Smart Requirements: How Smart Can They Get?

Danilo Assmann(✉)

Vector Informatik GmbH, Ingersheimer Str. 24, 70499 Stuttgart, Germany
danilo.assmann@vector.com

Abstract. In current practice requirements engineering is a text based process. The available theory and tools do not address the internal elements–the semantic structure–of requirements. We present an approach to extract a first domain model, which can also serve as basis for the system architecture, directly from the requirements. Besides the model, the approach provides also new and insightful metrics, which focus on product characteristics instead of process characteristics. The model and metrics can be used to fulfill the SPiCE (and AutomotiveSPICE 3) requirements, concerning consistency and completeness of requirement specifications.

Keywords: Requirements · SPiCE (ISO15504/ISO330xx) · AutomotiveSPICE 3 · Consistency · Model

1 What Is Wrong with Current Requirements Engineering?

One of the basic beliefs of software engineering still is that requirements are useful. They have an important influence on the final product. Actually they define and shape the product and capture the customer expectation in a reliable form [1, 2].

Coming back to the question in the header: nothing is wrong. But we can do more. We can improve our understanding of the requirements engineering process, and we can improve the usefulness of the requirements related work products. We need to make them worthwhile so developers can see and feel the advantage to create and maintain them.

Despite their importance, the way we treat requirements is not very advanced. Basically we can distinguish two existing engineering directions:

– usage of patterns (simplified grammar) [3–6]
– improved writing (clear terms and data dictionaries) [7, 8]

Even applying these techniques still leave requirements quite dumb text. This reflects also the current tooling: we can count and prioritize requirements. We can give them attributes, we can link text blocks. But actually not much has changed from using good old Word. We have sentences, paragraphs, and a lack of logic; with slightly improved usability.

Sure, there is also the area of formal specification (e.g., Z, VDM, B). Which is fascinating, amazingly complex, and still has little practical influence. [11–13].

Nevertheless, we can learn something from the usage of formal specification, and also from the successful application of function points:

P.M. Clarke et al. (Eds.): SPICE 2016, CCIS 609, pp. 431–436, 2016.
DOI: 10.1007/978-3-319-38980-6_32

– requirements describe the behavior (functionality) of the system
– everything we need to build the system, and understand the domain, is already in the requirements

In my own words: the *requirements* already *contain* a (in-complete) *model of the* application domain and the *system* (software).[1]

So how can we improve requirements engineering based on this knowledge? How can requirements become smart? And what does smart mean anyhow? Simple benefits of being smart should be:

– a complete and working data dictionary or object hierarchy
– completeness (per requirements, but also for the specification)
– consistency (for the specification and with other work products)
– freedom of conflict (for the specification)
– compliance to AutomotiveSPICE 3 and ISO 26262

That's what we want to achieve, when we make our requirements smart. Our approach to smartness relies on three techniques, which combine proven approaches with new ideas:

– usage of (domain specific) patterns
– domain-specific language (terms)
– explicit extraction of the semantic model

2 Our Concept

2.1 Why Do You Amplify "Domain Specific"?

For conferences and generic tool provider it is very nice to talk on an abstract level. But in practice most of the time life is much easier: you can be concrete. You have a given context. And when you build your model, based on this context you can make a lot of assumptions, which will simplify your life.

So in practice you will have only certain patterns of requirements. You will also have certain objects, messages, mechanisms, architectures. From what we learned so far, you can cluster your domain context in:

– software at runtime
– states (in case of finite state machines)
– messages, events
– data and data access
– runtime environment (other objects including hardware and other software)
– build environment (including configuration of the software)
– process (logical flow; technical process of the domain)

[1] Why incomplete? Many things we expect from the system are just basic needs, which are not specified anymore. It is basic knowledge of the domain. Refer to the Kano model for deeper insight on this topic [9, 10].

The objects in the last three groups will be specific to the topic (domain); and some of the others also. So it is not possible to provide a standard dictionary. Based on our experience the built up of such a dictionary does not take long.

The relations (defined by the verbs) follow the context. For generic contexts (data access, software, messages) we propose to use a generic standard set, with a minimal set of actions. They all should be clear and redundancy free defined.

E.g., "check" and "validate" have some overlap. So use only one of them, or use a third more clear/precise term.

2.2 What About Our Patterns?

Now we provide a short insight into our approach. Based on the needs from complying with the safety standard we provided a set of basic patterns (top-down). These patterns where applied in several AUTOSAR components. Basically we can distinguish three areas of patterns:

- services (functionality used via API, callback, callout)
  - e.g., <Subject> shall provide a service to <functionality>
- states (description of a finite state machine)
  - e.g., <Subject> shall provide a mode to <state>
- use cases (all pre-build and post-build configuration)
  - e.g., <Subject> shall be used in <scenario>

The next step was to take a sample (around 10 %) of used specifications and analyze the real-world requirements and refine the patterns. We did this in two steps:

(1) extracting refined patterns from the examples. E.g., Object - <pattern indicator> - relation - [definition] - attribute - object[2]
(2) After having the sample and the application distribution of patterns, we could derive meta-patterns (grammar): e.g., Object - [[object - condition] - relation - [definition] - [attribute] - object]

By additionally applying a simple writing guide, the readability of requirements improves. The structure helps to make clear what should happen through this requirement. It simplifies the process of writing, because you know what to place where.

Requirements for a safety process complying with ISO 26262 demand semi-formal notations (ASIL C and D). Our patterns fulfill this requirement. So we do not use formal specification, but a reduced set of words and grammar of natural language.

2.3 Does Semantics Mean Smart?

Now we become real "smart" by going to the next level. The refined patterns allow us to recognize (and check) the relevant elements of a requirement. This allows us to access the inner structure of a requirement and create a kind of a model.

[2] <pattern indicator> is the predefined text segment from our basic patterns above.

The model gives us possibility to transform the text into a graph. A graph shows the inherent structure of a requirement and also of a requirement specification. It is a sketch of the natural architecture. This changes also the perspective: instead of looking at requirements, the defined objects are the main elements of the model. The requirements provide only different views of object relation.

The object based view has the advantage that all relations to an object can be seen in one place, even if they are scattered through the whole document (or several documents). This simplifies the review for consistency and completeness (also in case of state machines). That is requirements "engineering". Not just read and check. So how does this work?

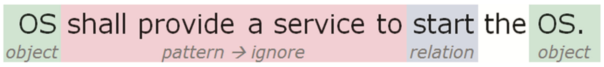

Fig. 1. Simple example for requirement with markup.

Basically we do two things: remove the basic pattern elements (and other phrases) *and* tag the words following the grammar of the sentence (Fig. 1).

As you can see in the examples the tagging is quite easy (Figs. 1 and 2). You do it sentence by sentence and the number of objects and relations is limited by the natural length.

Some direct benefits while doing it manually:

– deep check on patterns: how many requirements conform to which pattern? Justify with clear reasons all deviations from pattern.
– build and maintain a data dictionary: build up the data dictionary including an object model (hierarchy)
– relation check: keep your writing clear and tidy; also for the verbs. Do the same words (verbs) have same meaning, and have different words always different meaning. It is possible to create a white-list, which covers the common terms (Fig. 2).

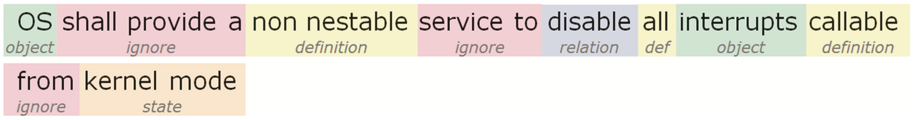

Fig. 2. Medium complexity requirement example.

2.4 Tooling

So up to now it is quite boring manual work, with high maintenance effort if you have to do it again for a new version of a specification.

The idea was to support this process with a very simple tooling. Which are around 100 lines of Java code.

It works completely text based (just text replace) and has currently no support for natural language processing. It performs a cleanup of unnecessary text, replaces matches with tags and counts and removes unused elements. As final output the tool provides tagged text. We use simple hash-tags with type identifier. During the learning process different output is possible, such as the not used words.

Besides the output, which can easily be translated in GraphViz syntax, it provides some metrics. We use it as a transformation tool (input to output) with a given threshold, so it reports after each run, if the transformation is "passed" or "failed" (Fig. 3).

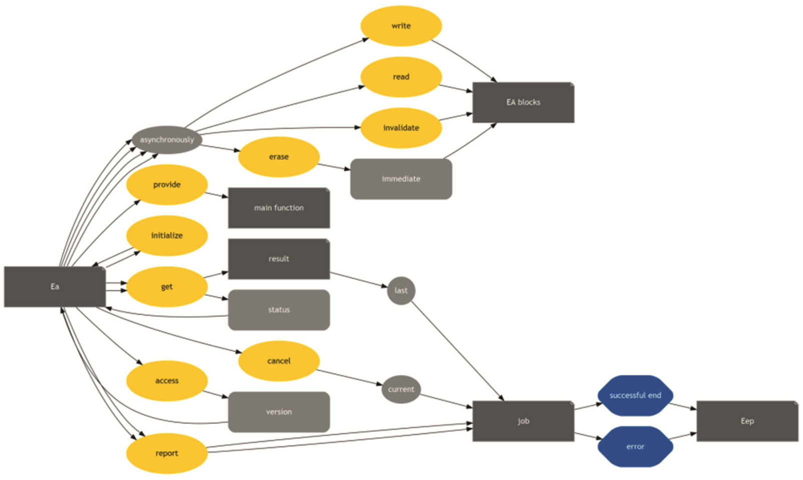

Fig. 3. Simple example of a generated requirement model.

2.5 Graph and Metrics

One advantage of the content based transformation of requirements into a model is that we can draw charts. And we can get these charts without any additional effort. They are kind of a side-effect. The chart gives you immediate feedback on the complexity and structure of your specification, and this in one page (Fig. 3). Basic metrics are visualized: #objects (#objects per requirement), #states, #levels (depth of requirements), #conditions, #relation (and the linkage [the edges]), #areas of cohesion. Based on these factors every requirement specification can be characterized. They define an individual "flavor". They have a characteristic on first sight.

3 Conclusion and Outlook

As mentioned throughout the paper, we haven't done but our first steps. So we still need more experience.

- Still learning: first we need to understand deviations in the process, then we will really understand the product.

- Improve tooling, if necessary: in commercial tooling NLP support needs to be provided. This will be still fast enough, but will give more precise results and much easier handling. E.g., the automation can be much higher during learning phase.
- Support for data dictionaries/object models: currently we rely only on word lists. From process modeling we have a lot of experience how to build hierarchical models. This can also be applied here.
- Diffs on versions: re-read for new versions of specifications will be fast (no effort) and allow checks on the model. So we can evaluate if there are relevant/risky changes.
- Integration with agile (e.g., SBE or BDS) practices: the notion would be to generate the headers of the data tables directly out of the requirements (objects/attributes/ environment/states/…). And with changes you can easily see if relevant elements were added or removed.

References

1. ISO 33060
2. ISO 12207 (2008)
3. Hagge, L., Houdek, F., Lappe, K., Paech, B.: Using patterns for sharing requirements engineering process rationales. In: Dutoit, A.H., McCall, R., Mistrík, I., Paech, B., et al. (eds.) Rationale Management in Software Engineering, vol. 1. Springer, Heidelberg (2006)
4. Konrad, S., Cheng, B.H.C.: Requirements patterns for embedded systems. In: Proceedings of the 10th Anniversary IEEE Joint International Conference on Requirements Engineering, pp. 127–136 (2002)
5. van Lamsweerde, A.: Requirements engineering in the year 00: a research perspective. In: Proceedings of the 22nd international conference on Software engineering, pp. 5–19, New York (2000)
6. Coplien, J.O.: Software design patterns: common questions and answers. In: Rising L. (ed.) The Patterns Handbook: Techniques, Strategies, and Applications, pp. 311–320 (1998)
7. Houdek, F.: Messung in der Erstellung und Prüfung von Lastenheften, Metrikon (2014)
8. Houdek, F.: Anforderungen verbessern mit DESIRe, REConf (2008)
9. Noriaki, K., Seraku, N., Takahashi, F., Tsuji, S.: Attractive quality and must-be quality. J. Japanese Soc. Quality Control (in Japanese) **14**(2), 39–48 (1984). ISSN: 0386-8230
10. Cadotte, E.R., Normand, T.: Dissatisfiers and satisfiers: suggestions from consumer complaints and compliments. J. Consum. Satisfaction, Dissatisfaction Complaining Behavior **1**, 74–79 (1988). ISBN: 0-922279-01-2, ISSN: 0899-8620
11. Behm, P., Benoit, P., Faivre, A., Meynadier, J.-M.: Météor: a successful application of B in a large project. In: Wing, J.M., Woodcock, J. (eds.) FM 1999. LNCS, vol. 1708, pp. 369–387. Springer, Heidelberg (1999)
12. Bernot, G., Gaudel, M.C., Marre, B.: Software testing based on formal specifications: a theory and a tool. Softw. Eng. J. **6**(6), 387–405 (1991)
13. Clarke, E.M., Wing, J.M., et al.: Formal methods: state of the art and future directions. ACM Comput. Surv. **28**(4), 626–643 (1996)