NLG4RE: How NL Generation Can Support Validation in RE

Bert de Brock[2] and Coen Suurmond[2]

[1] *University of Groningen, PO Box 800, 9700 AV, Groningen, The Netherlands*
[2] *Cesuur BV, Kerkstraat 7, 6883 HP, Velp, The Netherlands*

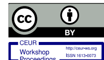
Abstract

Context and motivation: All too frequently functional requirements (FRs) for a (software) system are unclear. Written in natural language, FRs are underspecified for software developers; when written in formal language, FRs are insufficiently comprehensible for users. This is a well-known problem in RE. As long as this either/or dichotomy exists, FRs cannot be a "basis for common agreement among all parties involved", as Barry Boehm puts it.
Question/problem: On the one hand, FRs should unambiguously specify the functional behaviour of the system to be written or adapted, and on the other hand be fully understandable by the customer that must agree with them. What is required to achieve this goal?
Principal ideas/results: A specification must describe the Statics as well as the Dynamics. In our approach it consists of a Conceptual Data Model (the data structure, i.e., the Statics) plus a set of System Sequence Descriptions (SSDs) representing the processes (i.e., the Dynamics). SSDs schematically depict the interactions between the primary actor (user), the system (as a black box), and other actors (if any), including the messages between them.
We provide a set of rules to generate natural language expressions from both the Conceptual Data Model and the SSDs that are understandable by the user ('Informalisation of formal requirements'). Generating understandable representations of a specification is relevant for requirements validation tasks.
Contribution to validation: We introduce a form of Natural Language Generation (the NLG in the title) by defining a grammar and mapping rules to precise and unambiguous expressions in natural language, in order to improve understandability of the FRs and the data model by the user community.
Keywords
Functional Requirement, Explainability, Conceptual Data Model, System Sequence Description, Use Case, Grammar, Syntax-directed Mapping, Validation

CEUR-WS.org/Vol-3122/NLP4RE-paper-2.pdf

1. Introduction

In his seminal article [1], Barry Boehm defined software requirements engineering as "the discipline for developing a complete, consistent, unambiguous specification - which can serve as a basis for common agreement among all parties concerned - describing *what* the software product will do (but not *how* it will do it; this is to be done in the design specification)". For our paper, the three most relevant phrases of the definition are underlined. Firstly, the definition entails that unambiguous software development requires a specification of the behaviour of the software that must be as precise as the software itself, hence expressed in a formally defined language. (Please note: not all specification languages are formally defined.) Secondly, requirements engineering is about the *what* (the functional behaviour) and not about the *how*. Thirdly, the definition implies that the requirements must be fully

In: J. Fischbach, N. Condori-Fernández, J. Doerr, M. Ruiz, J.-P. Steghöfer, L. Pasquale, A. Zisman, R. Guizzardi, J. Horkoff, A. Perini, A. Susi, M. Daneva, A. Herrmann, K. Schneider, P. Mennig, F. Dalpiaz, D. Dell'Anna, S. Kopczyńska, L. Montgomery, A. G. Darby, and P. Sawyer (eds.): Joint Proceedings of REFSQ-2022 Workshops, Doctoral Symposium, and Poster & Tools Track, Birmingham, UK, 21-03-2022.
EMAIL: e.o.de.brock@rug.nl (B. De Brock); coen@cesuur.info (C. Suurmond)
ORCID: 0000-0003-4400-0187 (B. De Brock); 0000-0001-7229-3541 (C. Suurmond)

© 2022 Copyright for this paper by its authors.
Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
CEUR Workshop Proceedings (CEUR-WS.org)

understandable for all parties concerned. For software development, the definition implies that for a given specification different software designs and different implementations of those designs are not allowed to have any impact whatsoever on the functional behaviour of the software. For representatives of the user community (in whatever stakeholder role) the definition implies that they can fully check and validate the behaviour of the software-to-be in their user world. Hence our starting point is that (1) Functional Requirements (FRs) must be expressed in a formal language and (2) must be understandable in the user community. But (and this is a major but!), in many cases domain experts from the user community will have difficulties in reading specifications formulated in a specialized formal language. This will hamper the understanding that is required for the common agreement about the specifications. Hence, our paper will address the issue how formal specifications can be made understandable for the user community, as a condition sine qua non for a proper validation by the user community.

Generally speaking, various approaches are available that map Use Case descriptions to UML diagrams [2], goal modelling [3], BPMN [4], Petri nets [5], or other (semi-formal) concepts for evaluation. Although those concepts and notations might be suitable for validation by our *colleagues*, they are usually not really suitable for validation by *end users*. The reason is that those users might be experts in their own domain, but not in fully understanding (the consequences of) what is expressed in those notations. And even amongst colleagues such visuals models can be incomprehensible.

In [6], Allweyer analysed a small example model provided by the Object Management Group: "The process as such may indeed be small, but the diagram is not small … It is understandable for experts – but it is certainly not easy to understand. I would not dare to give this diagram to any business expert". We see such problems in practice again and again. Other issues with such approaches are a lack of formal definition of the specification language and the embedding of natural language descriptions of real world processes in process specifications. Leopold et al. [7] write "Due to the extensive symbol set, a complete formalization of BPMN would introduce unnecessary complexity". Our objective is to avoid such problems by (1) using a formally defined specification language, in combination with (2) a small set of mapping rules from expressions in the formal language to natural language expressions that are understandable in the user community (building on our earlier work presented in [8,9]).

Essentially, the results of an RE-phase for a system to be developed consist of a description of the *statics*, describing the relevant *data* (structures), and of the *dynamics*, describing the relevant *processes*. The *statics* are often given in the form of a *Conceptual Data Model*. We propose to give the *dynamics* in the form of *textual System Sequence Descriptions* (*tSSDs*). Together, the statics and the dynamics constitute a complete conceptual 'blue print' of the system to be developed, as summarized in Table 1:

Table 1. Overview of concepts

Aspect	Problem Analysis result
Statics / Data structure	Conceptual Data Model
Dynamics / Processes	Textual SSDs

Because Conceptual Data Models (CDMs) and Textual SSDs can be specified in a structured way, we were able to give complete mapping rules how, in a systematic way, CDMs and Textual SSDs can be mapped to (and therefore explained in) unambiguous expressions in natural language, say in English. Depicted schematically:

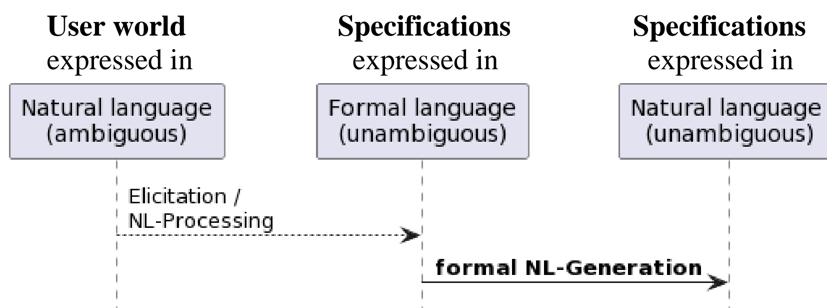


Figure 1: Generating unambiguous natural language from unambiguous formal language

We emphasize here that the question of the explanation of formally specified FRs to the user can be considered independent of the way the FRs were developed. Whether the requirements were developed linear, incremental, agile, or elsewise is immaterial for our purposes. And although we use one specific specification language, our approach is applicable to other formally and completely defined specification languages as well. We will limit ourselves to *functional* requirements which, of course, is not to deny the relevance of non-functional requirements!

The rest of the paper is organized as follows. Section 2 considers related work. Section 3 zooms in on the possible meanings of *explainability*. Section 5 introduces the notion of *textual SSDs*. Sections 4 and 6 give mapping methods to explain Conceptual Data Models and textual SSDs (respectively) in natural language to the user community. Section 7 zooms in on our early validation of the approach. Section 8 contains a discussion and Section 9 presents some conclusions.

2. Related work

Kossack et al. address our question in their paper "Improving the Understandability of Formal Specifications" [10]. The authors notice that reviewing a specification "relies on the assumption that the readers must be able to build a consistent and complete mental image of the model that is sufficiently precise to assess its correctness … Graphical notations, such as UML, are quite effective. Unfortunately, they lack the precise mathematical basis that is required to express and assert critical properties. On the other hand, mathematical-logical bases formalisms … are more appropriate for the latter purpose, however, they are difficult to understand". We completely agree with this dichotomy, and also with their (readability) guideline to at least carefully choose the name of variables. However welcome such improvements are, we do not think that such an approach will solve the problem of understandability of specifications in a formal language by domain experts.

Wiegers [11] has written that "a formal inspection of the software requirement specifications by project stakeholders who represent different perspectives is one way to determine whether each requirement has these desired attributes". However, his quality characteristics do not mention "understandable". While he observes that "only user representatives can determine the correctness of user requirements", he does not discuss how users can be expected to do this with formal specifications.

"A systematic literature review of use case research" by Tiwari and Gupta [12] discusses almost 120 papers. They notice that in writing use cases "their inherent utilization of natural language and a behaviour of requirement documentation in a semi-conventional way, mean that they are affected by issues such as ambiguity, redundancy, inconsistency and incompleteness. Several efforts have been made by researchers in order to address these issues by formalizing both behavioural and structural aspects of the use case specifications". However, the research is mainly about the production of formalized use cases and not about the issue how to validate a formalized use case by the average user. The authors found that "the most used inspection technique for use case validation is the checklist", which is a rather indeterminate way of reviewing a formal specification.

The systematic literature review by Yaman et al. [13] discusses 25 papers and mentions in its abstract that "In order to build successful software products and services, customer involvement and an understanding of customers' requirements and behaviours during the development process are essential", followed by their question/problem section which mentions learning **from** customers, but not learning **by** customers. It is not about a systematic validation of requirements either. The involvement of customers is about giving feedback about their experiences in using the software, not about a direct systematic validation of the requirements by the customer.

The general impression is that the focus of the papers on requirements engineering seems to be on the process, the model, the modeller, the tools, and the software quality. However, the role of the user in the validation of the requirements gets little attention. The user seems to be reduced to a source whose needs are to be elicited by the requirements engineer, rather than as an addressee of the resulting requirements. This is corroborated by a scan of the tables of contents of the REFSQ conferences from 2007 – 2021, where we observe that very few paper titles or abstracts indicate research on the issue how to present formally specified requirements back to the user community in such a way that genuine understanding and validation can be achieved.

3. Explainability explored

A basic question in exploring the concept of 'explaining' is: *Who* explains *what* to *whom*? Therefore, we clearly distinguish four different roles in software (SW) development involved: *User*, *Requirements Engineer*, *Software Designer*, and *Programmer*. This identifies the possible "who" and "whom" in explanations. The "what" concerns (1) user wishes / user world, (2) functional requirements specification, (3) software design, (4) software. This results in the diagram in Figure 2, showing interactions where an explanation could be given. The quoted parts below refer to the Oxford English Dictionary lemma 'explain' [14]:

1. The users make their world & wishes "plain or intelligible" to the requirements engineer
2. The requirements engineer will "state the meaning or import of" their FRs to the user
3. The formal FRs the requirements engineer gives to the SW designer should be self-explaining; any necessary explanation would reveal a shortcoming in the FRs
4. The SW designer does "make clear the cause, origin, or reason of" the SW design in relation to the FRs (i.e.: the designer shows how the design fulfils the requirements)
5. Similar to 3: The design should be self-explaining
6. Similar to 4: The programmer makes clear how the software fulfils the design

Our paper focuses on Arrow 2: The explanation of the FRs back to the user.

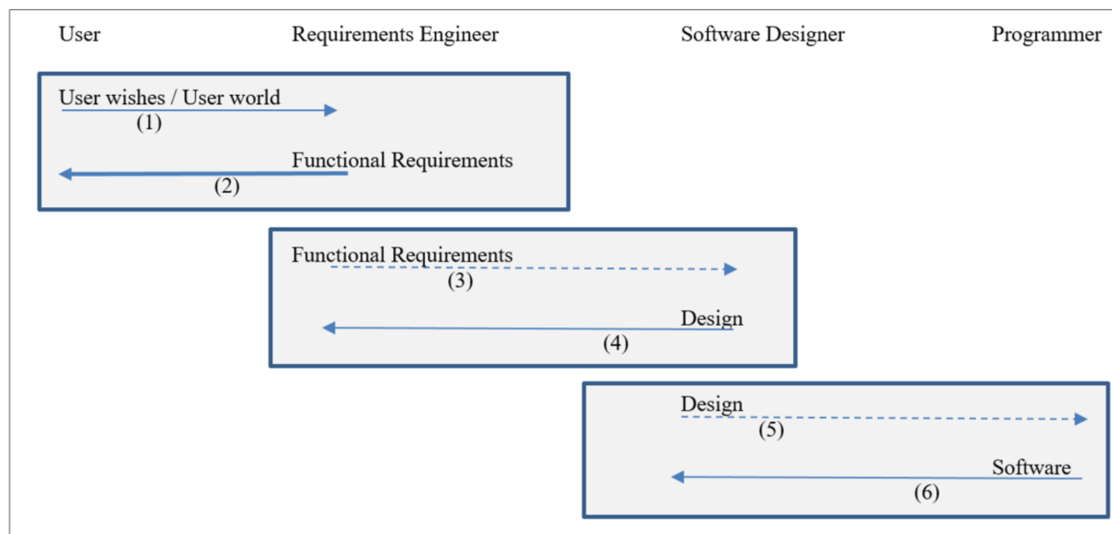


Figure 2: *Who* explains *what* to *whom*?

4. Explainability of a given conceptual data model

When defining a CDM, we use the notions of <u>concept</u> and <u>property</u> of a concept. A CDM usually consists of a set of concepts where each concept has a set of properties. A value for a property P might be <u>optional</u>, which we indicate by '[P]'. A property can be a <u>Yes/No-property</u> (a 'Boolean'), in which case we use the form 'P?'. A property P can refer to another instance of a concept, which we indicate by '^P' (where P is usually the name of the referenced concept). A property or combination of properties of a concept might be <u>uniquely identifying</u> within that concept (a.k.a. a 'key'). If a CDM consists of concept C_1 with properties $P_{1,1}$, …, $P_{1,n1}$, concept C_2 with properties $P_{2,1}$, …, $P_{2,n2}$ (etc.), until concept C_m with properties $P_{m,1}$, …, $P_{m,nm}$, then our 'explanation' runs as follows:

The system needs to contain:

- **For each relevant** C_1**: its** $P_{1,1}$ **, …, and its** $P_{1,n1}$**.**
- **For each relevant** C_2**: its** $P_{2,1}$ **, …, and its** $P_{2,n2}$**.**

⋮

- **For each relevant** C_m**: its** $P_{m,1}$ **, …, and its** $P_{m,nm}$**.**

We added the word ‘relevant’ because not each individual C might be relevant for the organization. Furthermore, the following subsequent adaptions apply:

1. If a value for a property P is optional then we replace ‘**its** P’ by ‘**optionally its** P’.
2. If a property refers to another instance of a concept (so, is of the form ‘^P’) then we replace ‘**its** ^P’ by ‘**a reference to its** P’.
   (Hence, in case of an optional reference we get ‘**optionally a reference to its** P’.)
3. If a property is a Yes/No-property (so ‘P?’) and P is a *verb phrase* then we use ‘**whether it is** P’ instead of ‘**its** P?’; if P is a *noun phrase* then we use ‘**whether it is a(n)** P’.
4. If a concept C represents a human being then we use ‘**his/her**’ instead of ‘**its**’ and ‘**whether he/she is** [**a(n)**] P’ instead of ‘**whether it is** [**a(n)**] P’.

For each property (combination) P_1, …, P_k of a concept C which is uniquely identifying within that concept, we can add the sentence

‘**The same value** [**combination**] **for** P_1**, …, and** P_k **should not occur twice.**’

immediately after the complete sentence ‘**For each relevant** C**: … .**’. The word ‘combination’ can be left out if k = 1 (so if one property in itself is uniquely identifying).

Alternatively, we could add all the uniqueness conditions under one separate heading, say ‘**Conditions**’, but in that case we must mention the concept it applies to. We could then use a sentence like ‘**There can be no two different** Cs **with the same value** [**combination**] **for** P_1**, …, and** P_k**.**’, as in our next example:

Example 1: Explanation of a conceptual data model

We give an example of a conceptual data model with one human being concept (Student) and two non-human being concepts (Course and Course Enrolment), with some Yes/No-properties, some optional properties and some combinations thereof. Furthermore there is one uniqueness condition consisting of one property (Student number), and twice a uniqueness condition with two properties (e.g., Faculty, Course code); with the uniqueness property (combination) underlined:

Student: Student number, Name, [Phone number,] [Freshman?,] Birth date
Course: Faculty, Course code, Name, [Master course?,] Description
Course Enrolment: ^Student, ^Course, Accepted?

The explanation of this conceptual data model applying our ‘explanation rules’ would result in:

The system needs to contain:

- **For each relevant** Student**: his/her** Student number**, his/her** Name**, optionally his/her** Phone number**, optionally whether he/she is a** Freshman**, and his/her** Birth date**.**
- **For each relevant** Course**: its** Faculty**, its** Course code**, its** Name**, optionally whether it is a** Master course**, and its** Description**.**
- **For each relevant** Course Enrolment**: a reference to its** Student**, a reference to its** Course**, and whether it is** Accepted**.**

Conditions:

- **There can be no two different** Students **with the same value for** Student number**.**
- **There can be no two different** Courses **with the same value combination for** Faculty **and** Course code**.**
- **There can be no two different** Course Enrolments **with the same value combination for** Student **and** Course**.**

Preceded by a short and simple legend explaining notions like *optionality* and *uniqueness conditions*, it should be well understandable by the customer. In principle, the users should be able to understand and confirm this, or correct this in case of incompleteness, ambiguity, or defect detection in the specifications/requirements.

Note that the enumerations do not have limitations on the number of concepts, properties, or relationships. Hence, we can also handle ‘large’ conceptual data models, i.e., models with a large number of concepts, properties, and/or relationships. In that case, we get long enumerations of (simple) statements. For such large conceptual data models, it might be useful to spread the concepts over sections (‘sub-areas’), such as Shipping, Warehousing, and Production (and maybe a section General).

We could extend the explanation by mentioning the data type per property as well, say by replacing $P_{j,k}$ by $P_{j,k}$ (**being a date**), by $P_{j,k}$ (**being a time**), by $P_{j,k}$ (**being a date and time**), by $P_{j,k}$ (**being an integer**), by $P_{j,k}$ (**being a decimal number**), by $P_{j,k}$ (**being a string**), or by $P_{j,k}$ (**being a string of exactly** *n* **characters**), etc.

5. Textual SSDs

As mentioned earlier, the dynamics / processes of FRs can be represented by textual SSDs, which are schematic depictions of the interactions between the primary actor (user), the system (as a black box), and other actors (if any), including the messages between them [15]. Following [8], we present here a grammar for textual SSDs, in BNF (Backus–Naur form). The terminals are written in bold. The nonterminal A stands for 'atomic instruction' (or step), P for 'actor' (or 'participant'), M for 'message', S for 'instruction' (or SSD), N for 'instruction name', and D for 'definition':

A ::= P ➡ P**:** M /* *'X ➡ Y: M' means: 'X sends M to Y'*
P ::= **System** | …
S ::= A | S **;** S | **begin** S **end** | **if** C **then** S [**else** S] **end**
| **while** C **do** S **end** | **repeat** S **until** C | **perform** N
| S **,** S | **maybe** S **end** | **either** S **or** S **end** /* *introducing non-determinism*
D ::= **define** N **as** S **end**

Informally, the construct 's1**,** s2' indicates that the order is irrelevant ('do s1 and s2 in any order'), 's1**;** s2' indicates 'do s1 first; then do s2'. The expression '**perform** N' represents the *Include* or *Call*. '**Maybe** s **end**' means 'do s or do nothing', and '**either** s1 **or** s2 **end**' means 'choose between doing s1 and doing s2'. **System** represents the system under consideration.

The values for the nonterminals B, P, M, and N are application-dependent, or 'domain specific' (except the terminal **System** for the nonterminal P). Those values will appear during the development of the specific application.

For *atomic instructions* where at least one actor/participant is **System**, we distinguish the following situations (where Actor ≠ **System**):

1. Actor ➡ **System:** i — Indicates the input messages the system can expect
2. **System** ➡ **System:** y — Indicates the transitions or checks the system should make
3. **System** ➡ Actor**:** o — Indicates the output messages the system should produce

Instruction (1) is called an *input step*, (2) an *internal step*, and (3) an *output step*. An atomic instruction not involving **System** is called an *external step*. A quite common, simple basic pattern is: *input step*, followed by an *internal step*, and then followed by an *output step*.

6. Explainability of a given textual SSD

We developed a mapping from textual SSDs to natural language (English in this case). The mapping rules originate from [9], but are slightly modified. Function F below inductively maps textual SSDs to English, assigning to each tSSD an expression in English in terms of the direct constituents of that tSSD, according to the *compositionality principle* [16]. Most mappings are straightforward, i.e., leave the language constructs as they are. The most important non-straightforward functions are:

1. F(Actor ➡ **System:** γ) $\stackrel{def}{=}$ **the** F(actor) **asks the System to** F(γ) /* for Actor ≠ **System**
2. F(**System** ➡ Actor**:** γ) $\stackrel{def}{=}$ **the System sends** F(γ) **to** F(actor) /* for Actor ≠ **System**
3. F(Actor ➡ Actor**:** γ) $\stackrel{def}{=}$ **the** F(actor) **does** F(γ) /* if the same actor is mentioned twice
4. F(e1**;** e2) $\stackrel{def}{=}$ F(e1)**.** <newline> F(e2) /* Sequential order is indicated by a dot
5. F(e1**,** e2) $\stackrel{def}{=}$ F(e1) **and** <newline> F(e2) /* Arbitrary order is indicated by '**and**'
6. F(n) **means:** F(e) **end** /* if n was introduced by '**define** n **as** e **end**'

Ad 1-3: F(actor) often is a (human) user but it could be an external system as well
Ad 3: If the same actor is mentioned twice, the step indicates what that actor has to do.

Often Actor = **System**
Ad 6: This 'follow-up' can be put after the complete mapping of the main text

See for the nature of the message γ in the basic steps the explanation of atomic instructions in Section 5. For an actor, message, or instruction name x, F(x) could simply be x itself when it was well-chosen, as in Example 2.

Essentially, the mapping boils down to replacing the basic steps by some standard sentence constructions, replacing '**;**' by '**.**' , replacing '**,**' by '**and**' , and replacing '**define** n **as**' by 'F(n) **means:**'. The marker '**end**' in the NL expression is needed to indicate the end (the 'scope') of a construct in the same way as in the formal language; cf. the use of markers such as "[end remark]" in NL texts by authors such as E.W. Dijkstra.

In [17] we applied the mapping rules to Larman's well-known large use case *Process Sale* [15]. The following example contains an initial fragment of it after application of our mapping rules and keeping the originally bold texts in bold. The complete elaboration can be found in [17].

processSale **means:**
  The Cashier **asks the System to** make new Sale**.**
  The System **does** create Sale**.**
  While customer has more items **do**
    perform handleItem**.**
    The System sends description and running total **to** Cashier
  end.
  The Cashier **asks the System to** end Sale**.**
  ⋮
end

7. Early validation of the approach

We not only applied our approach to Larman's large use case *Process Sale* but also to a specification document we were working on. Size of its conceptual data model: >35 concepts, >180 properties, and > 65 references. It generated 5 to 6 pages of sentences in English. Preceded by a short and simple legend explaining notions like *optionality* and *uniqueness conditions*, the conceptual data model turned out to be well understandable by the customer.

One of the authors has been working for many years in the food processing industry and has first-hand experience with a rich variety of domain experts in the industry, ranging from the owners of companies to operators on the shop floor. This experience has taught two lessons: (1) for real validation you need to work with the people who are actually "doing the processes" (whose knowledge of the process is "by acquaintance" and not "by description"), and (2) the capability of discussing rigorous functional specifications is not dependent on educational or organisational level, but on the capability of reflecting on the process.

One author applied the rules for generating NL-expressions for both the conceptual data model and the textual SSDs to our formal specifications. The other author ('from industry') subsequently assessed the understandability of the generated NL-specifications from the viewpoint of a typical domain expert (with first-hand knowledge of the processes and capable of reflecting on the processes).

The results from this validation of our approach showed that (1) this kind of representation of formal specifications in NL-expressions can considerably improve the understanding and discussion of the formal requirements by the customer and (2) the choice of property names in the conceptual data model is a really delicate issue for the understandability of the expressions (as also signalled in [10]).

Prototypes and test cases might be developed to check the behaviour in the user world. However, such prototypes and test cases are 'just' a means to an end but they cannot replace a complete check of the specifications. The customer must validate all the requirements as such. Therefore, we regard the generated NL-expressions as the crucial step in validation that must "serve as a basis for common agreement among all parties concerned", to cite Barry Boehm [1].

8. Discussion

The main question in this paper is how to achieve the combination of a full formal specification of the functional behaviour of the system (under development) with adequate explainability of that formal specification to the user community. The answer is based on the idea that any validation of the specification by the user community must be based on explainability (and understandability) of the system (under development) to the users.

As argued before, the relevance of both conditions should be self-evident: we don't want choices concerning the functional behaviour of the system "hidden" in the software. We want unreserved understanding by the user community of what will be delivered. Satisfying both conditions implies that the system behaviour can be fully known and accepted by the user community after a requirements phase. An additional point is that changing specifications later on (of course a normal phenomenon in a dynamic world) can be fully dealt with on the conceptual level, and does not require knowledge of implementation details.

The main contribution of the paper is to provide a method for mapping formal specifications to expressions in natural language (NL), for the *statics* part as well as for the *dynamics* part; so for the complete conceptual 'blue print' of the system to be developed.

This approach makes the formal specifications accessible and understandable for the users, without making concessions to the precision of the expressions. It is a necessary step in explaining the specifications to the users, in many cases to be followed by further discussions between user and requirement engineer about the meaning and impact of the specifications in the user world. But wherever such explanations and discussions may go, the reference for the system under development consists of the precise formal specifications and their mappings to understandable expressions in NL.

9. Conclusions and Future Work

In SW development, the specifications must simultaneously serve two purposes: providing a full formal specification of the system behaviour to be implemented, and being a basis for giving the user an understandable specification of the system under development to be validated. These two purposes are often considered to be conflicting, solutions either leaning towards formalism at the expense of understandability for the user, or leaning towards natural language descriptions at the expense of precision. In this paper we have shown that it is possible to have it both ways, by providing a full formal specification of the behaviour of the system under development in combination with syntax-driven mapping rules from the formal specifications to expressions in natural language. We have explored the different meanings of explainability in SW development, having separated (1) the user explaining "their world" to the requirements engineer to provide understanding, (2) the requirements engineer explaining their interpretation of the requirements to the user, providing insight in the relation between system and user world, (3) the SW designer explaining their design to the requirements engineer, demonstrating its completeness and conformity, and (4) similarly, the programmer explaining their software to the SW designer. In short: regarding functional behaviour, all formalization steps from user world to the system under development belongs to the domain of requirements engineering and formal specifications of functional behaviour are made accessible for user validation by generating expressions in natural language. All later discussions about the functional behaviour of the system can be based on the formal specifications only, making knowledge of its SW implementation unnecessary.

Future work. Since we can specify Conceptual Data Models (CDMs) and Textual SSDs in a systematic way and are able to give complete mapping rules to map CDMs and Textual SSDs to descriptions in natural language, it is possible to develop a tool for the automatic generation of such descriptions. For large CDMs, it might be useful to spread the concepts over sections (and subsections), e.g., with sub-areas such as Shipping (which sub-area could be further subdivided into Inbound and Outbound), Warehousing, and Production (and maybe a necessary section General). And maybe there might arise a need for a subdivision of the properties within a concept as well. Such structuring still has to be investigated. Finally, it will be useful to do more empirical validation of our approach in customer projects.

References

[1] B. Boehm: Software Engineering. IEEE Transactions on computers, vol. C25 no 12 (1976), pp. 1226-1241.

[2] Object Management Group, Unified Modeling Language (UML), https://www.omg.org/spec/UML/2.5.1/About-UML.

[3] E. Kavakli, P. Loucopoulos: Goal Modeling in Requirements Engineering: Analysis and Critique of Current Methods. In J. Krogstie, T. Halpin, & K. Siau (Ed.), *Information Modeling Methods and Methodologies: Advanced Topics in Database Research* (2005), pp. 102-124. IGI Global.

[4] M. Dumas et al: Fundamentals of Business Process Management. Springer, 2018.

[5] J.L. Peterson: Petri Net Theory and the Modeling of Systems. Prentice Hall, 1981.

[6] T. Allweyer: Human-Readable BPMN Diagrams. In L. Fischer (Ed.): BPMN 2.0 Handbook, 2nd ed. (2014) pp. 217–232. Future Strategies.

[7] H. Leopold, J. Mendling, A. Polyvyanyy: Supporting Process Model Validation through Natural Language Generation. IEEE Transactions on Software Engineering 40.8 (2014) pp. 818-840.

[8] E.O. de Brock: On System Sequence Descriptions, 2020. In M. Sabetzadeh et al (eds.): Joint Proceedings of REFSQ-2020 Workshops, Doctoral Symposium, and Tracks. Pisa (2020).

[9] E.O. de Brock: An NL-based Foundation for Increased Traceability, Transparency, and Speed in Continuous Development of Information Systems. NLP4RE (2019).

[10] F. Kossak, A. Mashkoor, V. Geist, C. Illibauer: Improving the Understandability of Formal Specifications: An Experience Report. In C. Salinesi, I. van de Weerd (Eds.): REFSQ 2016, Volume 8396 of LNCS (2016), pp. 184–199. Springer.

[11] K. Wiegers: Writing quality requirements. Software Development 7.5 (1999), pp. 44-48.

[12] S. Tiwari, A. Gupta: A systematic literature review of use case specifications research, Information and Software Technology 67 (2015), pp. 128-158.

[13] S.G. Yaman et al.: Customer Involvement in Continuous Deployment: A Systematic Literature Review. In M. Daneva, O. Pastor (Eds.): REFSQ 2016, Volume 9619 of LNCS (2016), pp. 249–265, Springer. DOI: 10.1007/978-3-319-30282-9_18

[14] The Oxford English Dictionary. Oxford University Press, Oxford, 1989.

[15] C. Larman: Applying UML and patterns. Pearson Education, 2005.

[16] Stanford Encyclopedia of Philosophy (in particular Compositionality), Stanford.

[17] E.O. de Brock: Converting a non-trivial Use Case into an SSD: An exercise. SOM Research Report 2018011, University of Groningen, 2018