A Domain-specific Language for Configurable Traceability Analysis

Hendrik Bünder[1], Christoph Rieger[2] and Herbert Kuchen[2]
[1]*itemis AG, Bonn, Germany*
[2]*ERCIS, University of Münster, Münster, Germany*

Keywords: Traceability, Domain-specific Language, Software Metrics.

Abstract: In safety-critical industries such as the aviation industry or the medical industry traceability is required by law and specific regulations. In addition, process models such as CMMI require traceability information for documentation purposes. Although creating and maintaing so-called traceability information models (TIM) takes a lot of effort, its potential for reporting development progress, supporting project management, and measuring software quality often remains untapped. The domain-specific language presented in this paper builds on an existing traceability solution and allows to define queries, metrics, and rules for company- or project-specific usage. The basis for such an analysis is a query expression to retrieve information from a TIM. Customizable metrics are then defined to compute aggregated values, which are evaluated against company- or project-specific thresholds using the rules part of the domain-specific language. The focus of this paper is to show how the combination of query, metric, and rule expressions is used to define and compute customizable analyses based on individual requirements.

1 INTRODUCTION

Traceability is the ability to describe and follow an artifact and all its linked artifacts through its whole life in forward and backward direction (Gotel and Finkelstein, 1994). Many companies create traceability information models for their software development activities either because they are obligated by regulations (Cleland-Huang et al., 2014) or because it is prescribed by process maturity models. However, there is a lack of support for the analysis of these models (Bouillon et al., 2013).

Recent research describes how to define and query traceability information models, (Maletic and Collard, 2009; Mäder and Cleland-Huang, 2013). This is an essential prerequisite for retrieving specific trace information from a TIM. However, far too little attention has been paid to taking advantage of further processing of the gathered trace information. In particular, information retrieved from a TIM can be aggregated in order to support software development and project management activities with a "real-time" overview of the current state of development.

On the other hand, research has been done on defining relevant metrics for TIMs (Rempel and Mäder, 2015), but the data collection process is non-configurable. As a result, potential analyses are limited to predefined questions and cannot provide comprehensive answers to ad hoc or recurring information demands. For example, projects using an iterative software development approach might be interested in the accomplishments of objectives within each development phase, whereas other projects might focus on a comprehensive documentation during the process of creating and modifying software artifacts.

The approach presented in this paper fills the gap between those two areas by introducing a sophisticated analysis language. As a foundation, query expressions can be used to retrieve information from TIMs and subsequent metric statements aggregate the results of an executed query. In addition, rule expressions can be specified so that metric values can be checked against individually configured thresholds. All three parts come with an interpreter implementation so that they cannot only be defined but also executed against a traceability information model. The analysis language builds on a traceability meta model that is an instance of the Eclipse Ecore model (Steinberg et al., 2008).

This paper contributes a domain-specific traceability analysis language to define queries, metrics and rules in a fully configurable and integrated way. Further, the feasibility of the elaborated analysis language will be demonstrated by a prototypical in-

BÃijnder H., Rieger C. and Kuchen H.
A Domain-specific Language for Configurable Traceability Analysis.
DOI: 10.5220/0006138503740381
In *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2017)*, pages 374-381
ISBN: 978-989-758-210-3

terpreter implementation for real-time evaluation of those trace analyses.

Having discussed related work in Section 2, Section 3 introduces the query and metric expressions that are used to retrieve information from TIMs. Afterwards, the definition of rules is presented, which completes the approach with a mechanism to compare metric values with predefined thresholds. In Section 4, DSL and our prototypical implementation are discussed before the paper concludes in Section 5.

2 RELATED WORK

Requirement traceability is essential for verifying the progress and completeness of a software implementation (Völter, 2013). While in the aviation or medical industry traceability is prescribed by law (Cleland-Huang et al., 2014), there are also process maturity models requesting a certain level of traceability (Cleland-Huang et al., 2012b). Traceable artifacts such as *requirement*, *unit of code*, or *test case*, and the links between those - such as *specifies*, *implements*, and *verifies* - constitute the TIM (Mader et al., 2013).

In contrast to the efforts made to create and maintain a TIM, only a fraction of practitioners takes advantage of the inherent information according to recent research (Cleland-Huang et al., 2014). However, Rempel and Mäder (Rempel and Mäder, 2015) showed that the number of related requirements or the average distance between related requirements have a positive correlation with the number of defects associated with these requirements. In addition, empirical data shows that traceability models also facilitate maintenance tasks and the evolution of software systems (Mäder and Egyed, 2015).

Due to the lack of sophisticated tool support, the opportunities discussed above are often missed (Bouillon et al., 2013). In contrast to the highly configurable traceability information models, traceability tools such as IBM Rational DOORS (IBM, 2016) just offer a predefined set of evaluations, often with simple tree or matrix views (Schwarz, 2012). As a consequence, especially company- or project-specific information regarding software quality and project progress cannot be retrieved and thus remains unused.

This paper introduces a textual domain-specific language (Mernik et al., 2005) that is focused on describing customized query, metric, and rule expressions in the domain of software traceability. The language is implemented using the Xtext framework that is part of the Eclipse ecosystem. Based on a grammar in EBNF-like format, a parser and an Eclipse Ecore Model, representing the meta model of the language, are generated (Bettini, 2013; Völter, 2013).

The IDE generated by the Xtext language workbench provides extensible features such as syntax highlighting, live validation, code completion, and automatic formatting (The Eclipse Foundation, 2016c). Additionally, Xtext validates references between concrete model elements that are available in the so-called scope of the current language element. The syntactically and semantically valid elements are determined by the configurable scope provider as part of the Xtext framework and may contain elements of different Ecore models (The Eclipse Foundation, 2016c).

3 DEFINING AND INTEGRATING THE DOMAIN-SPECIFIC LANGUAGE

3.1 Composition of Modeling Layers

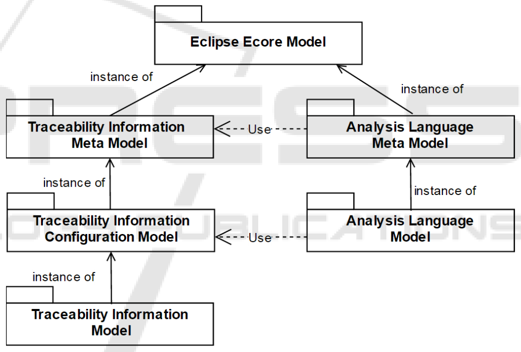
Eclipse Ecore Model
instance of
instance of
Traceability Information Meta Model
Use
Analysis Language Meta Model
instance of
instance of
Traceability Information Configuration Model
Use
Analysis Language Model
instance of
Traceability Information Model

Figure 1: Conceptual Integration of Model Layers.

Figure 1 shows the integration between the different model layers referred to in this paper, starting with the *Eclipse Ecore Model* as a shared meta meta model. The Xtext framework used to define the analysis language generates an instance of this model (The Eclipse Foundation, 2016c), representing the *Analysis Language Meta Model* (ALMM). Individual queries, metrics, and rules are specified within a concrete instance, the *Analysis Language Model* (ALM), by using the developed syntax.

Likewise, the *Traceability Information Model* used in this paper contains the actual traceability information, for example the concrete *Requirement "RQ1"*. It is again an instance of a formal abstract description, the so called *Traceability Information Configuration Model* (TICM). The TICM describes traceable artifact types, e.g. *Requirement* or *Java file*, and the available link types, e.g. *implements*. This model itself is based on a proprietary *Traceability Information Meta Model* (TIMM) that defines the basic constructs such as a traceable artifact type and a traceable link type by inheriting basic EClass and EReference elements of the Eclipse Ecore Model (Gronback, 2009).

Since the analysis language is related to the Eclipse Ecore Model, concepts such as EClass definitions can be referenced. Further references between concepts on the meta model layer (ALMM using TIMM) are the prerequisite for subsequently referencing instances on the concrete model layer. For instance, a query definition of the ALM could at some point reference the traceable artifact type "Requirement" in the TICM. On the model layer this reference is established by referring to a concrete instance of the traceable type, e.g. the result of the specific query "JavaClassesForRequirement" references "RQ1".

To structure the DSL, the analysis language itself is hierarchically subdivided into three components, namely the query, metric, and rule expressions. In order to establish clear interfaces, only query expressions may reference elements of the traceability information configuration model. Metric and rule expressions are built on top of query expressions and are thus independent from TICM and TIM. Live evaluation is performed by the query interpreter accessing the query expression's AST and applying the respective computation to the concrete TIM. The result of such a query execution (as elaborated in the following subsections) is then used by the interpreter to evaluate the metric and rule expressions.

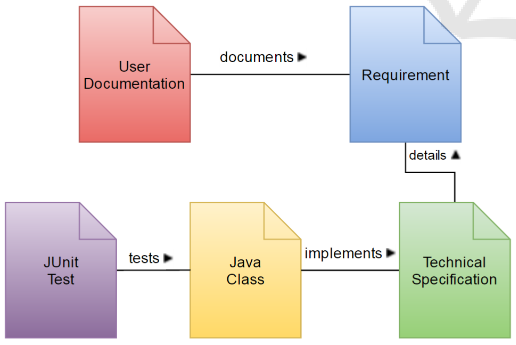

Figure 2: Traceability Information Configuration Model.

Figure 2 shows an example traceability information configuration model that defines the traceable artifacts and link types. The arrowheads in Figure 2 represent the primary trace link direction. However, trace links can be traversed in both directions (Cleland-Huang et al., 2012a). Besides being the abstract description for the TIM instances, the configuration model artifacts can be referenced within query language expressions.

3.2 Querying the Traceability Information Model

The query expressions are part of the analysis language which is defined as a textual domain-specific language using the Xtext framework.

```
query "tracesFromRequirementToJUnitTest"
traceFromTo(Requirement,JUnitTest) as paths
    .collect(paths.getStart.getName as name,
            count(1) as testCases)
    .groupBy(paths.getStart.getName)
```

Listing 1: Sample query definition.

Listing 1 shows an example query that retrieves the shortest path between each instance of *Requirement* and *JUnitTest* artifacts from a given TIM. The query definition starts with the keyword *query* that is followed by the actual name as string literal. The center of the query is the function *traceFromTo* that takes a source and a target traceable artifact as parameters. The available traceable artifacts are determined by the evaluation of the traceability information configuration model of the TIM. The function itself encapsulates an algorithm to find the shortest path between all instances of the two specified configuration model artifacts in the TIM. All resulting paths are assigned to the variable *paths* that is subsequently used to access these within the query. Each path object in the list offers several convenience functions such as `getStart` that returns the start element of the trace or `getEnd` that returns the last artifact of the trace.

The result of an executed query expression as well as the result of a metric or rule expression is returned as a tabular structure. For computing the query expression's result, the query interpreter iterates through all list entries in the variable *paths* that contain the results of the *traceFromTo* function called before. For every entry in the variable *paths* a new row in the tabular structure is created. The column definition is introduced by the keyword *collect*. For example in Listing 1, two column headings are defined called *name* and *testCases*, each introduced by the keyword *as*. The first part of the column definition is the reference to the variable *paths* that represent a list. The next part is introduced by "." which is followed by the functions available on a single entry of the path list. After entering ".", the Xtext proposal provider (The Eclipse Foundation, 2016c) will propose all functions available on each list element. Introduced by another ".", further method calls will be proposed based on the return type of the former function call. The second column specifies an aggregation function that counts all entries in a given column per row. Based on the column index passed as a parameter to the *count* function, the number of entries in each row of that column is counted. In general, the result of this function will be 1 per row since there is only one value per row and column but in combination with the *groupBy* function the number of aggregated values per cell is computed. The *groupBy* function of Listing 1 aggregates all start artifacts ("Requirement") with the same name.

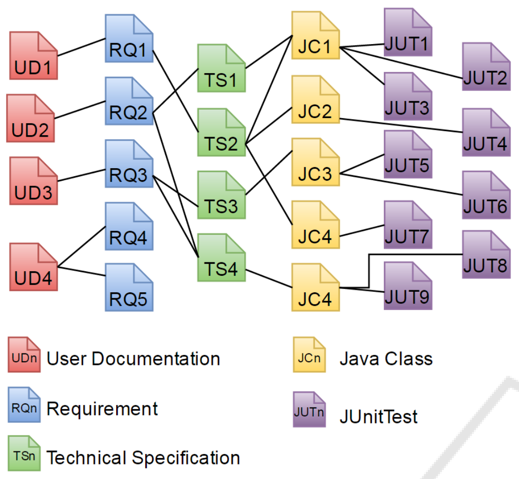

Figure 3: Traceability Information Model.

While queries can be expressed without an instance of the configuration model, the execution is done in a traceability information model. The query expression of Listing 1 is applied to the example TIM shown in Figure 3 by the query interpreter. The result is a tabular structure with three rows and two columns where the first row contains the name "RQ1" in column one and the number "5" as count of the available traces in column two, for example. Even though there are five *Requirement* artifacts in the given TIM, only three of them are linked to a *JUnitTest* artifact so that there are three rows in the tabular result description.

The query expressions offer a powerful and well-integrated mechanism to retrieve information from a given TIM. At this point the data retrieved from the TIM can be understood as raw data that needs to be further processed to make use of it. In the following, it is explained how query expression results can be aggregated to metrics that represent information on the quality and progress of a project.

3.3 Defining Individual Metrics

A software quality metric can be defined as a "function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which software possesses a given attribute that affects its quality" (Society, 2004). A well-known and simple software metric is *lines of code* (Riguzzi, 1996), but there are also specific metrics for traceability models. These include, e.g., the average distance between requirements, the coverage between two levels of specification or full-depth from requirement to the lowest level, and linkage statistics concerning the count of related higher/lower level trace artifacts (Rita J. Costello and Dar-Biau Liu, 1995; Rosenberg et al., 1998). Complimentary to recent research that focuses on specific traceability metrics and their relevance (Rempel and Mäder, 2015), the approach described in this paper introduces an analysis language to define individual metrics. By utilizing this approach, a project team or a quality department can establish and evolve custom metrics until they meet their specific requirements.

An Xtext grammar in EBNF-like format defines the available features including referencing queries, arithmetic operations, and operator precedence using parentheses. The metrics grammar of the analysis language itself has two main components. One is the result declaration that encapsulates the result of a previously specified query[1]. The other is an arbitrary number of metrics definitions that may aggregate query results or other metrics recursively.

```
MetricDefinition:
  'metric' name=ID '=' expression=
      MetricsExpression;

PlusOrMinus returns MetricsExpression:
  MulOrDiv(({Plus.left=current} '+' |
      {Minus.left=current} '-') right=MulOrDiv)*;

MulOrDiv returns MetricsExpression:
  MetricAtomic(({MulOrDiv.left=current} op=('*'|'/
      ')) right=MetricAtomic)*;

MetricAtomic returns MetricsExpression:
  '('MetricsExpression')' |
  {MetricsRef} metric=[MetricsDefinition]|
  {ColumnSelection} col=ColumnSelection
  {DoubleConstant} value=DOUBLE |
  {SumFunction} sum= SumFunction |
  {CountFunction} count=CountFunction;
```

Listing 2: Grammar rules for metric expressions.

Listing 2 shows the most important rules that describe the possible arithmetic expressions used in metrics. Since the corresponding parser generated by ANTLR works top-down, the grammar must not be left recursive (Bettini, 2013). Listing 2 shows the grammar to support the four basic arithmetic op-

[1]The formal description of the syntax of a query is quite lengthy and extends beyond the scope of this paper, in which we focus on the metrics and rules language. From the example in subsection 3.2, the reader gets an impression of what a query looks like.

erations as well as the correct use of parentheses. While the *PlusOrMinus* and *MulOrDiv* rules to enable ANTLR to handle a left recursive grammar are well described by Bettini (2013), the *MetricAtomic* rule contains the essential computations of the metrics grammar. First, the rule allows for the usage of constant double values. Second, metric expressions can contain pre-defined functions to sum up or count the results of a query. Third, columns from the result of a query can be referenced so that metric expressions per query expression result row can be computed. Finally, metric expressions can refer to other metric expressions to further aggregate already accumulated values as shown in the last line of Listing 6.

```
SumFunction:
  'sum' '(' columns+=ColumnSelection
       (',' columns+=ColumnSelection)* ')';
```

Listing 3: Rule for sum aggregation.

Exemplary for the predefined aggregation functions, the *SumFunction* rule shown in Listing 3 describes the syntax for summing up the results of a query. After the keyword *sum*, any number of *ColumnSelections* can be stated as parameters. To compute the result, the interpreter will iterate over all the rows of the query expression's result and sum up the values of the referenced columns. Listing 4 shows the actual *ColumnSelection* that is also used by the *CountFunction* and refers to the query expression's result columns. While a concrete metric is specified, the available columns are proposed in the editor based on the result declarations in the current scope. In addition, there is a live validation of the metric expressions that displays error markers as soon as a referenced result declaration or a column within the query expression result is no longer in scope.

```
ColumnSelection:
  resultDeclaration=[ResultDeclaration|ID] '.'
       column= [analysis::Column];
```

Listing 4: Rule for column selection.

Using the given grammar, reusable metric expressions can be defined to compute the number of related requirements (NRR) as described by Rempel and Mäder (2015), i.e., the number of directly and indirectly referenced requirements from one requirement to another for all requirement artifacts in the TIM. The basis for this metric are all trace links from the TICM shown in Figure 2 that could be instantiated to describe a trace between two requirements.

```
result relatedRequirements from
traceFromTo(Requirement, Requirement) as paths
.collect(paths.getStart.getName as srcRequirement,
         paths.getEnd.getName as trgtRequirement)

metric NRR=cnt(relatedRequirements.srcRequirement)
```

Listing 5: Metric: number of related requirements.

The inlined query expression in Listing 5 returns a tabular result that contains the shortest path between every pair of requirement artifacts from the given TIM. The path between each *Requirement* artifact can contain multiple other artifacts since the traceable links are bidirectionally navigable. In the example TIM from Figure 3, the trace from "RQ1" to "RQ2" contains the following artifacts "RQ1" to "TS2" to "JC1" to "TS2" to "RQ2". The problem of circular dependencies causing infinite loops in the computation of the available paths is solved by the depth-first algorithm implementation.

The first column of the tabular result structure contains the name of the source requirement in the path while the second column holds the name of the target requirement. The following NRR metric expression uses the *cnt* function to compute the number of related requirements.

Table 1: NRR Metric: Tabular Result Structure.

Requirement	NRR
RQ1	2
RQ2	2
RQ3	2
RQ4	1
RQ5	1

Table 1 shows the tabular result structure of the metric expression executed against the sample TIM from Figure 3. The tabular result structure is made of two columns derived from the query and metric expression as described above.

```
result pathLengthPerRequirement from
traceFromTo(Requirement, Requirement) as paths
.collect(paths.getStart.getName as srcRequirement,
         sum(paths.getLength) as pathLength)
.groupBy(paths.getStart.getName)

metric ADRR= pathLengthPerRequirement.pathLength/
     NRR
```

Listing 6: Composition of metric expressions.

Listing 6 shows the reuse of an existing metric to compute the average distance between two requirements as explained by Rempel and Mäder (2015). The query expression shown in the example returns the shortest path between two requirements already aggregated per source requirement. The subsequent metric takes the *pathLength* column from the query expression result and divides every entry by the already computed metric NRR shown in Listing 5.

The combination of configurable query expressions with configurable metric definitions allows users to define their individual metrics. The analysis language is complemented by the rules grammar, which is described in the following section.

3.4 Evaluating Metrics

A metric value itself delivers few insights to the quality or the progress of a project. However, comparing a metric value to a pre-defined threshold or another metric value exposes information. The grammar contains rules for standard comparison operations which are *equal, not equal, greater than, smaller than, greater or equals*, and *smaller or equals*. A rule expression can either return a warning or an error result that may both be detailed by an individual message. Since query and metrics result descriptions implement the same tabular result interface as described above, rules can be applied to both. Finally, the result of an evaluated rule expression is also stored using the same tabular interface.

```
WarnIf returns RuleSpecification:
'rule' name=ID '=' 'warnIf(' ruleBody=RuleBody')';

RuleBody:
('m:' metric=[metrics::MetricsDefinition]|
 'c:' column=[ResultDeclaration|ID] '.' column=
     [analysis::Column]) compareOperator=Operator
     compareTo=RuleAtomic ',' msg=STRING;
```

Listing 7: Syntax for rule expressions.

The *RuleBody* rule shown in Listing 6 is the central part of the rules grammar. On the left side of the *compareOperator*, a metric expression or a column from a query expression result can be referenced. During definition of the rule, the metric expressions and query result columns available in the current scope are proposed. The next part of the rule is the *comparisonOperator* followed by a *RuleAtomic* value to compare the expression to. The *RuleAtomic* value is either a constant number or a reference to another metric expression. The evaluation of rule, metric and query expressions during run-time is implemented using Xtend, a Java extension developed as part of the Xtext framework and especially designed to navigate and interact with the analysis languages Eclipse Ecore models (The Eclipse Foundation, 2016b).

```
rule checkADRR=warnIf(m: ADRR >4.0, "High average
    distance between related requirements")
```

Listing 8: Sample rule definition.

The metric specified in Listing 6 is referenced by an instance of the warning grammar rule shown in Listing 8 and it is compared to the value 4.0. In case that the metric contains a greater value, a warning message is produced and stored in the result object created by the rule interpreter. In other cases there will be an automatically produced message stating that the metric value is as expected. To create a staggered analysis, a warning and an error message for the same metric expression can be defined, thus classifying the result in two categories with varying severity. For example, an *ADRR* value greater than 4.0 causes a warning while an *ADRR* value higher than 6.0 causes an error message. The rule interpreter will recognize that there are two rule expressions based on the same metric and will only return one result.

The result of an interpreted rule expression contains not only information about the compared metric values but can also provide a meaningful warning or error message. All in all, the rule grammar finalizes the analysis language capabilities by providing mechanisms to compare aggregated information from a TIM against custom thresholds.

4 DISCUSSION

To demonstrate the feasibility of the designed analysis language and perform flexible evaluations of traceability information models, a prototype was developed. The analysis language is based on the aforementioned Xtext framework and integrated into Eclipse using its plug-in infrastructure (The Eclipse Foundation, 2016a). In addition, an interpreter was implemented that evaluates query, metric, and rule expressions ad hoc whenever the respective expression is modified and saved. Currently, both components are tentatively integrated in a software solution that envisages a commercial application. Therefore, the analysis language is configured to utilize a proprietary TIMM from which traceability information configuration models and concrete TIMs are defined.

Within our implementation, traceable artifacts from custom traceability information configuration models as shown in Figure 2 can be used for query, metric, and rule definitions. Due to an efficient implementation of the depth-first algorithm used by the *traceFromTo* function, queries are (re-)executed immediately when a query is saved. The efficiency of the depth-first algorithm implementation was verified by interpreting expressions using TIMs ranging from 1,000 to 50,000 traceable artifacts.

Table 2 shows the duration for interpreting the analysis expression from Listing 1 against generated

Table 2: Duration of Analysis.

Total Artifacts	Start Artifacts	Duration (in s)
1,000	300	0.012
8,000	1,500	0.1
50,000	8,500	2.2

TIMs of different sizes. The first column shows the overall number of traceable artifacts and links in the TIM. The second column displays the number of start artifacts for the depth-first algorithm implementation, i.e. the number of "Requirement" artifacts for the exemplary analysis expression. The third column contains the execution time on a computer with Intel Core i7-4700MQ processor at 2.4 GHz and 16 GB RAM. As shown, executing expressions can be done efficiently even for large size models.

Defining and evaluating analysis statements with the prototypical implementation has shown that the approach is feasible to collect metrics for different kinds of traceability projects. The analysis language can for example be utilized to create company-specific metrics. Within the same industry sector some companies use a model-driven approach, others apply test-driven development, or directly start coding from a textual requirement. When a company uses an entity DSL to describe the data model of the application, it could be valuable to compute the average number of attributes per entity, assuming that a high number indicates bad design. In case of a company deciding to directly code from a textual requirement, a metric to calculate the number of classes per requirement in relation to the number of words in the requirement definition might be reasonable to assess its specificity. If there is meaningful information in such company-specific metrics, the company initially needs to discover them. However, the more important finding is that company-specific metrics can be created, prototyped, and evolved easily by employing the analysis language.

Since the analysis language is based on a highly configurable TIMM, it allows for a large variety of traceable artifacts, including formerly unused documents such as documentation, test results or even tickets from collaboration tools (Delater and Paech, 2013). Including artifacts from different systems, metrics can also be used to indicate the quality or the progress of a certain software product. A metric to measure the quality of a software component could compute the number of defects related to a specific unit of code by traversing the TIM to find all links between those two artifacts. Using a project-specific rule to highlight code units causing a high number of defects gives an indication on where to perform quality measures, e.g. code reviews. The current progress of a software development project can be exposed by defining a staggered analysis. Taking the exemplary TICM from Figure 2, a first query could find all *Requirement* artifacts that are not linked to a *Technical Specification* artifact. From a project management perspective the design of these requirements could be understood as not started. The next part of the staggered analysis could retrieve the *Requirement* artifacts that are linked to a *Technical Specification* artifact, but have no trace to a *Java Class* artifact, therefore indicating that the implementation has not yet begun. Relating the described query expression results to the overall number of *Requirement* artifacts measures the project's progress with regard to different phases of development.

The approach presented in this paper is bound to limitations. In particular, an investigation of real world projects is pending in order to assess the impact of the developed DSL on software quality management practices.

In addition, any analysis is only as good as the underlying traceability information model, thus requiring wary treatment of metrics results. For example, we discovered missing trace links through our queries in a preliminary analysis of a real world TIM, as those were configured in the TICM but never instantiated in the TIM. On the positive side, analysis statements are usually defined and executed by domain experts, so that problems with the underlying data can often be identified and resolved quickly.

These are, however, no inherent limitations of the approach but rather constitute future work in deploying the DSL in real-world scenarios that benefit from traceability metrics.

5 CONCLUSION

The DSL described in this paper offers functions and expressions to analyse existing TIMs, structured in query, metric, and rule component. For retrieving traceable artifacts and trace links, query expressions can be defined. In a subsequent metric expression, the results of an interpreted query are condensed to a comparable value using arithmetic operations and aggregate functions. In order to assess this value in context, rule expressions are defined to compare metrics' values among each other or against a threshold value.

The introduced approach closes the gap between information retrieval, metrics definition, and result evaluation, thus forming a solid foundation for project- or company-specific metrics. Regarding flexibility, a configuration model makes it completely independent from the specific type of traced artifacts. Further, it is well integrated into an established workbench and development environment using Xtext and the Eclipse Modeling Framework. Features such as live validation and error markers detect broken or outdated expressions early and ensure a rich user experience.

The focus of future work should be on identifying new industry-relevant metrics by applying the proposed approach to real-world projects. Also, the data mining field offers statistical methods through association rules or regression algorithms to find patterns and gain insights from large data sources such as traceability models.

To sum up, the analysis language proposed in this paper offers an integrated approach to close the gap between querying traceability information models and defining configurable metric expressions. The concept of ad hoc evaluation of expressions was demonstrated in a prototypical implementation.

REFERENCES

Bettini, L. (2013). *Implementing domain-specific languages with Xtext and Xtend*. Community experience distilled. Packt Pub, Birmingham, UK.

Bouillon, E., Mäder, P., and Philippow, I. (2013). A survey on usage scenarios for requirements traceability in practice. *Lecture Notes in Computer Science*, 7830 LNCS:158–173.

Cleland-Huang, J., Gotel, O., Huffman Hayes, J., Mäder, P., and Zisman, A. (2014). Software traceability: Trends and future directions. In *Proceedings of the on Future of Software Engineering*, FOSE 2014, pages 55–69, New York, NY, USA. ACM.

Cleland-Huang, J., Gotel, O., and Zisman, A., editors (2012a). *Software and Systems Traceability*. Springer London, London.

Cleland-Huang, J., Heimdahl, M., Huffman Hayes, J., Lutz, R., and Maeder, P. (2012b). Trace queries for safety requirements in high assurance systems. *Lecture Notes in Computer Science*, 7195 LNCS:179–193.

Delater, A. and Paech, B. (2013). Analyzing the tracing of requirements and source code during software development. In *Requirements Engineering: Foundation for Software Quality*, pages 308–314. Springer Berlin Heidelberg.

Gotel, O. and Finkelstein, A. (1994). Analysis of the requirements traceability problem. *International Conference on Requirements Engineering*.

Gronback, R. C. (2009). *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley Professional, 1st edition.

IBM (2016). Rational doors. www.ibm.com/software/products/en/ratidoor.

Mäder, P. and Cleland-Huang, J. (2013). A visual language for modeling and executing traceability queries. *Software and Systems Modeling*, 12(3):537–553.

Mäder, P. and Egyed, A. (2015). Do developers benefit from requirements traceability when evolving and maintaining a software system? *Empirical Softw. Eng.*, 20(2):413–441.

Mader, P., Gotel, O., and Philippow, I. (2013). Getting back to basics: Promoting the use of a traceability information model in practice. *7th International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE)*, pages 21–25.

Maletic, J. I. and Collard, M. L. (2009). Tql: A query language to support traceability. *Proceedings of the 2009 ICSE Workshop on Traceability in Emerging Forms of Software Engineering, TEFSE 2009*.

Mernik, M., Heering, J., and Sloane, A. M. (2005). When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344.

Rempel, P. and Mäder, P. (2015). Estimating the implementation risk of requirements in agile software development projects with traceability metrics. In *Requirements Engineering: Foundation for Software Quality*, pages 81–97. Springer International Publishing.

Riguzzi, F. (1996). A survey of software metrics.

Rita J. Costello and Dar-Biau Liu (1995). Metrics for requirements engineering. *Journal of Systems and Software*, 29(1):39–63.

Rosenberg, L., Hammer, T. F., and Huffman, L. L. (1998). Requirements, testing and metrics. In *15th Annual Pacific Northwest Software Quality Conference*.

Schwarz, H. (2012). *Universal traceability*. Logos Verlag Berlin, [Place of publication not identified].

Society, I. C. (2004). IEEE standard for a software quality metrics methodology: IEEE std 1061-1998 (r2004). Technical report, IEEE Computer Society.

Steinberg, D., Budinsky, F., Merks, E., and Paternostro, M. (2008). *EMF: Eclipse Modeling Framework*. Pearson Education.

The Eclipse Foundation (2016a). PDE/user guide. http://wiki.eclipse.org/PDE/User_Guide.

The Eclipse Foundation (2016b). Xtend modernized java. http://www.eclipse.org/xtend/.

The Eclipse Foundation (2016c). Xtext documentation. https://eclipse.org/Xtext/documentation/.

Völter, M. (2013). *DSL engineering: Designing, implementing and using domain-specific languages*. CreateSpace Independent Publishing Platform.