) A group work on a complex problem like requirements elicitation is efficient since it implies mutual assistance and participation of engineers having complementary qualifications. However an excessive team may have a negative impact on the efficiency of requirements elicitation.
6) The use of a task management system, bug tracking system and a version control system helps to support controllable, goal-oriented, responsible interaction of team members and to meet project deadlines.

E. Roles of participants

The process of requirements elicitation is a process of organized and controlled interaction of participants:

1) Requirement analyst (analyst):
   - helps technical project manager to assign priorities to functions;
   - extracts functional requirements from the API documentation;
   - supplements the requirements catalog with requirements from other sources;
   - reveals issues in the requirements catalog and fixes them by himself;
   - reveals issues in the API documentation by himself and makes the author of the API documentation responsible for correcting those issues;
   - manages and actively participates in requirements verification procedures.
2) Author of the API documentation:
   - creates the API documentation;
   - reveals issues in the API documentation by himself;
   - fixes issues in the API documentation;
   - as an important source of functional requirements provides them to the analyst;
   - participates in requirements verification procedures.
3) Test engineer:
   - formulates a test completeness criterion;
   - traces tests to requirements;
   - as an important source of functional requirements provides them to the analyst;
   - reports issues found in the requirements catalogue during the testing process and makes the analyst responsible for correcting those issues;

- helps technical project manager to assign priorities to functions;
- participates in requirements verification procedures.

4) Developer of the computer program (developer):
   - as an important source of functional requirements provides them to the analyst;
   - answers to the questions about the implementation of the computer program;
   - helps technical project manager to assign priorities to functions;
   - participates in requirements verification procedures.
5) Technical project manager (manager):
   - ensures the requirements elicitation process to meet its deadline;
   - assigns priorities to functions;
   - organizes productive interaction of the team members.

The analyst or the author of the documentation creates a task in a bug tracking system for every issue found in the API documentation. The author of the API documentation becomes responsible for fixing the issues. The developer helps to resolve the issues. The bug tracker records all stages of the issue life-cycle from assigning a responsible person to confirmation that the error has been fixed in a new documentation version.

Likewise the test engineer reports all issues found in the requirements catalogue into the bug tracking system and assigns the analyst to be responsible for fixing those issues.

The authors of the API documentation use a version control system to jointly work on the API documentation. A new release of the API documentation is usually issued along with a new version of the computer program.

Likewise analysts use the version control system to jointly work on the requirements catalogue. A new release of the requirements catalogue is usually issued before transferring the requirements catalogue onto a new version of the API documentation.

The manager creates a central repository where he saves the project technical requirement, contacts of the participants, information about the software used in the project, links to valuable resources, local project documentation and other. The manager creates the following tasks in a task management system:

- to mark up documentation for certain functions according to their priorities;
- to test functions for which requirements are ready.

The analyst creates the following tasks in the task management system:

- to verify the requirements extracted for certain functions;
- to supplement the requirements extracted for certain functions (meetings, consultations, interviews, questionnaires).

F. Requirements elicitation process

The requirements elicitation process begins from a preparatory stage.

1) Preparatory stage: In the first step of the preparatory phase, the analyst structures functions provided for testing and requirements analysis. Functions are usually divided into subsets that correspond to individual subsystems. Subsets can also form functions performing related operations: send and receive a message, write and read some data, set and unset an attribute.

In the second step of the preparatory stage, the analyst builds a list of pages in the API documentation that may contain functional requirements for each interface function. In practice, interface functions are described in a single and separate page in the API documentation.

In the third step of the preparatory phase, the manager assigns the priorities to the interface functions. The analyst, the tester and the developer are also actively involved in the prioritization process. Priorities can be assigned to subsets of functions rather than to individual functions.

The requirements elicitation process consists of many tasks that are performed by the participants. Every task is linked to one interface function or to a number of related interface functions. The function's priority determines the task's priority. As a result, functions with higher priorities are handled first.

Prioritization depends on different factors: available human and time resources, criticality of subsystems or functions, complexity of subsystems or functions, the size of the documentation text describing a function, restrictions imposed by the project technical requirement document. Priorities may be changed during the requirements elicitation process.

The API documentation is the foundation on which the requirements catalogue is built. If the quality of the API documentation is low, then the analyst reveals systemic problems in the API documentation in the fourth stage of the preparatory stage. The problem is considered to be systemic if it affects a significant number of functions or functions having high priorities. The person responsible for fixing these problems is the author of the API documentation.

2) Requirements markup: The analyst marks up requirements in the documentation according to the priorities of the interface functions. Let us consider some challenges in the requirements markup process.

Is there a functional requirement in a text fragment or a set of text fragments? Some researchers suggest to focus on the keywords that may indicate the presence of a requirement in a sentence [13]. In this case the text of the API documentation should be written according to some syntax rules. For example, the requirements in POSIX [4] standard can be identified by the verb *must*: Upon successful completion of the function, pselect () and select () must return the total number of bits specified in bitmasks.

In general, a requirement can not be recognized in a given set of text fragments on the basis of syntax rules. A functional requirement is a statement about what the computer program, a subsystem or a function should do under a condition or unconditionally. The object of actions in a software system is data: a return value of a function, the internal state of the computer program, an exception, a message.

Let us assume that a set of text fragments formulates one or more requirements if:

- an action is performed or a number of actions;
- the subject of the action is the computer program, a subsystem or a function;
- the object of the action is some data;
- the action should be performed under a condition or unconditionally.

How a set of text fragments should be handled which can not be interpreted as a requirement? To be able to control the completeness of documentation markup, it is necessary to separate this set of text fragments from marked and unmarked text fragments. Requality tool supports nodes of type *text node*. All text fragments that does not contain any requirements are assigned to nodes of this kind.

How a single text fragment should be handled which describes a behavioral aspect of several functions? For example, this is the case for *select* and *pselect* functions in the POSIX [4] standard. Separate requirement sets are necessary to be able to trace tests checking two different functions.

Requirements management tools do not usually allow to assign one text fragment to several requirements. Requality allows to copy and paste sub-trees of the requirements catalogue. The copies of the requirements duplicate the original requirements except that the links from the requirements to the text fragments are not copied, i.e. the text fragments remain associated with the source sub-tree only.

How should the analyst resolve a nontrivial issue found in the API documentation? Some examples of nontrivial issues are the following: unclear meaning of a certain text fragment, a contradictory statement, an ambigous statement. In this case, the analyst creates a task in a bug tracking system, makes the author of the API documentation responsible for it to be resolved and sets the priority for the task in accordance with the priority of the function.

The analyst often understands how a particular issue in the API documentation can be fixed. For example, it can be a duplicate information. **Should the analyst refine the text fragments manually in the requirements catalog or should a task be created in a bug tracking system and assigned to the author of the API documentation to fix the issue?** The analyst is better to refine the text fragments manually in the requirements catalog. Then the analyst can create a low-priority task in a bug tracking system in order this issue to be fixed in the API documentation.

The context of text fragments can affect their meaning. **Should text fragments be manually refined in the requirements catalogue or should the context be always taken into account?** In general, we recommend to clarify context-dependent text fragments in the requirements catalog since the size of the context is not always bounded with a single paragraph and the context itself may be ambigous.

3) Suspension criterion for the requirements markup of a function: The requirements markup process for a function is gradually approaching a state when further progress is either limited or difficult. The main natural reason for this is the limited function description in the API documentation. In addition, a large number of issues may slow down the requirements markup process for a function.

Criterion IV.1 (Suspension criterion for the requirements markup of a function)**.** *For every text fragment in the function description holds:*

- *either the text fragment is marked up, i.e. referred to a requirement or a text node;*
- *or a task has been created in a bug tracking system concearning the text fragment.*

The criterion can be more complex. Some factors that can be used in the suspension criterion for the requirements markup of a function are the following:

- the number of issues revealed in the function description;
- the amount of marked-up text;
- the amount of time spent on elicitation of requirements for the function;
- the complexity of the function defined by the developer.

At the beginning of the next markup iteration, it is useful to schedule some time to self-testing the existing markup of the function. Breaks help to take a fresh look at the API documentation and the existing requirements.

4) Transfer of requirements catalog onto a new version of API documentation: The transfer of requirements to the new documentation version should be automated, as this is a routine and time-consuming process. We use the Requality tool for that purpose. One can select two document versions and start the transferring process. After the process has been finished, the tool shows a view where one can look over the transfer status for all text fragments in the document. There are several possible statuses:

- red: text fragments that have not been found in the new version of the documentation;
- yellow: text fragments for which some text has lost in the new version of the documentation;
- green: text fragments that have been completely transferred to the new version of the documentation.

Some of the transfer statuses require additional manual actions. Status "red" assigned to a text fragment requires a new text fragment to be selected in the new version of the documentation. As an alternative, one can confirm that the text fragment is no longer present in the new version of the documentation. Similarly, one should confirm that a yellow text fragment has really been changed or should reselect it in the new version of the documentation.

5) Verification of requirements catalog for a function: When the function description in the documentation has been completely marked up, it is necessary to decide whether the requirements are ready for testing or the catalog of requirements for the function should be verified first. A high complexity of a function is the key factor in favor of verification. The following values can be used to estimate the complexity of a function:

- the time spent to build the requirements catalogue for the function;
- the number of issues in a bug tracking system related to the function;
- the size of the function description in the API documentation;
- the number of completed iterations of the requirements elicitation process;
- an assessment of the function's developer.

There are several requirements verification methods. One of the most effective is the formal inspection [14]. A list of issues is formed as the main result of a verification procedure. All issues should be written into a bug tracking system. If the number of issuies is high, an additional verification procedure may be planned.

We recommend to verify requirements according to a preliminary created plan. Such a plan should contain:

- the object of verification, i.e. the requirements for a function or a set of functions;
- the place, the date, the start and the finish times;
- a list of participants and their tasks;
- the subject of verification, i.e. the requirements properties to be verified such as uniqueness, completeness, consistency;
- verification methods like formal inspection [14], equivalence partitioning [15], boundary value analysis [16], decision table analysis [10], meetings [12], consultation, interview [17], questionnaires [17] [18] and others.

6) Necessary conditions to stop the requirements elicitation process for a function: The complexity of a function usually allows a trained analyst to realize its behavior at a certain degree of detail. This understanding of how the function works is essentially an informal behavioral model of the function. The analyst should not have any doubts about the correctness of this behavioral model and there should not be any tasks in the bug tracking system dealing with this function respectively. In addition, in order all participants of the requirements elicitation process to have the same or similar behavioral models of the function, the requirements catalogue for the function should be jointly verified.

Proposition IV.1 (Necessary conditions to stop the requirements elicitation process for a function)**.** *After stopping the requirements elicitation process for a function, the following conditions should hold:*

- *the markup of the function description in the API documentation should have been completed;*
- *all tasks in the bug tracking system dedicated to correction of the function description in the API documentation should have been completed;*
- *all tasks in the bug tracking system dedicated to correction of the requirements catalogue for the function should have been completed;*
- *all tasks in the task managment system dedicated to supplementation of the requirements catalogue for the function from other sources should have been completed;*
- *all requirements verification tasks for the function in the task management system should have been completed and the subject of the verification included:*
  - *unambiguity of the requirements;*
  - *the accuracy of the leaf requirements in the requirements catalogue;*
  - *consistency of the set of requirements;*
  - *completeness of the set of requirements.*
- *the function has been tested and all found erros have been fixed.*

7) Feedback from functional testing: API testing is always automated. Test programs are to be developed. The checks in these programs should be written in accordance with the functional requirements. Tracing establishes links between tests and requirements being verified. Thus, testing allows us to naturally confirm such an important characteristic of requirements as verifiability.

The test engineer can discover a number of issues in the requirements during the process of requirements-based test design. In addition, inconsistencies between the requirements and the observed behavior of the computer program can be found during testing. Some of these inconsistencies may be due to issues in the requirements. The test engineer should create a task for all found issues in a bug tracking system. The analyst should be made responsible to fix the issues. The priority for a task is set in accordance with the priority of the function in which the issue has been found.

The process of testing a function improves understanding of the behavior of the function. The test engineer becomes an important source of functional requirements for the function. Therefore the analyst should make the test engineer to be a participant of requirements verification and supplementation procedures. This will help to ensure the completeness of the requirements catalogue.

8) Necessary conditions to stop the elicitation process:

Proposition IV.2 (Necessary conditions to stop the elicitation process)**.** *After stopping the requirements elicitation process, the following conditions should hold:*

- *the markup of the API documentation or its target part should have been completed;*
- *all tasks in the bug tracking system dedicated to correction of the API documentation should have been completed;*
- *all tasks in the bug tracking system dedicated to correction of the requirements catalogue should have been completed;*
- *all tasks in the task managment system dedicated to supplementation of the requirements catalogue from other sources should have been completed;*
- *all requirements verification tasks in the task management system should have been completed;*

- *all target API functions should have been tested and all found errors have been fixed.*

V. Aprobation

Our technique for functional requirements elicitation has evolved during a number of industrial projects. Those projects have been performed by the authors of this study and other researchers from the software engineering department of ISPRAS [19].

The finally formed techniqe has recently been applied to extract requirements for input-output multiplexing functions from POSIX [4] standard. These are the following functions:

- poll;
- select;
- pselect.

The above functions were implemented in a real time operating system. The API of the operating system was described in an API documentation. As a result of a multiiterative requirements elicitation process a requirements catalogue consisting of 317 functional requirements has been built. Dozens of erros were found in the API documentation:

- incompleteness of information;
- inaccuracy of statements;
- conflicts with POSIX standard.

Similar work has been performed for function *semReportStatus*. This function outputs information about semaphore objects opened in the operating system.

We have also used Requality to build requirement catalogues for some parts of the following standards and specifications: ARINC 653 [21], TTCN-3 interface specifications, several RFCs including RFC 826, RFC 760 and RFC 768.

VI. Related work

The subject area of this study is methods for requirements elicitation from texts written in a natural language. Researchers have being investigated this field for decades. One way or another, individual ideas or approaches expressed in this paper might already be published in books or applied in practice. However, we have failed to find any requirements elicitation methods characterized as follows:

- the method is aimed at a specific type of documentation, i.e. the user API documentation;
- the method uses feedback from functional testing to enhance the quality of requirements;
- the method is effective in practice due to the use of a specialized software tools like Requality requirements management tool.

Our research group has being developed Requality requirements management tool. We know for sure there are no publications concerning techniques for requirements elicitation on the basis of this software tool. We fill this gap by publishing this study.

There are several industrial requirements management tools alternative to Requality [22]. The main usage scenario of most of these software tools is development of requirements for a new software or hardware systems from scratch. Requirement management tools assist in building requirements catalogs and support relations between requirements originated at different levels of the software life cycle. These are:

- business requirements;
- system requirements;
- functional requirements.

Requirements management tools operate during the whole life cycle and propose many features including version control, analysis of relations, various reports and etc. Requlity has also been used to develop requirements for new software systems from scratch. In turn, other requirements management tools may also be used to mark up API documentation in order to elicit functional requirements from it.

NLP (natural language processing) methods [23] are widely used to extract various information like requirements from texts written in a natural language. NLP methods are usually well automated therefore they effectively analyze big text data.

Information about hierarchy of classes (resources) and methods of these classes is extracted from the API documentation in study [24]. Methods are divided into categories:

- create a resource;
- lock access to a resource;
- modify a resource;
- unlock access to a resource;
- delete a resource.

Hierarchy of classes is determined by inheritance of classes. Information is extracted with NLP methods. Then an automaton is created for every resource on the basis of the above information. The automaton is then used to reveal defects in the computer program. For instance, using a resource before creating it is a defect.

Our study and work [24] are similar in that they analyze the same object, i.e. the API documentation. The both studies have the same goal to improve the quality of computer programs. However the methods to reach this goal are different. We look for functional requirements. The authors of study [24] look for defects.

Study [24] illustrates abilities of automatic text analyzers with NLP methods. Skills of requirements analyzis are still required nowadays to build a relatively complete catalogue of functional requirements. As of our technique, it is labor intensive and difficult to scale.

NLP methods can be used to verify some characteristics of requirements. For instance, QuARS [25] software tool can reveal ambiguity of text and subjectivity of text (not a requirement but a personal opinion). LOLITA [26] software tool analyzes text and incorporates it in a semantic net [27]. Then possible text interpretations are looked up.

Complex lexis, syntax and morphology of some natural languages and many exceptions from language rules make it more difficult to apply NLP methods. From one hand, our technique does not have these restrictions. From the other hand, all these problems become responsibilities of the analyst.

Application of several requirements elicitation techniques gives synergistic effect. The choice of a complementary technique depends on human resources available, i.e. the number of engineers, their experience, qualification and etc. Techniques effectively complementing our requirements elicitation technique do not strictly relate to mark up of texts. Among them are the following methods:

- formal inspection [14];
- equivalence partitioning [15];
- boundary value analysis [16];
- decision table analysis [10];
- meeting [12];
- consultation;
- interview [17];
- questionnairies [17] [18].

Correction of problems in the API documentation is an important part of our requirements elicitation technique. There are methods specially designed to reveal errors in the API documentation. For instance, texts written in a natural language are analyzed with NLP methods and source code snippets are analyzed by a code analyzer in study [28]. Combination of two types of analyzes allows for inconsistencies to be found between a text fragment and a source code snippet.

A method for requirements elicitation from the user documentation for a legacy system is proposed in study [29]. The requirements extracted are then used to create a functional specification document for a new system similar to the legacy system. A number of heuristics forms the basis of the method. Text structure, key words, lexical, syntactical and other text characteristics are used to extract key features of the system, functional requirements, use cases and nonfunctional requirements according to those heuristics.

VII. Conclusion

In this paper, we propose a technique for functional requirements elicitation from the user API documentation. We have elaborated our technique during a number of industrial projects. Using this technique a requirements analyst can create a catalogue of functional requirements suitable for functional testing. Markup of all documentation text or the target part of it is a necessary condition to build the complete set of requirements. An acceptable quality of functional requirements is obtained due to systematic verification procedures and feedback from functional testing.

The functional requirements catalogue is built in the form of a tree in which every requirement has a unique identifier. This tree structure assists in refinement of functional requirements. A requirement can be defined by selecting a set of text fragments from the API documentation or by manually writing a text by a template.

The requirements elicitation process proposed in this study is accompanied with correction of the API documentation. We use Requality requirements management tool to transfer the requirements catalogue onto a new corrected version of the API documentation.

Many problems appear during the requirements elicitation process. We have proposed solutions for some text mark up problems in this paper. We use third party methods like formal inspection [14], equivalence partitioning [15], boundary value analysis [16], decision table analysis [10], meeting [12], consultation, interview [17] and questionnairies [17] [18] to verify properties of requirements like unambiguity, completeness, consistency and accuracy.

References

[1] *DO-178C. Software Considerations in Airborne Systems and Equipment Certification*, RTCA SC-205 and EUROCAE WG-12 Std., 01 2012.

[2] V. Kuliamin, N. Pakulin, O. Petrenko, and A. Sortov, "Formalizacija trebovanij na praktike," *Preprint Instituta sistemnogo programmirovanija RAN*, vol. 13, 2006.

[3] *ISO/IEC/IEEE International Standard - Systems and software engineering – Life cycle processes – Requirements engineering*, ISO and IEC and IEEE Std., 11 2018.

[4] *Portable Operating System Interface*, ISO and IEC and JTC 1/SC 22 Std., Rev. 9945:2009, 09 2009.

[5] D. Kildishev and A. Khoroshilov, "Developing requirements management tool for safety-critical systems," in *Proceedings of 2019 Actual Problems of Systems and Software Engineering (APSSE)*, 2019. [Online]. Available: https://doi.org/10.1109/APSSE47353.2019.00013

[6] A. Khoroshilov and D. Kildishev, "Formalizing metamodel of requirements management system," *Proceedings of ISP RAS*, vol. 30, no. 5, pp. 163–176, 2018.

[7] K. Eugene-Wiegers and J. Beatty, *Software Requirements*. Microsoft Press, 2013.

[8] Z. Zhang, "Effective requirements development-a comparison of requirements elicitation techniques," in *Proceedings of Software Quality Management XV : software quality in the knowledge society*, 2007, pp. 225–240.

[9] T. DeMarco, *Pioneers and Their Contributions to Software Engineering*, M. Broy and E. Denert, Eds., 1979.

[10] R. Shiffman and R. Greenes, "Improving clinical guidelines with logic and decision-table techniques: Application to hepatitis immunization recommendations," *Medical Decision Making*, vol. 14, no. 3, 1994. [Online]. Available: https://doi.org/10.1177/0272989X9401400306

[11] J. Beatty and A. Chen, "Visual models for software requirements. edition: 1," 2012.

[12] R. Ocker, J. Fjermestad, S. Hiltz, and K. Johnson, "Effects of four modes of group communication on the outcomes of software requirements determination," *Journal of Management Information Systems*, vol. 15, 6 1998. [Online]. Available: https://doi.org/10.1080/07421222.1998.11518198

[13] N. Niu and S. Easterbrook, "Extracting and modeling product line functional requirements," 09 2008, pp. 155–164. [Online]. Available: https://doi.org/10.1109/RE.2008.49

[14] M. Fagan, "Design and code inspections to reduce errors in program development," *IBM Systems Journal*, vol. 15, no. 3, 1976. [Online]. Available: https://doi.org/10.1147/sj.153.0182

[15] D. Richardson and L. Clarke, "A partition analysis method to increase program reliability," in *Proceedings of the 5th International Conference on Software Engineering*, ser. ICSE '81. IEEE Press, 1981, p. 244–253. [Online]. Available: https://doi.org/10.5555/800078.802537

[16] S. Reid, "An empirical analysis of equivalence partitioning, boundary value analysis and random testing," in *Proceedings Fourth International Software Metrics Symposium*, 1997. [Online]. Available: https://doi.org/10.1109/METRIC.1997.637166

[17] S. Sharma and S. Pandey, "Article: Revisiting requirements elicitation techniques," *International Journal of Computer Applications*, vol. 75, no. 12, pp. 35–39, August 2013.

[18] Y. Theng, S. Foo, D. Goh, and J.-C. Na, "Handbook of research on digital libraries: Design, development, and impact," pp. 287–297, 01 2009. [Online]. Available: https://doi.org/10.4018/978-1-59904-879-6

[19] [Online]. Available: https://www.ispras.ru/

[20] A. Godunov and V. Soldatov, "Baget real-time operating system family (features, comparison, and future development)," *Programming and Computer Software*, vol. 40, no. 5, pp. 259–264, 2014. [Online]. Available: https://doi.org/10.1134/S036176881405003X

[21] S. Santos, J. Rufino, T. Schoofs, C. Tatibana, and J. Windsor, "A portable arinc 653 standard interface," in *Proceedings of the 2008 IEEE/AIAA 27th Digital Avionics Systems Conference*, 2008. [Online]. Available: https://doi.org/10.1109/DASC.2008.4702767

[22] N. Gorelits, D. Kildishev, and A. Khoroshilov, "Requirements management for safety-critical systems. overview of solutions," *Proceedings of ISP RAS*, vol. 31, no. 1, pp. 25–48, 2019. [Online]. Available: https://doi.org/10.15514/ISPRAS-2019-31(1)-2

[23] J. Eisenstein, *Introduction to Natural Language Processing*. Cambridge, Massachusetts, London, England: The MIT Press, 2019.

[24] H. Zhong, L. Zhang, T. Xie, and H. Mei, "Inferring specifications for resources from naturallanguage api documentation," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, 2009, pp. 307–318. [Online]. Available: https://doi.org/10.1109/ASE.2009.94

[25] G. Lami, "Quars: A tool for analyzing requirement," Software Engineering Institute, Tech. Rep., 2005. [Online]. Available: https://doi.org/10.1184/R1/6582770.v1

[26] L. Mich, "Nl-oops: From natural language to object oriented requirements using the natural language processing system lolita," *Natural Language Engineering*, vol. 2, no. 2, pp. 161–187, 1996. [Online]. Available: https://doi.org/10.1017/S1351324996001337

[27] L. Schubert, R. Goebel, and N. Cercone, *Associative Networks*. Academic Press, 1979, ch. The structure and organization of a semantic net for comprehension and inference. [Online]. Available: https://doi.org/10.1016/B978-0-12-256380-5.50010-4

[28] H. Zhong and Z. Su, "Detecting api documentation errors," in *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages and applications*, 2013, pp. 803–816.

[29] I. John and J. Dörr, "Elicitation of requirements from user documentation," in *Proceedings of the the Ninth International Workshop on Requirements Engineering: Foundation for Software Quality*, vol. 3, 2003.