On requirements completeness analysis method

Viktoria Gingina
Institute for System Programming
at the Russian Academy of Science
Moscow, Russian Federation
e-mail: vgingina@ispras.ru

Abstract — **Requirements figure prominently in information system development. Both development phase of the system and phase of its verification depends on how qualitative requirements are. That is why it's required to describe requirements as accurately and correctly as it possible. One of the properties that define quality of requirements collecting is completeness. The paper shows that if one obtains sources and symptoms of the requirements incompleteness (completeness absence), classifies, generalizes and clarifies them then one can check requirements for these symptoms while collecting requirements or after that. This will sufficiently decrease the incompleteness of the requirements and thus improve their quality. The paper contains some symptoms of incompleteness have already been revealed and explains the reason of their appearance. These symptoms have been revealed by analyzing the documentation of some important industrial projects.**

Requirements, completeness, incompleteness, incompleteness sources, incompleteness symptoms

I. Introduction

There is the following definition of the term “requirement” in IEEE Standard Glossary of Software Engineering Terminology (1990) [1]:

(1) A condition or capability needed by a user to solve a problem or achieve an objective.

(2) A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents.

(3) A documented representation of a condition or capability as in (1) or (2).

The process of developing any information system begins with the phase of requirements collecting. Requirements are used on numerous occasions while developing the system. That is why there should be high quality criteria for requirements.

Requirements are used when modeling the system. They informs about what the system should do, what resources it can rely on, what constraints it should conform to and so on. Any changes and additions to the being developed system architecture is a very hard laborious process especially for the large divaricated one. It causes excessive complication and probably transformation of the architecture in a whole. Thus risks quantity is increasing and developing time is rising. The problem is apparent especially when interconnecting several systems: incorrect requirements can cause their incompatibility.

Requirements are also used to verify if the developed system corresponds to what it’s expected to be. Check-on conformance with the requirements is used. If tests used for the verification are not qualitative the testing process will take longer than it was planned. To create good test testers will have to clarify information with the help of analysts and developers and besides the opinion of these experts could be different in points that are incorrectly described in the requirements. Thus the quantity of the failed tests will arise. The results are: low-quality product, broken date of performance, increased cost. And if there is lack of means system development could be paused.
When using low-quality documentation risk rises.
Thus good well-stated, i.e. high-quality requirements are needed. Such requirements should meet following criteria [2]:

Adequacy – requirements meet customer wishes.

Unambiguity – different domain experts understand requirements equally.

Consistency – requirements are exhaustively formulated

Completeness – any situation has its own description (usually general) in the requirements. This paper is devoted to completeness achievement.

II. Requirements completeness

Requirement completeness criterion can be separated into two constituent parts [2]. Complete requirements should describe:
- firstly everything that customer wants to get of the system;

- secondly system behavior in any logically possible situations.

The first aspect is a task for customer analyst. But in the second aspect completeness analysis should be carried out by the requirements completeness expert.
K. Wiegers said that completed requirements give many priorities [3]:

- restriction of rework and redesign when developing,
- project risk decrease,
- system interoperability rise,
- more complete requirements gives fewer mistakes are founded while testing (functional, integration, etc.)

The problem of requirements completeness is not new and is usually considered in respect to rules of good and accurate requirements collecting: ways to interview customer and concerned persons, assessment of user-groups needs, questions that analyst must answer when collecting requirements of different type. But even if all these recommendations are executed requirements often are not complete. Why so?

K. Wiegers notes [3]: “Many software problems arise from shortcomings in the ways that people gather, document, agree on, and modify the product’s requirements. <…> the problem areas might include informal information gathering, implied functionality, erroneous or uncommunicated assumptions, inadequately defined requirements, and a casual change process”.

There are different reasons of incompleteness. One of them is a human factor. Since requirements are collecting by a human they just can’t be fully considered. A commonplace example: there are no constraints for calculation in a computing system. A person who collects requirements (system analyst) believes that some situations are an axiom understandable for everybody so there is no need to describe it in detail – this is another often problem. “System behavior is obvious, usual and do not need any additional explanation” – that is why ambiguous and not evident moments are often not described in requirements. We can also single out moments of “premeditated incompleteness”. This is a situation when customers didn’t get an agreement or a situation when customer wants to give carte blanche to developers or a situation when system behavior is hard to foretell (impartial non-determinism). K. Wiegers said in his book [4]: "Requirements are never finished or complete. There is no way to know for certain that you haven't overlooked some requirement, and there will always be some requirements that the analyst won't feel it is necessary to record".

In spite of this it is feasible to decrease requirement incompleteness as far as it possible.

Obtaining above-mentioned incompleteness sources gives some understanding of how to find it in requirements. As a result of the research it’s planned to get some check-list consists of check-questions that help to define incompleteness. And it’s planned to get a set of patterns and anti-patterns that describe situations able to be incomplete. If there is “computes <…>” expression and there is no a precision of computation in requirements this is an example of anti-pattern in requirements description. Element of such lists and sets is not an exact indicator of incompleteness but it’s an indicator for a potential place of requirements where description of system property or function is able to be incomplete. Usage of such lists and sets will help to find vulnerable moments in requirements and to define possible but not described situations.

Thus there is a problem to define and to classify possible sources of requirements incompleteness.

Attempts to reveal requirements incompleteness has been already done by the other researches. So it was suggested to describe system behavior in all possible conditions and if there is requirements for situation “A” to define what is happening in all “not A” cases [5]. Also it was suggested to consider requirements in respect to actor (what actor is responsible for what function), to describe all alternative action flows and to justify any requirement [6]. Another method recommends checking if requirements are for all system elements [7]. From the point of view of incompleteness source obtain method such approaches are quite one-sided. Method suggested in this paper includes above-mentioned approaches but considers them as special cases of incompleteness that are not enough. Obtaining incompleteness sources allows to research the problem more profoundly and to discover more symptoms of incompleteness and thus to assess requirements completeness more correctly.

III. Requirements incompleteness sources

Technical documentation for a few industrial systems has been analyzed. This allowed to obtain some symptoms of requirements incompleteness. Considered systems are developed within the framework of large project. This fact vividly demonstrates that requirements incompleteness is critical for implementation and is usual even for quite a good documentation.

In our examples symptoms of incompleteness are expressed as anti-patterns. These are situations that are mostly frequent for the considered projects.

(1) There is a condition-element but not all flow-branches are described.

If there is a description for successful work of some function there should be a description for an erroneous situation. If there is description for “then” condition there should be a description for “else” (“otherwise”). Similarly if there is function description for a set of parameter values there should be a description for any other possible values. E.g. if it’s settled that there is some action flow for positive values of real-type parameter then it doesn’t mean that nothing happen for zero and negative values of this parameter. Probably author of the requirements has such a behavior in his mind but in that case he should explain and describe it clearly. Otherwise we have incompleteness in the requirements.

So in the LSB specification [8] there was not a description for g_date_clamp function [9] behavior in a case when “date” parameter value was in range between "min_date" and "max_date" parameters. In the same specification there was g_main_context_iteration function [10] description only for the situation when "may_block" parameter was "TRUE". Situation with "FALSE" value has been omitted.

Special cases of such incompleteness are function requirements that have no description for 0 or NULL values of function parameters. Also there should be clear description for float parameters in the case of Nan and Inf values.

For example in the documentation for a huge industry system S critical defects have been found: a behavior of system interface functions was not described for the case when these functions got faulty incoming data (0 or NULL) instead of file pointer. In SUS 3.0 [11] in the description of ualarm() function [12] there was not requirement for the function behavior when “useconds” parameter is 0.

(2) Changing the data is described in one action branch but there is no any description for the same data in the other branch.

Probably it doesn’t change but the absence of the clear description indicates incompleteness. Some parameter, object pointer, picture on a web-form, content of a file, everything function can affects on - that is what we consider as data in this situation. Values of all these elements form system state. Complete requirements should describe how every function influences to these data elements in all action flows (or there should be a clear instruction that nothing is changed). Furthermore it’s required to note an indirect influence of the sub-functions.

In the above-mentioned system S interface there is a parameter ERROR_ID that gets an identifier of the error took place when using the function. But earlier there were not instructions for correct function processing. An assumption that value was not changed founded to be wrong. In reality the parameter got a “noerror” value.

(3) A new function, type, object, term is used but never described.

For the first view such an omission can look absurd but it quite often occurs. In well-formulated requirements you can suddenly find a link to some function, parameter, data element that is described or explained nowhere in the documentation. For example in the LSB specification [13] in the svcudp_create() function [14] description it was noted that this function was called similarly to svcudp_bufcreate(sock, SZ, SZ) function call. But svcudp_bufcreate() function was never described in the specification. The reason of this incompleteness can be the uncoordinated documentation writing and changing. An ordinary misprint can take place too. And there is a possibility that the function just has been forgotten to describe. Obviously such an error is peculiar to divaricated systems because for a requirements writer it’s more difficult to imagine complex system in a whole. If there are more than one writer the problem will be interconnection between them.

IV. Sources of analyzed documentation

Data analyzed to find incompleteness symptoms is documentation on three industrial systems.

Examples of the defects found in Linux specifications requirements got from the official information of Verification Center of the Operating System Linux [15]. The Center is based at the Institute for System Programming of the Russian Academy of Sciences (ISP RAS) [16]. It integrates a group of projects of developing open source tests and automated verification techniques for Linux-systems. The Center is supported by the Russian Federal Agency [17] for Science and Innovations, by the international consortium The Linux Foundation [18].

Linux Verification Center works on checking that Linux implementations are conform to requirements and specifications. There is information of testing results and found inequalities on Center site. There are many problem reports marked as “incompleteness” among them. As LinuxTesting.org documentation is officially published on their Internet page there is a possibility to show incompleteness symptoms using some real examples of the project.

Other two projects are commercial, closed and do not publish such an information. Internal documents being used for analysis are test-cases and defects reports registered in bug-tracking system in one case and test report documents in the other case.

V. Conclusion

This paper is devoted to a necessity of qualitative description of the system requirements. The paper discloses an importance of the requirements incompleteness and shows its critical influence. Some incompleteness symptoms are obtained and obvious examples of incompleteness demonstration are provided for the huge industrial systems. Obtaining incompleteness symptoms allows improving the project documentation that will give an opportunity to avoid problems when implementing and testing system. It’s planned to continue documentation analysis for different projects to find and classify other incompleteness sources and to get check-lists, patterns and anti-patterns sets. Also it’s planned to examine possibility of incompleteness sources obtain by formalization of requirements collecting process, by requirement modeling, by specification of verification tests and by analyzing results of static and dynamic implementation analysis.

References

[1] http://standards.ieee.org/reading/ieee/std_public/description/se/610.12-1990_desc.html

[2] V.Kuliamin, N.Pakulin, O.Petrenko, A.Sortov, A.Khoroshilov, Requirements formalization on practice, Preprint 13, ISP RAS, Moscow, 2006 (in Russian)

[3] K.Wiegers, Software Requirements: Practical Techniques for Gathering and Managing Requirements Throughout the Product Development Cycle, 2nd edition, Microsoft Press, Redmond, Wash., 2003

[4] K. Wiegers, More About Software Requirements: Thorny Issues and Practical Advice, Microsoft Press, Redmond, Wash., 2006

[5] R. S. Carson, Requirements Completeness: A Deterministic Approach, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.8.735

[6] A. Sarkar, Requirement Management in Testing, Infosys Technologies Limited, Bangalore, unpublished

[7] S. Robertson, J. Robertson, "Mastering the Requirements Process Second Edition", Addison Wesley Professional, 2006

[8] Linux Standard Base Desktop Specification 3.1, Chapter 12. Libraries, 12.2 Interfaces for libglib-2.

[9] http://www.gtk.org/api/2.6/glib/glib-Date-and-Time-Functions.html#g-date-clamp

[10] http://www.gtk.org/api/2.6/glib/glib-The-Main-Event-Loop.html#g-main-context-iteration

[11] The Open Group Base Specifications Issue 6 IEEE Std 1003.1, 2004 Edition (SUS 3.0), System Interfaces

[12] http://www.opengroup.org/onlinepubs/000095399

[13] Linux Standard Base Core Specification 3.1, Chapter 13. Base Libraries, 13.5. Interface Definitions for libc

[14] http://refspecs.freestandards.org/LSB_3.1.0/LSB-Core-generic/LSB-Core-generic/baselib-svcudp-create-3.html

[15] http://linuxtesting.ru

[16] http://www.ispras.ru

[17] http://www.fasi.gov.ru

[18] http://www.linuxfoundation.org