Requirements Defects over a Project Lifetime: An Empirical Analysis of Defect Data from a 5-year Automotive Project at Bosch

Vincent Langenfeld[1,2], Amalinda Post[1], and Andreas Podelski[2]

[1] Robert Bosch GmbH, Automotive Electronics, Stuttgart, Germany
[2] University of Freiburg, Germany

Abstract. [**Context and motivation**] Requirements defects are notoriously costly. Analysing the defect data in a completed project may help to improve practice in follow up projects. [**Question/Problem**] The problem is to analyse the different kinds of requirements defects that may occur during the lifetime of an industrial project, and, for each kind of requirement defect, the respective number of occurrences and the cost incurred. [**Principal ideas/results**] In this paper, we present a post hoc analysis for an automotive project at Bosch. We have analysed 588 requirements defects reported during the elapsed project lifetime of 4.5 years. The analysis is based on a specific classification scheme for requirements defects which takes its eight attributes (incorrect, incomplete, etc.) from the IEEE 830 standard and refines them further by distinguishing nine possible defect sources (relating to parameters, wording, timing, etc.). The analysis yields that a large chunk of the requirements defects (61%) stems from incorrectness or incompleteness. The requirements defects that are the most costly to fix are incompleteness and inconsistency. [**Contribution**] The insights gained from the analysis of the defects data allow us to review several design decisions for the requirements engineering process and to suggest new ones (such as to incorporate the classification of the requirements defects into the requirements review and into the defect reporting).

1 Introduction

Requirements defects are notoriously costly. In order to derive effective measures that help avoid common requirements defects, we need to know more about requirements defects as they occur during the lifetime of an industrial project. Typical questions are: What are the different kinds of requirements defects that occur? Which kind occurs relatively often? Which kind of requirements defect is relatively costly to fix?

In this paper, we present a post hoc analysis for an automotive project at Bosch. We have analysed the requirements defects reported during the elapsed project lifetime of 4.5 years.

The project in question is the development of a commercial DC-to-DC converter for a *mild hybrid* vehicle. The project had a runtime of approximately

five years. During this runtime, six hardware samples were produced, together with 25 software versions with a total of 2500 changes (these changes include both, defect fixes and additional functionalities). The project has more than 10.000 requirements, including customer, system, software, hardware, mechanic and test requirements. The analysis presented in this paper is based on the 588 defects in system requirements (the set of system requirements changed during the runtime of the project; its size at the end of the project is around 2000). The development process in the project followed the V-model. A review was done after every development step. The review of the system requirements was done by the engineers in the respective domain, in the presence of a system tester (as a walkthrough or as an inspection, depending on the complexity of the change of requirements). The project team consisted of about 50 team members. The work of at least 30 out of the 50 team members depended directly on the system requirements (to develop the hardware, software, mechanic or derive test cases). Out of the 50, five team members were responsible for system requirements.

In the analysis, we have used a classification scheme which is based on the IEEE 830 standard and which we have further refined with the classification of the defective part of a requirement. Our results demonstrate the applicability of the defect classification scheme, and the insight into common requirements defects in the project, we gained thereof. The analysis yields that a large chunk of the requirements defects (61%) relate to either *incorrectness* or *incompleteness*. The most costly requirements defects (most costly to fix) are *incompleteness* and *inconsistency*. In the remainder of the paper, we will present the analysis and its results based on the classification, as well as the conclusions we have drawn for improving the practice in follow up projects.

The paper is organised as follows. Section 2 describes the general approach followed by the analysis. Section 3 presents the results of the analysis. Section 4 presents the lessons learned and the conclusions we have drawn for improving the practice in follow up projects. Section 5 discusses potential threats to validity. Section 6 gives an overview of related work and puts the concepts used in this paper into a larger context. Section 7 presents concluding remarks.

2 The General Setup of the Analysis

2.1 Goals and Questions

The analysis is part of a larger research effort to investigate how the requirements engineering process can be changed in order to improve the quality of the system requirements specification. The idea is to exploit the wealth of information which is accumulated in the defect reports gathered during the lifetime of an industrial project. Concretely, we take the already mentioned DC-to-DC project at Bosch. Over the whole period of 4.5 years of the project lifetime, the requirements defects were documented in 588 defect reports. The defect reports were used to fix the defects.

The first step to extract information from this rather large number of defect reports is to choose a classification of the requirements defects. We base our

classification on the IEEE 830 standard which lists attributes that determine the quality of requirements specifications in software projects. It is widely agreed that the attributes according to the IEEE 830 standard are useful to define the quality of a requirements specification because generally, the defect of a requirement results from a violation of one of the attributes. Thus we can classify the defect according to the attribute that is violated, whether the requirement is *incorrect*, *ambiguous*, *incomplete*, *inconsistent*, *not ranked*, *not verifiable*, *not modifiable*, or *not traceable*. Here the requirements defect may be named after the violation of the attribute that results directly or indirectly in the requirements defect (for example, *not traceable* is not a requirements defect per se but may result in one).

We will use the classification of the requirements defects according to the violation of the attributes of requirements in the IEEE 830 standard for the analysis. In particular we will analyse the following questions.

1. **What classes of requirements defects occur most often?** We analyse the requirements defects in these classes for common features that could help us to detect these requirements defects or even prevent them.
   With regards to the requirements engineering process, a good strategy may be to concentrate on requirements defects in these classes before others.
2. **What classes of requirements defects occur least often?** Requirements defects in these classes may fall into one of two cases, depending on whether they occur more rarely or whether they are just detected more rarely. We need to consider both cases and either find the reason why the DC-to-DC project does not suffer from those requirements defects, or improve the detection of those requirements defects.
   With regards to the requirements engineering process, the obvious consequence of the knowledge of the absence of requirements defects in one class is to re-allocate the corresponding effort to the detection of defects in other classes.
3. **What classes of requirements defects are most costly to fix?** The number of defects per class is not sufficient per se; we furthermore need to take into account that the amount of time needed to fix a requirements defect can vary considerably, especially if detected in a later development phase.
   With regards to the requirements engineering process, the most costly defect classes call for more involved detection and prevention methods.
4. **What classes of requirements defects are least costly to fix?** The later in the development phases the requirements defects in those classes are detected, the higher becomes the risk that they induces new defects.
   With regards to the requirements engineering process, the obvious consequence of the knowledge of such classes is thus to invest the comparatively little cost to fix the defects to prevent them from becoming costly later.

2.2 Collecting the Data of the Analysis

The 588 requirement defect reports that were issued over the lifetime of the DC-to-DC project stem from two different sources, namely the requirements specification

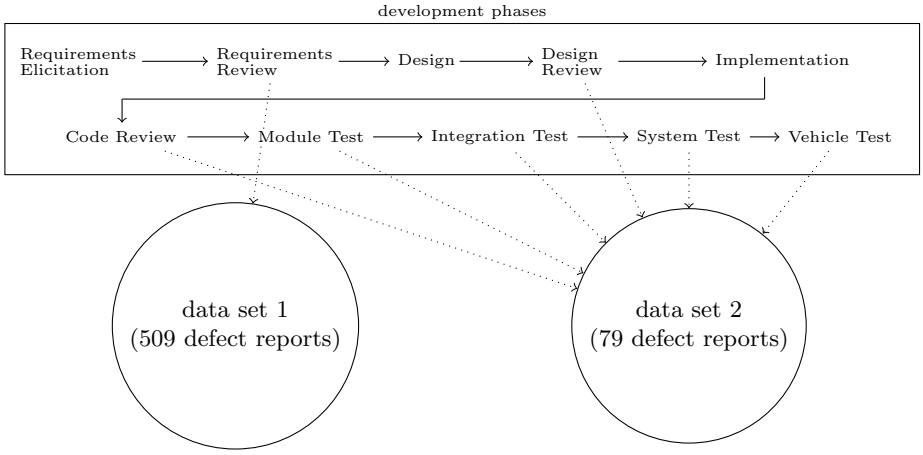

Fig. 1. The two data sets used for the analysis: data set 1 contains the requirements defects that were detected in the requirements review phase, data set 2 contains the requirements defects that were detected in later development phases. The dotted arrows depict the flow of defect reports.

reviews (data set 1) and the fault reporting system for later development phases of the project (data set 2); see Figure 1.

The 509 requirements defect reports in data set 1 (i.e., those detected during the requirements reviews) each contain a rather brief explanation of the requirements defect and a proposed solution for its removal.

The 79 requirements defect reports in data set 2 (i.e., those detected in later development phases) contain detailed information on the defect, the development phase the defect has been injected in, the development phase the defect has been detected in, and the time effort (in man-hours) it took to fix the defect and its ramifications.

2.3 Performing the Analysis

The classification of the defect reports was conducted by two employees of the DC-to-DC project, each of them being responsible for one of the two data sets. The workload between the two employees was even. In fact, due to the free structure of the reports in data set 2, the analysis of a defect in a report in data set 2 was relatively involved and time consuming.

We spent considerable effort to ensure the objectivity of the classification of the requirements defects. Fortunately, a requirements engineer who had been employed in the DC-to-DC project from the very beginning, was constantly available for questions about hard cases. We have analysed the *stability* of the classification (stability meaning that the results of the classification are not dependent on the person who performs the analysis), using 16 randomly selected samples of the defect reports from both data sets. With 75% agreement (Cohen's

Kappa of $\kappa = .57, p < .001, n = 16$), we obtain that the result of the classification is *moderately stable*, in the terminology of [2].

During the course of the analysis, we found that the classification based on IEEE 830 (as described in Section 2.1 was too coarse to yield informative results with regard to our first research question. We have refined the classification further, based on nine possible *defect sources*, where we use *defect source* to refer to the specific part of a requirement which is the cause of the requirements defect. We list the nine *defect sources* below.

parameter The defect lies in a parameter (for example, the value of the parameter is written directly in the text, or parameter has a wrong value, or the parameter has the wrong unit).

variant The defect lies in the elements that are used to document or manage variants or versions (for example, the marker to indicate that a requirement is valid only for one version, is missing).

wording The requirement is not written compliant to formulations, template phrases or the desired precision of requirements that were agreed upon in the project (for example, usage of words like 'would' or 'should').

timing The defect is in a specified timing parameter (for example, timing is not correct or even possible).

state machine The defect is related to the state machine that is modelling the system behaviour (for example, the guard of a state change is missing).

calculation The defect is related to a calculation or comparison (for example, wrong sign, wrong comparison, use of wrong variable or function in a calculation).

figure The defect is in a figure or related to a figure (for example, the depicted process is labelled with wrong numbers).

organisation The structure of the requirements document is flawed (for example, missing/wrong links to resources, misplaced/duplicated requirements).

functionality The requirements defect is related directly to a description of a functionality and none of the eight other defect sources applies (meaning, the requirement has a defect that cannot be fixed by removing one of the other defect sources).

The above list of defect sources is based on the ideas behind the defect classification schemes described by Chillarege *et al.*[1]. The *defect sources* are mutually orthogonal (no requirements defect can be assigned to two *defect sources*). The *defect sources* cover as many requirements defects as possible with only few classes. We have been able to successfully apply the above classification scheme for requirements defects in the DC-to-DC project; the investigation whether the classification scheme is generally applicable (for requirements defects in other projects) goes beyond the scope of this work.

3 Results of the Analysis

The results of the classification of the 588 requirements defects in data sets 1 and 2 are presented in Tables 1 and 2. Table 1 refers to the classification

	incorrect	ambiguous	incomplete	inconsistent	not ranked	not verifiable	not modifiable	not traceable	Σ
data set 1	175	41	135	22		29	72	35	509
data set 2	30	7	19	9	1		3	10	79
Σ	**205**	48	**154**	31	1	29	**75**	45	588

Table 1. The result of the first step of the analysis which uses the classification based on the IEEE 830 standard.

of the requirement defects based on the IEEE 830 standard (as described in Section 2.1), and Table 2 to its refinement based on the defect source (as described in Section 2.3). Whereas the classification based on the IEEE 830 standard covers 100% of of the 588 requirements defects in data sets 1 and 2, the refined classification covers only 67% (395 out of 588). The remaining third of requirements defects cannot be assigned to one of the nine defect sources used for the refined classification.

Next we use the classification in order to analyse the four questions which we have formulated in Section 2.1.

What classes of requirements defects occur most often?

Table 1 shows that 61% of the requirements defects (359 of 588) belong to two out of the eight possible classes, namely *incorrect* and *incomplete*. In order to analyse the requirements defects in more depth, we will use the classification according to the defect source; see Table 2.

parameter As in programming, parameters are used to abstract away from concrete values. Concrete values used in requirements are not written directly into the requirement; instead, they are referenced by a parameter. The concrete value of the parameter is defined in a specific data base. For example, the variants of the DC-to-DC converter get defined only through the assignment of variant-specific values to the set of parameters (except for special cases; see below).

Out of the 86 requirements defects whose defect source can be assigned to *parameter*, 18 fall into the class *incomplete* (16 in data set 1 and 2 in data set 2). Here, simply the assignment of concrete values to parameters in the data base had been forgotten. Another 33 out of those 86 requirements defects fall into the class *incorrect* (26 in data set 1 and 7 in data set 2). We note that 7 cases out of the 33 (which happen to be among the 26 that belong to data set 1) share a pattern. That is, the parameters had been assigned tentatively (to some seemingly plausible value) before the information on the

	data set 1									data set 2								
Detail / Iso	incorrect	ambiguous	incomplete	inconsistent	not ranked	not verifiable	not modifiable	not traceable	Σ_1	incorrect	ambiguous	incomplete	inconsistent	not ranked	not verifiable	not modifiable	not traceable	Σ_2
parameter	26	1	16			1	8	22	74	7	2	2				1		12
variant	13	4	34	3			1	4	59								6	6
wording	35	12	9	2		14	13		85									
timing	4		4						8			2	2					4
state machine	15	1	3	2					21	5		7	2					14
calculation	15								15	7								7
figure	6	1	9	2			1	1	20									
organisation	7	3	9	1			7	8	35		1		1	1			2	5
functionality	11	4	1	2		2	3		23	3	1	2	1					7
no category	43	15	50	10		12	39		169	8	3	6	3			2	2	24
Σ	175	41	135	22		29	72	35	509	30	7	19	9	1		3	10	79

Table 2. IEEE-830 classification and defect source. Using the nine defect sources we managed to classify 395 defects (340 from data set 1 and 55 from data set 2).

hardware was available, just in order to be able to run a test, and then, once information on the hardware was available, the update to the correct value was forgotten.

variant Out of the 65 requirements defects whose defect source can be assigned to *variant*, 34 of them fall into the class *incomplete* (all of them stem from data set 1, i.e., none is from data set 2). To give an example, a requirement defining the characteristics of the cooling fan, which obviously applies only to the variants of the DC-to-DC converter that actually have a fan, had not been marked as such.

All these 34 requirements defects have in common that they arise from forgetting the treatment of a special case. As explained above, the variants of the DC-to-DC converter were usually defined through the assignment of variant-specific values to the set of parameters. That is, a requirement that refers to only one variant (as in the example above) is a special case. Such variant restrictions had to be written as part of the requirement, which could easily be forgotten. Allocating a specific attribute in the database of system requirements to record the variants a requirements applies to might have helped with variant management but might have introduced other issues.

wording The 85 requirements defects whose defect source can be assigned to *wording* all stem from data set 1, i.e., none are from data set 2. This means that, if a *wording* defect was detected then it was detected in the requirements review phase. The fact that none was detected in a later development phases means that either all of them were detected through the requirements reviews,

or, at least, those that were not did not cause any follow-up defect, at least not one that was detected in later development phases.

We now consider 12 out of the 85 *wording* defects that fall into the class *ambiguous*. The particularity of the defects in this subset is that their follow-up defects can be hard to detect due to the subtle ways in which they exteriorise. To give an example, the wording `average` is ambiguous (for example, because the exact set of samples is not specified). Different interpretations of the term `average` may lead to results whose incorrectness is not immediately apparent.

The *wording* defects that belong to classes other than *ambiguous* are rather of cosmetic nature, i.e., with little or no potential to cause damage (because the user gives the requirement its intended, rather than its actual meaning). We give two examples to demonstrate this. The first example: `After the request to rise the target voltage, the PCU reaches [voltage]` belongs to the class *incorrect*; the correct wording is: `After the request to rise the target voltage, the output voltage U_HV reaches [voltage]` (it is not the DC-to-DC converter that reaches the voltage but the output voltage; PCU stands for Power Control Unit). The second example: `[. . .] has an output voltage level [. . .] before t_LV_CTRL has elapsed` belongs to the class *not verifiable*); the correct wording is: `[. . .] has an output voltage level [. . .] at the latest when t_LV_CTRL has elapsed`.

Our analysis determines that the high number of *wording* defects (belonging to classes other than *ambiguous*) stem from copying similar requirements that already had the defect.

state machine Out of the 21 requirements defects whose defect source can be assigned to *state machine* (3 in data set 1 and 7 in data set 2), 10 fall into the class *incomplete*. To give an example, non-determinism was introduced by accident, e.g., by forgetting guards on outgoing transitions (where the transition to an *error state* should be chosen in any case, if possible).

calculation All of the 22 requirements defects whose defect source can be assigned to *calculation* (15 in data set 1 and 7 in data set 2) fall into the class *incorrect*. To give an example from data set 1, the requirement: `The overshoot caused by the LV jump must not exceed U_HV_DUMP_OVERSHOOT` should have been: `The overshoot caused by the LV jump must not exceed U_HV_Target + U_HV_DUMP_OVERSHOOT`. In the 7 cases that belong to data set 2 (i.e., requirements defects detected not during the requirements review but in later development phases), the requirements defects were particularly costly to fix (8 man-hours per requirements defect, on average). All of these 7 cases in data set 2 correspond to the same kind of mistake, namely a wrong sign or the wrong comparative symbol ($<$ instead of $>$, etc.).

figure The 20 requirements defects whose defect source can be assigned to *figure* all stem from data set 1, i.e., none are from data set 2. In the DC-to-DC project figures are always backed by requirements (written as text), which information from figures could be validated with. The fact that none was detected in a later development phase means that either all of them were detected through the requirements reviews, or, at least, those that were not

did not cause any follow-up defect, at least none that was detected in later development phases.

What classes of requirements defects occur least often?

There are three classes of requirements defects that occur least often: *not verifiable* (29), *inconsistent* (31), and *not ranked* (1); see Table 1.

not verifiable The fact that the number of requirements defects that fall into the class *not verifiable* is relatively low can be explained by the combination of two measures taken for the two processes of requirements elicitation and requirements review. For the process of requirements elicitation, the requirement engineers formulated the functional requirements while having in mind their translation into a restricted subset of natural language which itself maps directly to a formal language (a subset of temporal propositional logic; see [9]). For the process of requirements review, from the beginning of the project, a test engineer had to be present in every review meeting.

inconsistent The relatively low number of requirements defects that fall into the class *inconsistency* may seem surprising at first, given that the project has more than 1600 system requirements. The explanation for the low number lies in the fact that the project is the development of a new product and that the set of requirements engineers did not change over the whole project lifetime (i.e., the risk of inconsistency between new and old requirements was relatively small).

As we will discuss further below, the cost for fixing can be relatively high for requirements defects that fall into the class *inconsistency* (29h per requirements defect on average; 86h in the worst case).

ranked There is only one requirements defect that falls into the class *ranked* (which stands for *ranked for importance and stability*). Even though the project mostly adheres to the IEEE 830 standard for requirements specifications, an exception is made in this class and it was decided to omit the ranking of requirements. In the project, all system requirements are *equally important* since every single one of them gets implemented in the final product (customer requirements that need not to be implemented are not elicitated as system requirements).

What classes of requirements defects are most costly to fix?

Table 3 shows that the most expensive requirements defects fall into the class *inconsistent* (29 man-hours per requirement), the class *incomplete* (17 hours), and the class *incorrect* (12 hours).

On single cases, the cost for fixing a requirements defect of the class *inconsistent* can be rather high: 86 hours in the example of a requirements defect with the error source *state machine* which was detected during system testing. The reason that it was not detected in the requirements reviews lies in the fact

	average effort (in hours)									requirements defects								
detected by	incorrect	ambiguous	incomplete	inconsistent	not ranked	not verifiable	not modifiable	not traceable	avg.	incorrect	ambiguous	incomplete	inconsistent	not ranked	not verifiable	not modifiable	not traceable	% per phase
reqs review*	3	1	5	6				3	3.4	2	1	5	4				2	18
design review	9	3	11	1			3		5.4	9	3	1	1			1		20
module test	5		28						16.5	1		1						3
system test	14	16	16	86	3		3	1	19.9	10	1	7	1	1		1	2	30
vehicle test	33	23	38	23			11	8	22.7	2	2	4	3			1	6	24
other	10		2						6.2	3		1						5
avg.	**12**	11	**17**	**29**	3		6	4		30	7	19	9	1		3	10	

Table 3. The effort spent on fixing a requirements defect (in man-hours per requirements defect, on average, rounded to integers). By effort we mean the set of activities that were needed to fix the defect in the requirement and all of its ramifications, including reviewing, implementation, and testing. The columns refer to the IEEE 830 classification. The rows refer to the development phases. The requirements defects stem from data set 2, i.e., from development phases later than the requirements review phase. This applies also to the requirements defects in the first row marked reqs review*. These stem from work on the requirements that took place after the requirements review phase, for example when the requirement was refined (into software, hardware, or mechanic requirements), when another (closely related) requirement was added, or when formal analysis in the style of [7,8] was applied. The average is calculated on the basis of the corresponding set of requirements defects whose size is listed in the table on the right, under the heading "requirements defects".

that it involves 16 requirements from different requirements documents. The 16 requirements specify interacting conditions on error signals.

Regardless of the class into which a requirements defect falls, the later in development it is detected, the higher is the effort necessary to fix it. This general tendency is confirmed by the numbers in Table 3. The effort lies between 3 and 5 hours for the early development phases (reqs review* and design review), whereas it rises to 23 hours for the latest development phase (vehicle testing).

Table 3 refers to only 76 out of the 79 requirements defects in data set 2. For three requirements defects, two of class *incorrect* with the defect source *calculation*(33 hours resp. 18 hours), one of class *incorrect* with the defect source *parameter* (32 hours), we were unable to determine the development phases in which they were detected.

We did not set up a table for the classification of requirements defects according to the defect source because the basis for calculating the average cost would become somewhat thin. We only mention that the most costly defect sources are *functionality* (14 man hours per requirements defect on average), *state machine* (14 hours), and *parameter* (12 hours).

What classes of requirements defects are least costly to fix?

Table 3 shows that the least expensive requirements defects fall into the class *not traceable* (4 man- hours per requirement on average) and the class *not modifiable* (6 hours) (we do not take into account the class *not ranked* for the same reasons as explained above).

Among the development phases, the highest cost occurs with the requirement defects detected in the latest development phase, i.e., vehicle test (8 resp. 11 man-hours per requirement on average). We observe, however, the increase of cost with respect to the early development phases (requirements review* and design review, 3 man hours per requirement on average) is not as drastic for *not traceable* and *not ranked* as with the costly requirements defect classes which we have discussed above.

The maximal cost for fixing a single requirements defect of the class *not traceable* was 18 hours. The maximal cost for fixing a single requirements defect of the class *not modifiable* was 6 hours. This is still far away form the maximal cost of 86 hours for fixing a single requirements defect of the class *inconsistent*.

4 Lessons Learned

In Section 3 we have presented a post hoc analysis of the data collected during a 5-year industrial project. In this section, we will present the conclusions which we have drawn and which may help to improve the practice in follow-up projects.

The results of the analysis seem to justify a number of decisions that have been made regarding the requirements engineering processat the beginning of the DC-to-DC project. We list these decisions below.

Include test engineers in the project from the beginning. In every requirements review session, a test engineer participated. As the analysis reveals, the effect of the decision is that *not verifiable* requirements were detected during the reviews and not later during testing.

Separate parameters in the requirements from their concrete values. The requirements are formulated using a parameter, i.e., a name for a value (instead of the value itself). The parameter is bound to a concrete value only in the parameter data base. The motivation behind this decision is to help the management of variants (since the set of parameter values can be defined individually for each variant and the set can be exchanged without modifying the requirements). The analysis reveals that this decision introduced a rather large number of requirements defects. However, these requirements defects are of the kind that can be detected automatically. Since the analysis also reveals that the number of modifiability defects linked to parameters and version management is small, the decision seems well justified. Another effect of the decision is to minimise the risk of incorrectness defects (due to forgotten updates of parameters values). Since the analysis also reveals that incorrectness defects are among the most costly to fix, the benefit is apparent.

Develop the requirements specification in a refinement process along the functional structure. Concretely, in the project, the functionality of the DC-to-DC converter was decomposed into sub functionalities with defined interaction and responsibility; this decomposition was iterated until the single parts could be described by few requirements. This means a lot of effort spent on the front-loading (with a detailed system concept, with respect to both, a functional and a component view, which was then used to organise and detail the system requirements). Since the analysis reveals that the number of inconsistency defects in the DC-to-DC converter project was rather low (considerably lower than, e.g., in the projects studied in [6,4,7]), the decision seems effective in decreasing the risk of inconsistency defects.

We next list a few recommendations for the requirements engineering process that seem justified in light of the analysis.

Apply automated tools to detect inconsistencies. Table 1 shows that 9 out of 31 requirements defects that fall into the class *inconsistent* were not detected during the requirements review phase. Table 3 shows that those 9 requirements defects have a rather high cost for fixing the defect of 29 man-hours per requirement on average (as we have described above, in one case, where the cost amounts to 86 hours, the inconsistency involves 16 requirements which specify an intricate interaction between error signals). In the future, the systems that we develop will become even more complex, and the risk that a requirements defect escapes the manual review process will become even higher. This calls for the use of automated tools that use model checking techniques to detect even elaborate forms of inconsistencies between requirements (and even between timing constraints); see, e.g., [7,8,9]. The use of automated tools involves an initial extra effort which is needed for formulating requirements in a machine-readable format. Our analysis suggests that the investment of such an effort might pay off.

Include the type of the requirements defect in the defect report. In our analysis, we specified the type of a requirements defect by the class (in the classification based on the IEEE 830 standard) and/or by the defect source (the defect part of the requirement). The person who writes the defect report will know the type and to write it down seems to create only little overhead. In contrast, to reconstruct the type from a defect report is a rather involved and time consuming task (a task that was necessary in our post hoc analysis). The immediate availability of the type of the requirements defect means that this useful information can already been taken into account during the requirements engineering process, for example in review meetings.

Analyse requirements defects in order to screen the requirements engineering process. Without an analysis of the requirements defects, information on the requirements defects lies dormant in the data base of defect reports. Information such as the information gathered in Tables 1-3 is, however, useful to review decisions that have been made regarding the requirements engineering process. This information is useful continuously during the project, and it is useful in order to give recommendations for follow-up projects.

5 Threats to Validity

In this section, we analyse threats to validity defined in Neuendorf [5] and Wohlin [11].

5.1 Construct Validity

Experimenter Expectancies [11] Expectations of an outcome may inadvertently cause the raters to view data in a different way. This threat applies to the classification of the defects, as one of the raters was aware of the results reported in the related work. However, the other analyst was not familiar with those results. The reliability analysis in Section 3 suggests that the classification was not biased.

Semantic Validity This threat arises if the analytical categories of texts do not correspond to the meaning these texts have for particular readers. In this analysis the classes are clearly defined by IEEE 830 [3] so this threat is minimised. For the definition of the defect source we named the source in a most unambiguous way, gave an explanation of the source and several brief examples to minimise this threat.

5.2 External Validity

Sampling Validity [5] This threat arises if the sample is not representative for requirements defects. In this study we analysed all requirements defects detected during the project's elapsed runtime written down either in review reports or in the fault data base. There is the risk that not all defects were tracked this way. However, as it is not allowed in the project to change requirements without a tracking number to a change request this risk is low. Another risk is that the project is not yet finished. However, as the product will go into production in six months and the product has passed thorough testing both at Bosch and at the customer we expect that all critical defects are already uncovered. Still the results may not be transferable to other projects. In this project special care was taken to ensure testable and modifiable requirements. Therefore we assume that the results presented in Section 3 may differ with that respect from other projects.

Interaction of Selection and Treatment [11] This threat arises if the raters in this study (see Section 3) are not representative for Bosch engineers. The classification was done by a student and a PhD student at Bosch, and supervised by a requirements engineer at Bosch. The requirements engineer also classified a small sample. The reliability analysis in Section 3 suggests that the classification is sufficiently independent of the raters with respect to the IEEE classes.

5.3 Conclusion Validity

Low statistical significance [11] This threat is concerned with statistical tests of low power. The stability analysis conducted for the IEEE 830 uses a small sample of only 16 reports. This stability analysis should give a picture of the stability of the classification. For cases where the analysis were unsure of the classification a requirements engineer from the project was consulted.

There were only 79 reports of requirements defects that slipped the requirements review at the end of the requirements development phase, thus the number of data points in data set 2 is fairly low, especially for the calculation of the average times in Table 3. This cannot be helped, as we took all defect data from the project, so we could not increase the selection.

6 Related Work

The work most closely related to ours is perhaps the work by Ott in [6] which also describes an empirical analysis on requirements defects (there, at Daimler AG). The work in [6] refers to requirements on a higher level than the system requirements to which our work refers. The requirements in [6] would be considered customer requirements at Bosch. Another difference lies in the granularity of the analysis. The work in [6] uses a classification on the same level of abstraction as our classification based on the IEEE 830 standard. The work in [6] does not refine the analysis in the way we do by considering the defect sources (*parameter*, *variant*, etc.). The classes in [6] cannot be mapped 1-1 to the classes in our work. But still, one can observe that in the distribution of requirements defects according to the work in [6] and in our work are compatible.

The work by Lauesen and Vinter in [4] describes the analysis of requirements defects in two comparatively small requirement specifications for a noise source location system (107 and 94 requirements, compared to over 1600 requirements in our work). There, about 60% of the requirements defects related to unstated demands (i.e., to incompleteness), which is high in comparison to the corresponding number in our analysis (26%).

The idea to refine the classification based on the IEEE 830 standard by considering defect sources is inspired by the Orthogonal Defect Classification (ODC) used by Chillarege *et al.* in [1]. More precisely, our notion of defect source is comparable to the notion of *defect type* in [1]. Instead of using the notion of *defect trigger* in [1], we use the development phase in which the requirements defect was detected (design review, system test, etc.).

Their defect categorisation is based on two groups: the defect type, which is a defect description implied by the eventual correction (e.g. assignment, function, algorithm, documentation), and the defect trigger, which describes the condition the defect surfaced under (e.g. concurrency, timing, boundary conditions). We use the basic ideas of ODC for a deeper analysis of our requirements defects by using the defective part of the requirement is similar to the defect type in ODC.

In contrast with our work on requirements defects, the work by Walia *et al.* in [10] considers the *requirements error* (i.e., the (human) error done while

working on requirements; in contrast, the requirements defect is the manifestation of a requirements error in the requirements specification). The work in [10] uses a classroom experiment in order to classify requirements errors according to, e.g., human failure, process, or documentation error. Our initial attempts to analyse requirements errors for the DC-to-DC project using the classification of [10] were not successful (due to the lack of stability, i.e., the analysis results were not robust under the change of analyst). We leave the analysis of requirements errors in an industrial project to future work.

7 Conclusion and Future Work

We have analysed the set of 588 requirements defects reported in the DC-to-DC project at Bosch with over 1600 system requirements during a lifetime of 4.5 years. We have formulated the insights gained from the results of the analysis and we have used them to review decisions regarding the requirements engineering process at the beginning of the DC-to-DC project and to give recommendations for new decisions.

We have refined the initial classification of requirements defects, which is based in the IEEE 830 standard using the notion of *defect sources*. The resulting classification turned out to be useful tool for the analysis of requirements defects in the DC-to-DC project. It is an interesting topic of future work to evaluate whether this classification is more universally applicable or whether it can be used as the basis of a universally applicable classification of requirements defects.

Acknowledgements We thank Hermann Kaindl for useful discussions that substantially helped to improve the presentation of this paper.

References

1. R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M. Wong. Orthogonal Defect Classification - A Concept for In-Process Measurements. *IEEE Trans. Software Eng.*, 18(11):943–956, 1992.
2. K. E. Emam. Benchmarking Kappa: Interrater Agreement in Software Process Assessments. *Empirical Software Engineering*, 4(2):113–133, 1999.
3. IEEE Computer Society. Software Engineering Standards Committee and IEEE-SA Standards Board. IEEE Recommended Practice for Software Requirements Specifications. Institute of Electrical and Electronics Engineers, 1998.
4. S. Lauesen and O. Vinter. Preventing Requirement Defects: An Experiment in Process Improvement. *Requir. Eng.*, 6(1):37–50, 2001.
5. K. A. Neuendorf. *Content Analysis Guidebook*. Sage Publications, Thousand Oaks, 20002.
6. D. Ott. Defects in natural language requirement specifications at Mercedes-Benz: An investigation using a combination of legacy data and expert opinion. In *2012 20th IEEE International Requirements Engineering Conference (RE), Chicago, IL, USA, September 24-28, 2012*, pages 291–296, 2012.

7. A. Post, J. Hoenicke, and A. Podelski. rt-Inconsistency: A New Property for Real-Time Requirements. In *Fundamental Approaches to Software Engineering - 14th International Conference, FASE 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, pages 34–49, 2011.
8. A. Post, J. Hoenicke, and A. Podelski. Vacuous real-time requirements. In *RE 2011, 19th IEEE International Requirements Engineering Conference, Trento, Italy, August 29 2011 - September 2, 2011*, pages 153–162, 2011.
9. A. Post, I. Menzel, and A. Podelski. Applying Restricted English Grammar on Automotive Requirements - Does it Work? A Case Study. In *Requirements Engineering: Foundation for Software Quality - 17th International Working Conference, REFSQ 2011, Essen, Germany, March 28-30, 2011. Proceedings*, pages 166–180, 2011.
10. G. S. Walia and J. C. Carver. A systematic literature review to identify and classify software requirement errors. *Information & Software Technology*, 51(7):1087–1109, 2009.
11. C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, and B. Regnell. *Experimentation in Software Engineering*. Springer, 2012.