Specifying Feature-Dependent Maintainability Requirements in an Operational Manner - Results From a Case Study with Practitioners

Philipp Haindl and Reinhold Plösch

Institute of Business Informatics - Software Engineering
Johannes Kepler University
Linz, Austria
{philipp.haindl,reinhold.ploesch}@jku.at

Abstract. The TAICOS constraint language allows to express feature-dependent non-functional requirements as quantitative constraints using a compact set of time series operations, time filters, and comparison operators. For each of these constraints, thresholds can be defined on feature-level using metrical and ordinal scales. The fulfillment of these constraints can be automatically evaluated throughout the engineering cycle and shall support data-driven decision making for improving software quality on feature-level.
In this paper we present the results of an empirical case study with 14 practitioners who formulated maintainability requirements using the TAICOS constraint language for different features in an operational manner. After formulating the constraints we asked the practitioners to rate scope, expressiveness, and suitability of the constraint language and the respective practical benefits and weaknesses of the language. Also, we condensed nine recurrent maintainability aspects from the interviews and analyzed which time series operations, filters, and data types the practitioners used for each maintainability aspect.
The case study reveals that the constraint language is expressive, suitable, and its scope fully sufficient to specify maintainability requirements on feature-level. Also, we observed that particularly the company specific modularization of software features has an impact on the suitability of the constraint language. The practitioners positively highlighted the ease of use, compactness, and understandability of the constraint language and remarked that additional requirements engineering support is essential to effectively utilize all language capabilities.

Keywords: Constraint Language · Maintainability Requirements · Quality Engineering · Quality Constraint Specification

CEUR-WS.org/Vol-2725/paper17.pdf

1 Introduction

Software features not only need to fulfill different functional but also different qualitative requirements, mainly depending on the features' usage contexts and engineering-related considerations of the manufacturer. As an example, the

Copyright © 2020 for this paper by its authors.
Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

maintainability of a feature will likely be more important for a manufacturer if the respective feature is frequently used by its customers, plays a vital role in an important software product or is a core feature in a software product line. Likewise, in a mobile sports tracking app the feature calculating the runner's pace will demand high performance characteristics as even small performance lags lead to incorrect calculation of the pace. Contrarily, the feature which uploads the running route data after the run to a server can have more relaxed performance requirements since performance lags do not negatively affect this feature's functionality. In many software projects often the most important threshold of a non-functional requirement (NFR) is singled out among all features and defined as the *common threshold* that must be met by all features. Consequently this can lead to increased engineering effort to meet this common threshold for features which could also suitably provide their functionality with a more relaxed threshold. For instance, a software manufacturer might select the number of code smells as an important indicator of maintainability and that shall be low.

To address this problem we presented the TAICOS quality model [11] that allows to specify NFRs on the level of individual features and introduced the concept of *constraints* to evaluate the fulfillment of feature-dependent NFRs quantitatively. In this context, we understand a feature as a distinct functional unit of work in a software system that satisfies a corresponding functional requirement. We also presented an operational constraint language and a concrete runtime framework [12] for it which allows to automatically evaluate the fulfillment of feature-dependent NFRs in DevOps. The language defines the operationalization of quantitative measures used in the constraints and concrete evaluation criteria using individual thresholds for each feature. Particularly to ensure the suitability of the constraint language in practical settings, it is vital that the language is easily comprehensible for practitioners, provides the required language elements for their purposes, and is sufficiently expressive to let them specify NFRs and evaluation criteria for individual features.

This paper complements our previous work in this context and provides an empirical validation of the constraint language in a case study with practitioners. We particularly asked practitioners from different companies and industry sectors to use the constraint language to specify maintainability requirements specific for their company in an *operational manner*. They were asked to define the *operational acquisition* of measures for their maintainability requirements as well as formulate concrete constraints which allow an *operational evaluation* of these measures. Afterwards we conducted interviews with the participants to examine their experiences.

The remainder of this paper is organized as follows. We give an overview of related work and the demarcation of our approach to other works in Section 2. In Section 3 we describe the research context and the study design to answer the research questions. Following, in Section 4 we succinctly describe our constraint language before we present the results from our case study in Section 5. We outline possible threats to the validity of these results in Section 6, before concluding our paper and sketching relevant future work in Section 7.

2 Related Work

We separate the related work in this field into two streams of research. The first stream covers studies examining specific quality factors of domain-specific languages (DSLs). Haugen et al. [13] presented a structured questionnaire that assesses three dimensions of DSLs - expressiveness, transparency, and formalization. Merilinna and Parssinen [19] examined benefits when using DSL approaches in comparison with traditional approaches. In their work the authors particularly investigated the differences in user abstraction when using either of these approaches. Kosar et al. [16] compared program understanding between general-purpose programming languages and DSL approaches using a cognitive dimension framework and showed that program understanding is 15% better for DSL approaches. Hermans et al. [14] performed an empirical case study using questionnaires to identify six success factors of DSLs (e.g., learnability, usability, expressiveness, reusability, development costs, and reliability). Johanson and Hasselbring [15] elaborated on an empirical study that examines the potential benefits of DSL approaches over general-purpose languages. The study was conducted with domain experts who were not familiar with programming and they were asked to perform representative tasks of their domain with a general programming language and a DSL likewise. The validation quantitatively compared the effectiveness and efficiency of both approaches and showed the DSL approach being more accurate and requiring less time of the domain experts.

The second stream of relevant research pertains to empirical studies specifically examining the usability of DSLs. Concretely, the usability of a DSL is the quality that makes it easy for users to understand, learn, and apply it [2,4,17]. Several studies [4,7,8,17,20,21] have shown that any difficulties in this context might lead to higher maintenance effort of the textual artefacts developed with this DSL. To evaluate the usability of DSLs, Albuquerque et al. [2] proposed an approach based on the CDN (Cognitive Dimensions of Notations) framework [6]. The CDN framework defines several cognitive dimensions (e.g., consistency and error-proneness) to thoroughly evaluate the usability of a written notation such as DSLs. Most importantly, their research identified two main cognitive dimensions that predominantly influence the usability of a DSL - expressiveness and conciseness. In the CDN framework both dimensions are again subdivided into eight subdimensions in total, which in fact allows a very detailed evaluation of DSL usability for each subdimension. Barišić et al. [5] proposed an usability evaluation method for DSLs that complies with iterative user-centered evaluation practices. This makes their approach especially applicable to repeatedly evaluate a DSL and to resolve any usability insufficiencies early in the development cycle. With the USE-ME framework, Barišić et al. [3] presented a dedicated framework for DSL usability evaluation. It allows to model the context, goals, and evaluation procedure of a DSL or tool under study in a formal way. For the validation the authors conducted a case study in which participants needed to apply the framework, followed by an interview subsequently. The results showed the correctness of the method also under limited time and understandability and overall satisfaction with applying the framework from the participant's perspective.

In this paper we present an empirical validation of several quality characteristics of the TAICOS constraint language. These quality characteristics have been identified by other authors as being relevant for the practical applicability, usability, and suitability of DSLs, such as constraint languages. In our case study we let practitioners apply the constraint language to formulate maintainability requirements and thereby concretely examined (a) scope and expressiveness, (b) suitability, (c) the context-dependent usage of language elements, and (d) benefits and weaknesses of the constraint language.

3 Research Context and Study Design

The research presented in this paper complements our previous works [11,12] in the context of specifying, measuring, and evaluating feature-dependent NFRs and originated from an industrial cooperation with a large-scale software manufacturer. Given this industrial context, the guiding principle behind this research was on developing a constraint language that is easily understandable and can be applied by the different roles in industrial software projects. While we observed the quantitative runtime characteristics of the different language elements in a previous study [12], in this study we solely focus on the qualitative aspects of the TAICOS language elements from a user perspective.

DSLs intend to provide language elements for expressing the requirements of a specific problem domain, using the concepts suitable for that particular domain [18]. In the context of this study, this means assessing whether the TAICOS constraint language supports the concepts that are suitable from a user perspective to express maintainability requirements on feature-level. A particular benefit of DSLs is that they help their users raise the level of abstraction up to a level that is suitable to solve their problems more effectively than general-purpose languages, which in turn more actively and productively engages users in the software development process [2]. Obviously, the degree that the TAICOS constraint language fulfills these aspects can only empirically be answered with the help of domain experts that concretely apply the language for their maintainability requirements. As suggested by Fabre et al. [9], an empirical validation of DSLs shall take qualitative data, e.g., users' feedback after applying the language, but also quantitative data, e.g., context-dependent usage of language elements to detect typical usage patterns, into account.

3.1 Case Study Design

To gather post-usage quantitative and qualitative feedback about the constraint language from practitioners, we designed our case study to consist of three consecutive parts - an *introductory*, a *practical*, and an *interview* part.

Introductory Part. At the beginning, we gave the practitioners a succinct introduction into the TAICOS constraint language, i.e., the overall objective of the language and the individual purpose and functionality of each language element (cf. Section 4). Concretely we showed them how the TAICOS language

allows to define the *acquisition* of measures for NFRs using instruments, select appropriate data types for measures, and to refine the *evaluation* of measures as constraints on feature-level. The practitioners were asked to raise questions during this presentations in case of any difficulties of understanding.

Practical Part. Following, the practitioners had to select three features of a single software product in their companies, whereby it was important that the selected features differ in their qualitative requirements, e.g., that individual maintainability requirements were slightly higher for one feature and more relaxed for another. We then asked the practitioners to define five maintainability requirements in a qualitative manner typical for their companies and relevant for these features. An illustrative maintainability requirement for this task would be that the number of code smells must be low.

The practitioners then used these qualitative maintainability requirements to define quantatitive measures and specify constraints for each feature in an operational manner. To accomplish this task they needed to (a) define an *instrument* to acquire a measure, (b) select a *data type* to hold this measure in a variable, (c) decide whether a maintainability requirement needs to regard the historical development of this measure and thus requires a *times series operation* and *time filter*, (d) select a *comparison operator* and define a *threshold* for this constraint. We explicitly did not help the practitioners in formulating the constraints in any way, but provided them the introductory slides as these already contained all necessary information to solve that task. The formulation of constraints was repeated until each resulting constraint was formally correct.

Interview Part. For the interviews after the practical part we designed a questionnaire based on the guidelines of Runeson and Höst [22]. As suggested by Yin [24] we also conducted a pilot interview with one highly experienced expert and included the gathered feedback to improve the questionnaire itself. As a result from the pilot interview we refined certain rating scores of language elements to assure a clear understanding of the answering possibilities.

The questionnaire comprised 10 open questions and 13 closed questions on a 4-point Likert scale and was separated into three parts: The first part captured industry sector of the participant's company, roles in projects, and years of experience with software engineering. In the second part, the participants rated different quality aspects such as suitability, expressiveness, and scope of the individual language elements questions. The open questions additionally intended to examine benefits, challenges, drawbacks, and practical considerations regarding suitability and applicability of the constraint language in the companies of the practitioners. Finally, in the third part, we showed the practitioners how constraint evaluation results of a popular messenger app are visualized on feature-level using the TAICOS web application. The answers to these questions serve as preliminary feedback for the continuing improvement of this application and will be presented in another paper.

We conducted 14 face-to-face interviews with software testers, quality managers, software architects and senior software engineers, DevOps engineers, and

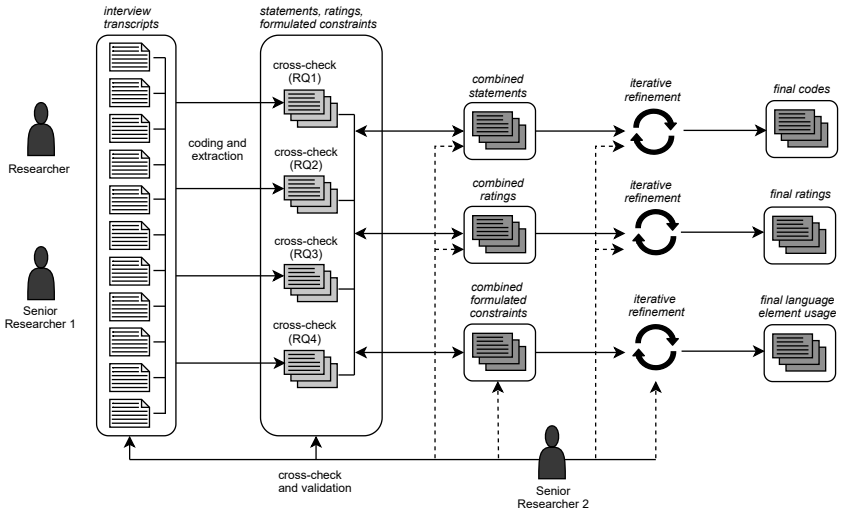

Fig. 1: Research process for iteratively deriving final codes, ratings, and usage frequency of individual language elements (adapted from [23]).

software development training experts from different companies in Austria and Germany. The participants' companies have a median employee count of 900 and ranged from small enterprises to companies with more than 100000 employees. Industry sectors comprise public administration, banking, commerce, software consulting, property management, electronic engineering, automotive, and software product manufacturing. The participants have an average of 14 years professional experience with software engineering and each participant typically helds multiple roles in the projects. The primary roles of the participants are software development (33%), software architecture (28%), test and quality management (17%), technology training (11%), and software R&D (11%).

3.2 Research Questions

The case study seeks to answer the following four research questions:

- **RQ1: Are scope and expressiveness of the TAICOS constraint language appropriate to specify feature-dependent maintainability requirements?** We refined this research question into two subquestions.
  - *RQ1.1: Are the individual constraint language elements sufficiently expressive to define feature-dependent maintainability requirements?* This question focuses on the expressiveness, i.e., the semantic preciseness and clarity, of the language elements for the user.
  - *RQ1.2: Is the scope of the individual constraint language elements appropriate to specify feature-dependent maintainability requirements?* This question examines whether the language elements appropriately cover the required functionality to express maintainability requirements.

- **RQ2: Is the TAICOS constraint language suitable to specify feature-dependent maintainability requirements?** While the previous research question focused on the individual language elements, this research question asks whether the combination of the different language elements is suitable for expressing maintainability requirements.
- **RQ3: What are typical usage patterns of the TAICOS constraint language depending on the maintainability aspect addressed in a constraint?** This analysis is particularly important for further improving the constraint language as it makes the users' implicit understanding of the language and possible deficiencies, such as missing language elements, more tangible [9].
- **RQ4: What are benefits and weaknesses of using the TAICOS constraint language for specifying feature-dependent maintainability requirements?** This question seeks to identify the benefits, challenges, and weak spots of the constraint language and shows limitations of the language and sketches possible directions for its further improvement.

3.3 Data Analysis Procedure

Each interview took between 90 and 120 minutes and was conducted and transcribed by one researcher. Following, the interview transcripts were analyzed by two researchers and the answers to the closed questions codified quantitatively. The answers to the open questions where coded in the transcripts but no qualitative data extracted at that point. We waited to categorize the statements until all interviews were conducted to ensure that we do not exclude potentially relevant qualitative data from the start. Figure 1 shows the research process we followed for the data analysis of the interviews and the formulated constraints.

Then the qualitative statements were categorized by two researchers conjointly following a grounded theory [1,10] approach. The objective of this step was to assign each statement meaningfully to a category and we iteratively refined these categories and combined similiar statements whenever appropriate. The refinement and coding was repeated until consensus was reached among the two researchers. In addition, a second senior researcher monitored the process and cross-checked the coded transcripts but was not involved in the transcription of the interviews itself.

We coded the formulated constraints solely quantitatively, whereby we extracted the concrete language elements, their metadata, and the maintainability aspect that the constraint was formulated for. Similar to the answers to the open questions we refined the extracted maintainability aspects iteratively until consensus was reached among the two researchers.

In total, we extracted 397 statements (28 on average per interview) from the 14 interviews. After several iterations we condensed 43 categories out of the qualitative statements. Finally, we identified nine recurring maintainability aspects from the formulated constraints. Our aggregated findings from analyzing the interviews and the formulated constraints are presented in Section 5.

4 TAICOS Constraint Language

In this section we give a short introduction to the TAICOS constraint language to the extent necessary for the further understanding of the paper. For a more comprehensive presentation of the language's grammar and syntax we refer to our other publication [12]. Basically, the TAICOS constraint language is a DSL for specifying feature-dependent NFRs in an operational manner using quantitative measures and fulfillment criteria. The language is conceptually based on the extended QUAMOCO meta quality model [11] and distinguishes between *features*, for which individual *constraints* can be defined, and *instruments*, which acquire *measures* from external systems and make them accessible as *variables* in the constraints. *Interests* reflect coarse-grained qualitative objectives that must be taken into account by all features, e.g., that the number of code smells must be low. These objectives are refined on feature-level as quantifiable constraints, i.e., with a concrete threshold of acceptable code smells for each individual feature. The number of code smells can be directly acquired through an instrument that queries this measure from a dedicated code quality server, e.g., SonarQube. To also evaluate the development of a measure over time, the TAICOS constraint language provides time filters and basic operations for time series.

4.1 Language Elements

Following, we elaborate the relevant elements of the TAICOS constraint language used in this case study in more detail. Also, we highlight important `keywords` and `language examples` in the text that shall support its better understanding.

Instruments. Represent connectors to external systems that retrieve a specific measure from an external system using predefined software interfaces. To integrate further measures that are relevant for specifying NFRs, the set of instruments can easily be extended by implementing certain software interfaces.

Variables and Data Types. The value retrieved from an instrument can be assigned to a variable. When declaring a variable, the user also defines its data type and the corresponding instrument. The language allows three data types for variables: `measure` for numerical metrics, `rule` for the number of rule violations, and `rating` for 1-letter quality ratings on a scale from A to E. The type of a variable also imposes restrictions on the use of time series operations and comparison operators. Time series operations are only possible for numerical data, i.e., for variables with a declared type `measure` or `rule`. Also, the comparison of a variable's value with a threshold value inevitably requires that both values are of the same type.

The name of a variable can be arbitrarily chosen and also be reused for different constraints and features. This allows to use them very flexible similar to a programming language. For instance, one could declare a variable `codeSmells` for the above example as `codeSmells as measure from Sonar("codeSmells")`. This expresses that the numeric variable `codeSmells` is acquired from the SonarQube instrument, which accesses it using the key `codeSmells` from that system.

Time Series Operations and Time Filters. Particularly to analyze the development of a measure over time, our language provides five time series operations: `min`, `max`, `median`, `avg`, and the (linear regression) `gradient` of the time series. The relevant time frame to obtain the time series of a measure is specified by time filters. Our constraint language provides the three time filters `days`, `weeks`, `months` with a numerical parameter indicating the number of time units to go backwards in time.

Constraints. A constraint refines how a specific interest can be individually regarded by a feature. It defines quantitative fulfillment criteria so that the declared variable, hereon applied time series operations and time filters, can be compared with a threshold value. We differentiate three types of constraints:

(a) **Constraints comparing the most recent value of a variable with a threshold.** This is the simplest type of constraint and works with any type of variable. The constraint `SearchFriends: cyclomaticComplexity < 8` for instance requires the cyclomatic complexity of the feature `SearchFriends` to be lower than 8.

(b) **Constraints utilizing benchmark results of rule violations.** This constraints evaluates whether the number of rule violations lying within a specific quartile of the benchmark base. It requires a variable of type `rule` and is declared by appending `.benchmark` to the variable name. The expression `undocumentedMthds as rule from Sonar("squid:UndocumentedApi")` defines a variable `undocumentedMthds` to hold the number of undocumented methods. The constraint `SearchFriends: undocumentedMthds.benchmark < Q50` demands that the number of rule violations in the `SearchFriends` feature is lower than in 50% of the benchmarked projects.

(c) **Constraints comparing the result of a time series operation with a threshold.** Such constraints can only be expressed for numerical variables, i.e., of type `measure` or `rule`. They require a concrete time series operation and a time `filter` to be specified in the constraint. For instance, the constraint `SearchFriends: median(cyclomaticComplexity, days(7)) < 12` demands that the median cyclomatic complexity of this feature must not have been greater than or equal to 12 in the last 7 days.

Comparison Operators and Thresholds The constraint language supports comparison operators such as <,>,>=,<=, and ==. Thresholds can be specified using metrical values without unit of measurement, e.g., the number of rule violations, or ordinal scaled values, such as e.g., `Q25 - Q100` to express quartiles.

5 Case Study Results

During the practical part of the case study, the participants formulated 222 feature-dependent constraints for their company-specific maintainability requirements with our constraint language. In the following we present the detailed results from our case study sequentially with the research questions.

5.1 Scope and Expressiveness (RQ1)

Figure 2 illustrates that the participants consistently rated the scope of the language elements positively, either being sufficient or fully complete for specifying maintainability requirements on feature-level. In the Likert scale for rating the scope we particularly differentiated between the scores *overcharged* (i.e., not all language elements are relevant), *fully complete* (i.e., no further elements shall be added), *sufficient* (i.e., further non-critical language elements need to be added for more complex requirements), and *not sufficient* (i.e., critical language elements are missing). No participant rated the scope of any language element to not be sufficient for specifying maintainability requirements on feature-level.

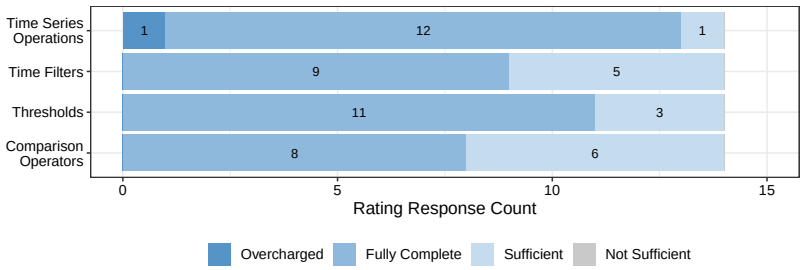

Fig. 2: Ratings of the Scope of the Language Elements.

Among the language elements, the scope of time series operations and thresholds were rated by most practitioners as being *fully complete* (12 and 11 ratings respectively). One participant rating the scope of time series operations as *overcharged* argued that *"for rule violations and quality violations in general, average and median operations are questionable, as they must never be any violations. The number of violations always must be zero"*. This participant also mentioned that *"this is not a limitation of the language by itself, since it does not force me to apply these operations in all contexts"*. The one participant rating the scope of time series operations as *sufficient* mentioned the sum operator as important for his projects, which the language currently does not provide. Five participants mentioned that sprints/program increments, time intervals, quartals, hours, and the number of certain events would be further relevant time filters for them. Three participants responded that they would also like to express thresholds using enumerations and intervals. Six participants mentioned boolean, similarity, and set operators as well as operators for expressing *much larger/smaller than* as further relevant comparison operators.

The analysis of the participants' rating of the expressiveness of the language elements draws a similar picture. As shown in Figure 3 the vast majority of participants rated the expressiveness of the elements as *very good*. No participant rated the expressiveness as *not sufficient*. We also observed that participants who rated the scope as *sufficient* and elaborated missing language elements in turn also rated the expressiveness on a lower score.

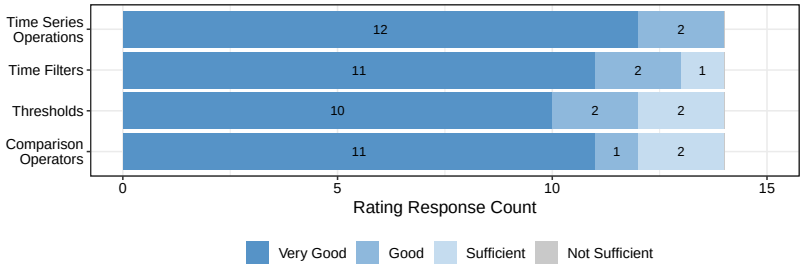

Fig. 3: Ratings of the Expressiveness of the Language Elements.

5.2 Suitability (RQ2)

We also asked the participants to rate the suitability of the TAICOS constraint language on a 4-point Likert scale and to justify their answers. Thereby we intentionally did not differentiate between individual language elements, as the language's suitability cannot be meaningfully answered on this isolated level.

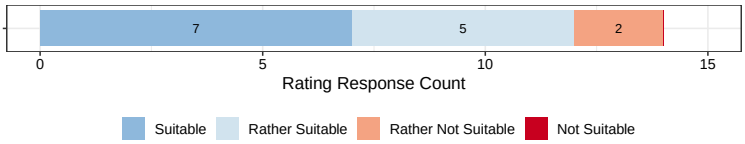

Fig. 4: Suitability of the Constraint Language.

As depicted in Figure 4, a majority of 12 participants positively rated the constraint language as either *suitable* or *rather suitable*. However, two participants rated the language as *rather not suitable*. They argued that due to the modularization of the features in their companies and their product-specific reuse it would be difficult to define meaningful constraints on feature-level. Also they stressed that, given a number of over 400 features in an average software product of their companies, it would be essential for them to rather define general constraints for all features and only exceptions from these general constraints. Though, both participants did not question the value of the constraint language itself, and accentuated that these additional requirements towards the language relate to their industry sectors (automotive and electronic engineering).

5.3 Language Usage Patterns per Maintainability Aspect (RQ3)

To answer this research question, we qualitatively coded the maintainability aspect addressed in a constraint and also extracted the quantitative data which language elements have been used in the constraint. The objective of this research question was to identify the typical usage patterns of the constraint language for the different maintainability aspects. During the final analysis of the data, we then condensed nine maintainability aspects that have repeatedly been

choosen by the participants to formulate constraints for. Concretely we condensed the following maintainability aspects: *technical debt* (54 constraints), *security* ratings and vulnerabilities (6 constraints), *readability* of variables, method and class names (24 constraints), *object-oriented design* (12 constraints), *dependencies* such as cyclic class and external framework dependencies (9 constraints), *correctness* reflected through test code coverage, bug counts, or code assertions (54 constraints), *documentation* (21 constraints), cognitive and cyclomatic *complexity* (24 constraints), and *coding style* (18 constraints).

To allow a meaningful comparison of the language elements, we normalized their usage frequency on the level of subelements, i.e., for each individual data type, time series operation, and time filter.

Usage Patterns of Data Types. In Figure 5 we illustrate which data types have typically been used in constraints to address different maintainability aspects. It can be depicted that the `measure` data type is most intensively used for constraints targeting on *technical debt* and *correctness*. Variables of type `rule` dominate in *documentation* and *coding style* constraints, whereas `rating` variables are primarily used for *technical debt* and *security* constraints. In total, we counted 49 `measure`, 20 `rule`, 6 `rating` usages among all interviews once per maintainability aspect.

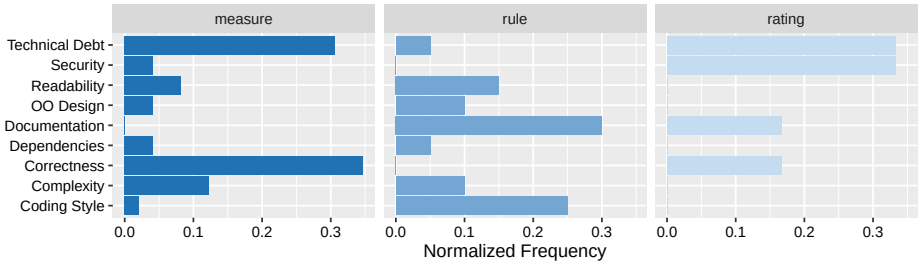

Fig. 5: Usage Patterns of Data Types.

Usage Patterns of Time Series Operations. Figure 6 shows how the participants applied the time series operations for the 9 extracted maintainability aspects. Interestingly, the `min` operation solely was used to address *correctness*, e.g., that a minimum of test coverage must be reached or a certain number of code assertations must be in place. The `max` operations primarily was used to ascertain that the *technical debt* not exceeds a certain threshold. Similarly, the `avg` operation was additionally used for *readability* constraints and to address *correctness* concerns. The `median` operation only was used for two maintainability aspects: *readability* and *correctness*.

Another interesting finding is that the `gradient` operation was evenly used for the majority of maintainability aspects. A similar usage pattern is also noticeable for the `benchmark` operation, which was also evenly used but for a smaller set of maintainability aspects. We also observed that participants never used

time series operations in constraints for *object-oriented design*, but instead for this purpose solely relied on constraints that compare the most recent value of a variable with a threshold (cf. Section 4.1, type a). In total, we counted 5 `min`, 13 `max`, 11 `avg`, 2 `median`, 17 `gradient`, and 5 `benchmark` usages among all interviews once per maintainability aspect.

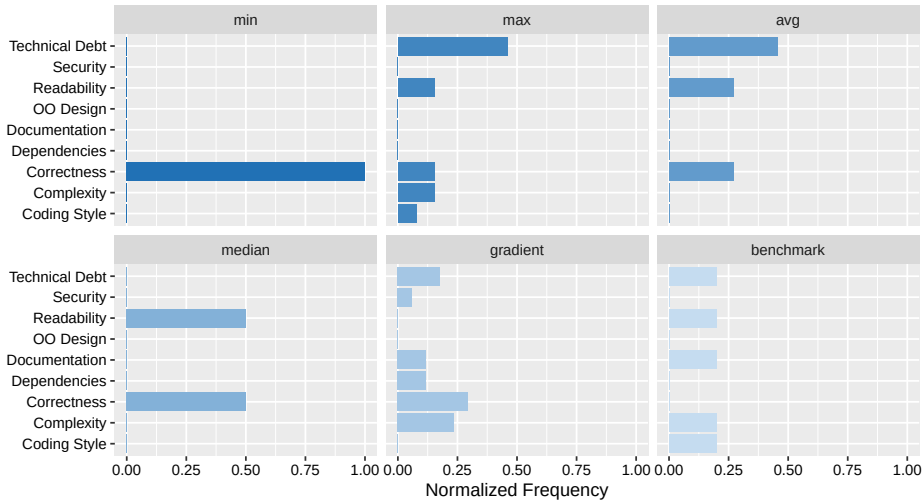

Fig. 6: Usage Patterns of Time Series Operations.

Usage Patterns of Time Filters. Lastly, in Figure 7 we visualize the results from analyzing the usage patterns of time filters in the context of the different maintainability aspects. The `days` filter was intensively used for constraints addressing *correctness* or *technical debt* and in a very similar way as the `weeks` filter. In constrast to these filters, the `months` filter was used more evenly also for other maintainability aspects, but predominantly for constraints addressing *technical debt*, *readability*, and *correctness*. In total, we counted 7 `days`, 20 `weeks`, and 20 `months` usages among all interviews once per maintainability aspect.

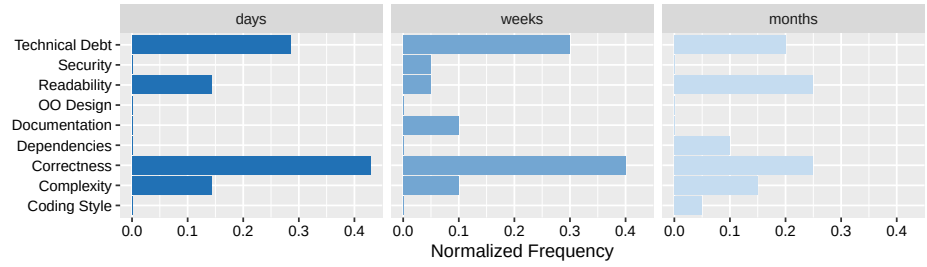

Fig. 7: Usage Patterns of Time Filters.

5.4 Benefits and Weaknesses (RQ4)

Lastly we asked the participants to elaborate on the perceived benefits and weaknesses of the TAICOS constraint language. We qualitatively analyzed their statements and summarize them as follows: Respectively, five participants marked the ease of use and its compactness as benefits of the language, with one emphasizing that *"the language could be applied as it is with little effort"*. Three participants highlighted the understandability and two mentioned the conceptual separation of instrument and constraint as a benefit of the language. They explained that this separation would allow to exchange instruments easily and also makes the source of a variable and its acquisition more explicit for the user of the language.

Asked about the weaknesses of the language, two participants responded that additional methodological requirements engineering support is essential to effectively utilize all language capabilities throughout the product lifecycle. One participant each critized the scope of features, i.e., their definition based on directories and files, the required formal understanding to apply the language, the inability to define default thresholds for features and exceptions, and scalability issues without tool support for large numbers of features and constraints.

6 Threats to Validity

We see a threat to **construct validity** in the different interpretation of the questions by the participants, which is mainly due to their different roles and experiences. We addressed this threat by showing each participant concrete definitions of the terminology and discussed any ambiguities. When summarizing their statements, we also considered background and role of each practitioner to determine from what view and with what intention the statement was given.

The foremost threat to **internal validity** can be seen in the participants' individual understanding of the language elements, which became apparent when they formulated the constraints. We regard this threat as negligible because as soon as we noticed any uncertainties about the language elements we showed their defintion and asked follow-up question to ascertain that the participant understood correctly.

Also, researcher-biased judgments are possible during data extraction and analysis of the interview transcripts, i.e., which codes are derived and refined thereof. We mitigated this threat by discussing the extraction and refinement of the codes among both researchers until we agreed on a solid set of codes.

We addressed the threat to **external validity** by selecting experts who operate in different companies and industry sectors, and also selected two experts maximum per company. However, we see a threat to the generalizability of the results to other industries due to their different maintainability requirements.

7 Conclusion and Future Work

In this paper we presented the results of an empirical case study with 14 practitioners to examine the scope, expressiveness, and suitability, benefits, and

weaknesses of the TAICOS constraint language for specifying feature-dependent maintainability requirements. The practitioners specified these maintainability requirements in an operational manner, i.e., the operational acquisition of maintainability measures as well as operational evaluation criteria on feature-level.

In total, the participants formulated 222 feature-dependent maintainability constraints. The majority of participants positively rated the scope of the language elements as *fully complete* to specify feature-dependent maintainability requirements, with no rating as *not sufficient*. Similarly, the vast majority of participants rated the expressiveness of the language elements as *very good*. Overall, 12 participants rated the language as *suitable* or *rather suitable*. Two participants argued that due to the modularization and product-specific reuse of features in their companies, the language is *rather not suitable* in their companies, i.e., automotive and electronic engineering.

We also condensed nine recurring maintainability aspects from the formulated constraints and analyzed the usage patterns of the language elements depending on the addressed maintainability aspect. The participants predominantly declared metrical variables of type `measure`, particularly to specify constraints evaluating the correctness of features. When analyzing time series of measures, participants frequently used the `gradient` operation evenly for all aspects and the `avg` operation particulary for evaluating technical debt. Participants preferably selected the `weeks` or `months` time filter to specify time frames, particulary in the context of correctness, readability, and technical debt.

Future work shall specially concentrate on developing advanced time filters, set-based comparison operators, tool support for large constraint specifications and on providing methodological requirements engineering support for feature-dependent requirements elicitation and constraint specification.

References

1. Adolph, S., Hall, W., Kruchten, P.: Using grounded theory to study the experience of software development. Empirical Software Engineering **16**(4), 487–513 (2011).
2. Albuquerque, D., Cafeo, B., Garcia, A., Barbosa, S., Abrahão, S., Ribeiro, A.: Quantifying usability of domain-specific languages: An empirical study on software maintenance. Journal of Systems and Software **101**, 245–259 (2015).
3. Barišić, A., Amaral, V., Goulão, M.: Usability driven DSL development with USE-ME. Computer Languages, Systems & Structures **51**, 118–157 (2018).
4. Barišić, A., Amaral, V., Goulão, M., Barroca, B.: Quality in use of DSLs: current evaluation methods (2011)
5. Barišić, A., Amaral, V., Goulão, M., Barroca, B.: Evaluating the Usability of Domain-Specific Languages Information. In: Software Design and Development: Concepts, Methodologies, Tools, and Applications, pp. 2120–2141. IGI Global (2014)
6. Blackwell, A.F., Britton, C., Cox, A., Green, T.R.G., Gurr, C., Kadoda, G., Kutar, M.S., Loomes, M., Nehaniv, C.L., Petre, M., Roast, C., Roe, C., Wong, A., Young, R.M.: Cognitive Dimensions of Notations: Design Tools for Cognitive Technology. In: Beynon, M., Nehaniv, C.L., Dautenhahn, K. (eds.) Cognitive Technology: Instruments of Mind. pp. 325–341. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2001)

7. Deursen, A., Klint, P.: Little languages: little maintenance? Technical Report, CWI (Centre for Mathematics and Computer Science), NLD (1997)
8. van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: an annotated bibliography. ACM SIGPLAN Notices **35**(6), 26–36 (2000).
9. Favre, J.M., Gasevic, D., Lämmel, R., Pek, E.: Empirical Language Analysis in Software Linguistics. In: Malloy, B., Staab, S., van den Brand, M. (eds.) Software Language Engineering. pp. 316–326. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2011)
10. Glaser, B., Strauss, A.: The Discovery of Grounded Theory: Strategies for Qualitative Research. Routledge, New Brunswick (2000)
11. Haindl, P., Plösch, R., Körner, C.: An Extension of the QUAMOCO Quality Model to Specify and Evaluate Feature-Dependent Non-Functional Requirements. In: 2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA). pp. 19–28. IEEE, Kallithea-Chalkidiki, Greece (2019).
12. Haindl, P., Plösch, R., Körner, C.: An Operational Constraint Language To Evaluate Feature-Dependent Non-Functional Requirements. In: 2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA). accepted for publication. IEEE, Portorož, Slovenia (2020)
13. Haugen, y., Mohagheghi, P.: A multi-dimensional framework for characterizing domain specific languages. In: Proceeding of the 7th OOPSLA Workshop on Domain Specific Modeling (2007)
14. Hermans, F., Pinzger, M., van Deursen, A.: Domain-Specific Languages in Practice: A User Study on the Success Factors. In: Schürr, A., Selic, B. (eds.) Model Driven Engineering Languages and Systems. pp. 423–437. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2009)
15. Johanson, A.N., Hasselbring, W.: Effectiveness and efficiency of a domain-specific language for high-performance marine ecosystem simulation: a controlled experiment. Empirical Software Engineering **22**(4), 2206–2236 (2017).
16. Kosar, T., Oliveira, N., Mernik, M., João, M., Pereira, M., Repinšek, M., Cruz, D., Rangel Henriques, P.: Comparing General-Purpose and Domain-Specific Languages: An Empirical Study. Computer Science & Information Systems **438** (2010).
17. Langlois, B., Jitia, C., Jouenne, E.: DSL Classification (2007)
18. Le, D., Dang, D., Nguyen, V.: On domain driven design using annotation-based domain specific language. Comp. Lang., Systems & Structures **54**, 199–235 (2018).
19. Merilinna, J., Parssinen, J.: Comparison between different abstraction level programming: experiment definition and initial results. Montreal, Canada (2007)
20. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Computing Surveys **37**(4), 316–344 (2005).
21. Nishino, H.: How can a DSL for expert end-users be designed for better usability? a case study in computer music. In: CHI '12 Extended Abstracts on Human Factors in Computing Systems. pp. 2673–2678. CHI EA '12, Association for Computing Machinery, Austin, Texas, USA (2012).
22. Runeson, P., Höst, M.: Guidelines for conducting and reporting case study research in software engineering. Empirical Software Engineering **14**(2), 131–164 (2009).
23. Vierhauser, M., Rabiser, R., Grünbacher, P.: A Case Study on Testing, Commissioning, and Operation of Very-large-scale Software Systems. In: Companion Proceedings of the 36th International Conference on Software Engineering. pp. 125–134. ICSE Companion 2014, ACM, New York, NY, USA (2014).
24. Yin, R.K.: Case Study Research and Applications: Design and Methods. SAGE Publications, Inc, Los Angeles, 6th edn. (2017)