A Model-Driven Approach for Configurable Evaluation of Traceability Information

Hendrik Bünder

itemis AG,
Bonn, Germany
Email: buender@itemis.de

***Abstract*—Requirement traceability is the ability to explicate and pursue the relations between all artifacts that specify, implement, test, or document a software solution. Besides being required by laws and regulations in safety-critical industries, the traceability information models can give insight on project progress and quality. Yet, only a few companies are utilizing this competitive advantage due to missing tool support. The paper introduces an integrated solution to define and execute company- or project-specific analysis statements written in a dedicated domain-specific language. First, the capabilities of the Traceability Analysis Language are demonstrated by defining coverage, impact and consistency analysis. Every analysis is defined as a rule expression that compares a customizable metric's value (aggregated from the traceability information model) against an individual threshold. The focus of the Traceability Analysis Language is to make the definition and execution of information aggregation and evaluation from a traceability information model configurable and thereby allow users to define their own analyses based on their regulatory, project-specific, or individual needs. Further, the analyses are applied to a model according to the Automotive Software Process Improvement and Capability Determination (A-SPICE) standard. Second, the underlying grammar, as well as the mechanisms to make data retrieval configurable, are explained. Finally, the paper reports on case study findings at a tier one automotive supplier company. The case study revealed that the possibility to introduce custom data retrieval functions is crucial in real-world scenarios. Further, the case study showed that the traceability analysis language supported the tier one automotive supplier in the process of being A-SPICE re-certified.**

Keywords–Traceability; Domain-Specific Language; Software Metrics; Model-driven Software Development; Xtext.

I. INTRODUCTION

The paper builds upon previous work [1] and elaborates on the specification of the query language, additional configuration options, and extended case study results. While the initial contribution was focusing on configurable analysis expressions utilizing the rule, metric and grammar language, this works extends the approach by introducing custom data retrieval functions. Additionally, the paper elaborates on the grammar of the query language. Finally, detailed findings from the case study at a tier one automotive supplier are explained. In addition to analyzing the runtime behavior of the analysis statements, the results of the case study include a detailed analysis of the different user groups, their information needs, and how the traceability analysis language (TAL) was utilized to satisfy these needs.

Traceability is the ability to describe and follow an artifact and all its linked artifacts through its whole life in forward and backward direction [2]. Although many companies create traceability information models for their software development activities either because they are obligated by regulations [3] or because it is prescribed by process maturity models, there is a lack of support for the analysis of such models [4].

On the one hand, recent research describes how to define and query traceability information models [5][6]. This is an essential prerequisite for retrieving specific trace information from a Traceability Information Model. However, far too little attention has been paid to taking advantage of further processing the gathered trace information. In particular, information retrieved from a traceability information model (TIM) can be aggregated in order to support software development and project management activities with a real-time overview of the state of development.

On the other hand, research has been done on defining relevant metrics for TIMs [7], but the corresponding 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 needs. For example, projects using an iterative software development approach might be interested in the achievement of objectives within each development phase, whereas other projects might focus on a comprehensive documentation along the process of creating and modifying software artifacts.

The approach presented in this paper fills the gap between both areas by introducing the Traceability Analysis Language. By defining coverage, impact, and consistency analyses for a model based on the Automotive Software Process Improvement and Capability Determination standard. Use cases for the Traceability Analysis Language features are exemplified. Analyses are specified as rule expressions that compare individual metrics to specified thresholds. The underlying metrics values are computed by evaluating metrics expressions that offer functionalities to aggregate results of a query statement.

The TAL comes with an interpreter implementation for each part of the language, so that rule, metric, and query expressions cannot only be defined, but can also be executed against a traceability information model. More specifically, the analysis language is based on a traceability metamodel defining the abstract artifact types that are relevant within the development process. All TAL expressions therefore target the structural characteristics of the TIM.

In addition to elaborating on potential use cases for the TAL, the paper reports on first industrial experience. The case study was executed at a tier one automotive supplier where the TAL was used in 5 projects. Further, the users where categorized in three groups and their specific traceability information needs were analyzed. The TAL, its interpreter, and the graphical user

interface integration were tested with these user groups to meet their information requirements.

The contributions of this paper are threefold: first, it provides a domain-specific Traceability Analysis Language to define rules, metrics, and queries in a fully configurable and integrated way. Second, it demonstrates the feasibility of this work with a prototypical interpreter implementation for real-time evaluation of those trace analyses. In addition, it illustrates the TAL's capabilities in the context of the A-SPICE standard and reports on extended results from a case study in the automotive sector.

Having discussed related work in Section II, Section III presents the capabilities of the TAL by exemplifying impact, coverage, and consistency analyses. In Section IV the underlying grammar for rule, metrics, and query definitions and their expected runtime behavior are explained. Section V reports on extended findings from a case study conducted at a tier one automotive supplier. In Section VI, the language, the prototypical implementation, and case study results are discussed before the paper concludes in Section VII.

II. Related Work

Requirements traceability is essential for the verification of the progress and completeness of a software implementation [8]. While, e.g., in the aviation or medical industry traceability is prescribed by law [3], there are also process maturity models requesting a certain level of traceability [9].

Traceable artifacts such as *Software Requirement*, *Software Unit*, or *Test Specification*, and the links between those such as *details*, *implements*, and *tests* constitute the TIM [10]. Retrieving traceability information and establishing a TIM is beyond the scope of this paper and approaches for standardization such as [11] have already been researched.

In contrast to the high effort that is made to create and maintain a TIM, only a fraction of practitioners takes advantage of the inherent information [3]. However, Rempel and Mäder (2015) have shown that the number of related requirements or the average distance between related requirements have a positive correlation with the number of defects associated with this requirement. Traceability models not only ease maintenance tasks and the evolution of software systems [12] but can also support analyses in diverse fields of software engineering such as development practices, product quality, or productivity [13]. In addition, other model-driven domains, such as variability management in software product lines, benefit from traceability information [14].

Due to the lack of sophisticated tool support, these opportunities are often missed [4]. On the one hand, query languages for TIMs have been researched extensively, including Traceability Query Language (TQL) [5], Visual Trace Modeling Language (VTML) [6], and Traceability Representation Language (TRL) [15]. On the other hand, traceability tools mostly offer a predefined set of evaluations, often with simple tree or matrix views, e.g., [16]. Hence, especially company- or project-specific information regarding software quality and project progress cannot be retrieved and remains unused.

Our approach integrates both fields of research using a textual domain-specific language DSL [17] that is focused on describing customized rule, metric and query expressions. In contrast to the Traceability Metamodelling Language [18] defining a domain-specific configuration of traceable artifacts, the work builds on a model regarding the specification of type-safe expressions and for deriving the scope of available elements from concrete TIM instances.

III. An Integrated Traceability Analysis Language

The capabilities of the TAL will be demonstrated by defining analyses from the categories of coverage, impact and consistency analysis as introduced by the A-SPICE standard [19]. In addition to these rather static analyses, there are also traceability analyses focusing on data mining techniques as introduced by [13]. Even though some of these could be defined using the introduced domain-specific language, they remain out of scope of this paper.

A. Scenarios for Traceability Analyses

The first scenario focuses on measuring the impact of the alteration of one or more artifacts on the whole system [20]. Recent research has shown that artifacts with a high number of trace links are more likely to cause bugs when they are changed [7]. Moreover, the impact analysis can be a good basis for the estimation of the costs of changing a certain part of the software. This estimation then not only includes the costs of implementing the change itself, but also the effort needed to adjust and test the dependent components [21].

The second scenario appears to be the most common, since many TIM analyses are concerned with verifying that a certain path and subsequently a particular coverage is given, e.g., "*are all requirements covered by a test case*" or "*have all test cases a link to a positive test result*" [4]. In addition to verifying that certain paths are available within a TIM, coverage metrics are mostly concerned with the identification of missing paths [10].

The third use case describes the consistency between traceable artifacts. Besides ensuring that all requirements are implemented, consistency analyses should also ensure that there are no unrequested changes to the implementation [22]. Consistency is generally required between all artifacts within a TIM in accordance to the Traceability Information Configuration Model (TICM), so that all required trace links for the traced artifacts are available [19].

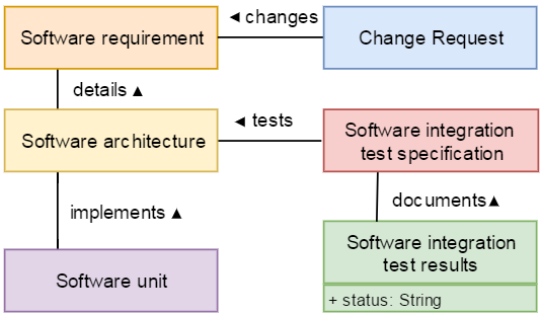

Figure 1. Traceability Information Configuration Model.

Figure 1 shows a simplified TICM based on the A-SPICE standard [19] that defines the traceable artifact types *Change Request, Software Requirement, Software Architecture, Software Unit, Software Integration Test Specification*, and *Software Integration Test Result*. Also, the link types *changes, details, implements, tests*, and *documents* are specified by the configuration model. The arrowheads in Figure 1 represent

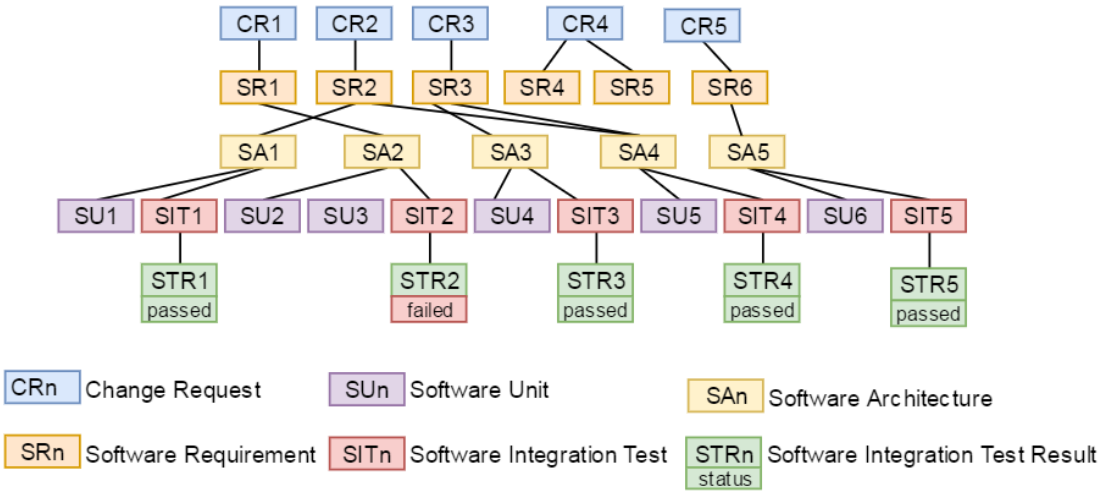

Figure 2. Sample Traceability Information Model.

the primary trace link direction, however, trace links can be traversed in both directions [23]. The traceable artifact *Software Integration Test Result* also defines a customizable attribute called "status" that holds the actual result.

Considering the triad of economic, technical, and social problem space, the flexibility to adapt to existing work practices increases the productivity of a traceability solution [24]. Therefore, configuration models provide the abstract description of traced artifact types in a company context. A TIM captures the concrete artifact representations and their relationships according to such a TICM and constitutes the basis for the analyses (cf. Section IV).

Figure 2 shows a traceability information model based on the sample TICM described above. The TIM contains multiple instances of the classes defined in the TICM that can be understood as proxies of the original artifacts. Those artifacts may be of different format, e.g., Word, Excel or Class files. Within the traceability software, adapters can be configured to parse an artifact's content and create a traceable proxy object in accordance to the TICM. In addition, the underlying traceability software product offers the possibility to enhance the proxy objects with customizable attributes. The *Software Integration Test Result* from Figure 1, for example, holds the actual result of the test case in the customizable attribute "status".

1) Impact Analysis: The impact analysis shown in Figure 3 checks the number of related requirements (NRR) [7] starting from every *Software Requirement* by using the aggregated results of a metric expression, which is based on a query. The analysis begins after the *rule* keyword that is followed by an arbitrary name. The right hand side of the equation specifies the severity of breaking the rule stated in the parentheses. In this case, a rule breach will lead to a warning message with the text in quotation marks. The most important part of the analysis is the comparison part that specifies the threshold, which in this case is a number of related requirements greater than 2. If the metrics' value is greater, the warning message will be returned as a result of the analysis.

The second component of the TAL expression is the metric expression that in this case, counts the related requirements. Each metric is introduced by the keyword *metric*, again followed

```
result relatedReqs =
  tracesFrom Software Requirement to Software Requirement
  collect(start.name ->srcRequirement,
          end.name   -> trgtRequirement)
metric NRR = count(relatedReqs.srcRequirement)
rule NRRWarning = warnIf(NRR>2, "A high number of related
    software requirements could provoke errors.")
```

Figure 3. Metric: Number of related requirements (NRR).

by an arbitrary name, which is used to reference a metric either from another metric or from a rule as shown in Figure 3. The expression uses the *count* function to compute the number of related requirements. The *count* function takes a column reference to count all rows that have the same value in the given column. In the metric expression shown above, all traces from one *Software Requirement* to a *Software Requirement* have the name of the source *Software Requirement* in their first column, so that the *count* function will count all traces per *Software Requirement*. As shown in Table I, the result of the metric evaluation is a tabular data structure with always two columns. The first holds the source artifact and the second column holds the evaluated metric value. For the given example, the first column holds the name of each *Software Requirement* and the second column contains the evaluated number of directly and indirectly referenced *Software Requirement*s.

TABLE I. NRR METRIC: TABULAR RESULT STRUCTURE.

Software Requirement	*NRR*
SR1	0
SR2	2
SR3	2
SR4	1
SR5	1
SR6	1

Finally, the metric is based on a query expression that is used to retrieve information from the underlying TIM. The *tracesFrom... to...* function returns all paths between source and target artifact passed into the function as parameters. In comparison to expressing this statement in other query languages such as Structured Query Language (SQL), no knowledge about the potential paths between the source and target artifacts in the TIM is needed.

Figure 3 shows that the columns of the tabular result structure are defined in the brackets after the keyword *collect*. In the first column the name of the *Software Requirement* of each path is given and in the second column the name of each target *Software Requirement* is given. Both columns can contain the same artifacts multiple times, but the combination of each target with each source artifact is only contained once.

2) Coverage Analysis: Figure 4 shows a coverage analysis that is concerned with the number of related test case results per software requirement. In contrast to the analysis shown in Figure 3, it introduces two new concepts.

```
result tracesSwReqToTestResult =
  tracesFrom Software Requirement
  to         Software Integration Test Result
  collect(start.name-> name, count(1)-> tcrs)
  where(end.status = "passed")
  groupBy(name)
rule lowTC = warnIf(tracesSwReqToTestResult.tcrs < 2,
"Low number of test results!")
rule noTC = errorIf(tracesSwReqToTestResult.tcrs < 1,
  "No test results found!")
```

Figure 4. Software Requirement Test Result Coverage Analysis.

First, the analysis is not dependent on a metric expression, but directly bound to a query result. Since metric and query expression results are returned in the same tabular structure, rules can be applied to both. Second, the analysis shown in Figure 4 demonstrates the concept of a staggered analysis, i.e., one column or metric is referenced once from a warning and error rule, respectively. The rule interpreter will recognize this construct and will return the analysis result with the highest severity, e.g., when the error rule applies, the warning rule message is omitted. The rules shown above ensure that the test of each *Software Requirement* is documented by at least one test result. However, to fulfill the rule completely, each *Software Requirement* should be covered by two *Software Integration Tests* and subsequently two *Software Integration Test Results*.

TABLE II. COVERAGE ANALYSIS: TABULAR RESULT STRUCTURE.

Software Requirement	*Analysis Result*
SR1	No test results found!
SR2	Ok
SR3	Ok
SR4	No test results found!
SR5	No test results found!
SR6	Low number of test results!

Table II shows the result of the staggered analysis. The test coverage analysis returns an "Ok" message for two of the six *Software Requirements*, while one is marked with a warning message and the remaining three caused an error message.

The query expressions result is limited to *Software Integration Test Results* with status "passed" by evaluating the customizable attribute "status" using a *where* clause. Since the query language offers some functions to do basic aggregation, it is possible to bypass metric expressions in this case. In Figure 4 the aggregation is done by the *groupBy* and the *count* function. The second column specifies an aggregation function that counts all entries in a given column per row based on the column name passed as parameter. 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 resulting tabular structure contains one row per *Software Requirement* with the respective name and the cumulated number of traces to different Software Integration Test Results as columns.

3) Consistency Analysis: The following will show two consistency analysis samples to verify that all *Software Requirements* are linked to at least one *Software Unit* and vice versa.

```
result consistetSrcTrgt =
  tracesFrom Software Requirement to Software Unit
  collect(start.name ->name, count(1)-> targets)
  groupBy(name)
rule notCoveredError = errorIf(swUnits<1, "The software
    requirement is not implemented.")
```

Figure 5. Consistency Analysis.

Figure 5 shows a consistency analysis composed of a rule and a query expression. The rule *notCoveredError* returns an error message if the number of traces between *Software Requirements* and *Software Units* is smaller than one, which means that the particular *Software Requirements* is not implemented.

TABLE III. CONSISTENCY ANALYSIS: SOFTWARE REQUIREMENT IMPLEMENTATION.

Name	*Analysis Result*
SR1	Ok
SR2	Ok
SR3	Ok
SR4	The Software Requirement is not implemented!
SR5	The Software Requirement is not implemented!
SR6	Ok

Table III shows the result of the analysis as defined in Figure 5. For "SR4" and "SR5" there is no trace to a *Software Unit* so that the analysis marks these two with an error message. To verify that all implemented *Software Units* are requested by a *Software Requirement*, the query can easily be altered by switching the parameters of the "tracesFrom... to..." function and by changing the error message. Table IV shows the result of the altered analysis revealing that "SU3" despite all others has not been requested.

TABLE IV. CONSISTENCY ANALYSIS: SOFTWARE UNIT REQUESTED.

Name	*Analysis Result*
SU1	Ok
SU2	Ok
SU3	The Software Requirement has not been requested!
SU4	Ok
SU5	Ok
SU6	Ok

These examples show that the language offers extensive support for retrieving and aggregating information in TIMs. The following sections will demonstrate how the TAL integrates with the traceability solution it is build upon, and how the different parts of the language are defined.

IV. COMPOSITION OF THE TRACEABILITY ANALYSIS LANGUAGE

The following focuses on the technical foundations of the TAL. After giving an overview over the modeling layers the paper continues on elaborating the grammar definitions of query, metrics and analysis language.

A. Modeling Layers

Figure 6 shows the integration between the different model layers referred to in this paper, starting from the *Eclipse Ecore Model* as shared meta meta model [25]. The Xtext framework, which is used to define the analysis language generates an instance of this model [26] to represent the *Analysis Language Meta Model* (ALMM). Individual queries, metrics, and rules are specified within a concrete instance, the *Analysis Language Model* (ALM), using the created domain-specific language. An interpreter was implemented using Xtend, a Java extension developed as part of the Xtext framework and specially designed to navigate and interact with the analysis language's Eclipse Ecore models [27].

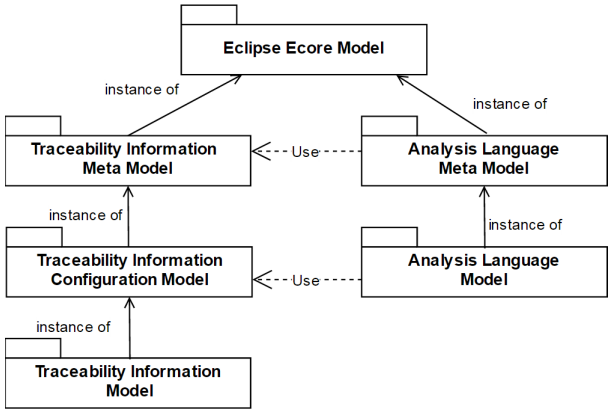

Figure 6. Conceptual Integration of Model Layers.

Likewise, the *Traceability Information Model* used in this paper contains the actual traceability information, for example the concrete software requirement *SR1*. It is again an instance of a formal abstract description, the so called TICM. The TICM describes traceable artifact types, e.g., *Software Requirement* or *Software Architecture*, and the available link types, e.g., *details*. This model itself is based on a proprietary *Traceability Information Meta Model* (TIMM) defining the basic traceability constructs such as an artifact type and link type. To structure the DSL, the TAL itself is hierarchically subdivided into three components, namely rule, metric, and query expressions.

B. Rule Grammar

Since a query result or a metric value alone delivers few insights into the quality or the progress of a project, rule expressions are the main part of the TAL. Only by comparing the metric value to a pre-defined threshold or another metrics' value information is exposed. 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 after executing the comparison including an individual message. Since query and metrics result descriptions implement the same tabular result interface, rules can be applied to both. Accordingly, the result of an evaluated rule expression is also stored using the same tabular interface.

```
WarnIf = ID '=' 'warnIf(' RuleBody ')';
RuleBody = (MetricDefinition | ResultDeclaration '.' Column)
      Operator RuleAtomic ',' MESSAGE;
```

Figure 7. Rule Grammar.

The *RuleBody* rule shown in Figure 7 is the central part of the rule grammar. On the left side of the *Operator* a metric expression or a column from a query expression result can be referenced. The next part of the rule is the comparison *Operator* followed by a *RuleAtomic* value to compare the expression to. The *RuleAtomic* value is either a constant number or a reference to another metrics expression.

C. Metrics Grammar

Complimentary to recent research that focuses on specific traceability metrics and their meaningfulness [7], the approach described in this paper allows for the definition of individual metrics. An extended Backus-Naur form (EBNF)-like Xtext grammar defines the available features including arithmetic operations, operator precedence using parentheses, and the integration of query expressions. The metrics grammar of the TAL itself has two main components. One is the *ResultDeclaration* that encapsulates the result of a previously specified query. The other is an arbitrary number of metrics definitions that may aggregate query results or other metrics recursively.

```
MetricDefinition = 'metric' ID '=' MetricExpression;

MetricExpression = Term  { ('+' | '-') Term};
Term = Factor { ('*' | '/') Factor};
Factor = SumFunction | CountFunction | LengthFunction |
     DOUBLE | ColumnSelection | MetricDefinition | '('
     MetricExpression ')' ;
```

Figure 8. Grammar rules for metrics expressions.

Figure 8 shows a part of the metric grammar defining the support for the basic four arithmetic operations as well as the correct use of parentheses. Since the corresponding parser generated by Another Tool for Language Recognition (ANTLR) works top-down, the grammar must not be left recursive [28]. First, the rule *Factor* allows for the usage of constant double values. Second, metric expressions can contain pre-defined functions such as sum, length, or count to be applied to 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. Thereby, interpreting metric expressions can be modularized to reuse intermediate metrics and to ensure maintainability.

The metrics grammar as part of the TAL defines arithmetic operations that aggregate the results of an interpreted query expression. The combination of a configurable query expressions with configurable metric definitions allows users to define their individual metrics.

D. Query Grammar

The analyses defined using metric and rule expressions depend on the result of a query that retrieves the raw data from the underlying TIM. Although there are many existing query languages available, a proprietary implementation is currently used because of three reasons.

First, the query language should reuse the types from TICM to enable live validation of analyses even before they are executed. The Xtext-based implementation offers easy mechanisms to satisfy this requirement, while others such as SQL are evaluated only at runtime. Second, some of the existing query languages such as SQL or Language Integrated

Query (LINQ) are too verbose (cf. Figure 11) or do not offer predefined functions to query graphs. Finally, other languages such as SEMMLE QL [29] or RASCAL [30] are focused on source code analyses and do not interact well with Eclipse Modeling Framework (EMF) models.

```
Query:
 ('result' ID)? QFeatureCall QCollect? QWhere? QGroupBy?;

QFeatureCall:
ID ( '(' (ID (',' ID)* )? ')')?;

QCollect:
'collect' '('QMemberFeatureCall(',' QMemberFeatureCall)*')';

QWhere:
 'where' '(' QCompareExpression (('AND'|'OR')
     QCompareExpression)* ')';

QGroupBy:
'groupBy' '('QMemberFeatureCall(',' QMemberFeatureCall)*')';

QMemberFeatureCall:
ID ('.' ID)* '->' ID;

QCompareExpression:
ID ('='|'!='|'<'|'>'|'>='|'<=') ID;
```

Figure 9. Query Language Grammar.

In Figure 9, a slightly simplified grammar of the Query language in Extended Backus-Naur Form (EBNF) is shown. The postfix operators "?" and "*" indicate optional and arbitrarily repeated features, respectively. "|" divides alternatives. Terminal symbols appear in single quotes. The grammar first shows the `Query` rule that may contain `QFeatureCall, QCollect, QWhere, QGroupBy` statements.

The `QFeatureCall` rule contains the ID of the function to be called followed by the expected parameters in parenthesis. While the grammar itself looks very simple, the real benefit comes from using the Xtext language workbench. During definition of a query expression, Xtexts ScopeProvider looks for functions that are visibile (or "in scope") for the analysis. The ProposalProvider takes the visible elements and creates a proposal for each function within the editor.

By default, the Traceability Analysis Language comes with pre-defined functions such as `tracesFromTo, artifactsWithoutTraceFromTo, or linkedArtifacts`. In addition, it is possible to implement company-, or project-specific functions. For such a function to be proposed in the editor, a certain interface has to implemented. Further, as Figure 10 shows, annotations are used to classify the function.

```
@FunctionType(priority = 200, type = ARTIFACT)
public Iterable<TVMDArtifact> notLinkedArtifacts(
 @ScopeSource(ALL_ARTIFACT_TYPES) @ScopeType(TYPE_INSTANCE)
     TVMCArtifactType type) {
  return ...
}
```

Figure 10. Sample Function. Returning all not linked artifacts.

First, the annotation in Figure 10 states the `priority` and the `type`. The `priority` parameter manages the execution in case of multiple function calls at a time. The `type` determines, which kind of elements are returned. A function may return instances of elements from the traceability configuration model, such as *Artifact* or *Link*. The latter annotation attribute is evaluated by Scope- and ProposalProvider in order to avoid invalid analysis statements. Second, the method itself has to return a typed iterable that contains the result of the graph traversal. Finally, every parameter of a function has to be detailed in order to enable rich user experience through the calculated scope and proposals. `ScopeSource` specifies if the particular parameter is either from the traceability configuration model or a query result that is further processed. `ScopeType` determines if a meta-model type or an instance of such is returned.

The defined interface, allows for custom functions to be seamlessly integrated into the existing set of query functions. The underlying framework can offer the same editor support as for pre-defined functions, because the custom functions are detailed in terms of parameters, return types, and priority.

After having explained the `QFeatureCall` and its underlying concepts, the following will demonstrate the remaining query grammar. The `QCollect` defines the columns of a query result and contains an arbitrary number of `QMemberFeatureCalls`. Each `QMemberFeatureCall` defines one column by calling methods on the `QFeatureCall` result (cf. Figure 5). After potentially defining a chain of method calls, the result can be assigned to a variable after the keyword —>.

The so defined columns can be re-used in the subsequent `QWhere` and `QGroupBy` statements. The where-clause filters the result of a function called through a `QFeatureCall`. After the keyword *where* an arbitrary number of `QComparisonExpressions` follow that support binary comparison operations. In order to filter based on multiple criteria the comparison statements can be combined with the logical operators `AND` and `OR`. In addition, the result can be further aggregated using `QGroupBy`. Following the keyword *groupBy* a list of attributes is stated as criteria for grouping the result. The query grammar includes many concepts and language support for data retrieval and aggregation. However, more specialized aggregations such as calculating averages or medians the metrics grammar was created.

The query expressions offer a powerful and well-integrated mechanism to retrieve information from a given TIM. Especially, the integration with the traceability information configuration model enables the reuse of already known terms such as the trace artifact type names. Furthermore, complex graph traversals are completely hidden from the user who only specifies the traceable source and target artifact based on the TICM. For example, the concise query of Figure 4 already requires a complex statement when expressed in SQL syntax (cf. Figure 11). If the graph traversal functions included in the TAL are not sufficient, custom functions can be implemented to do specific or more complex data retrieval.

```
SELECT r.name, count(u.id) AS tcrs
FROM SwRequirement r
INNER JOIN SwRequirement_SwArchitecture ra ON r.id=ra.r_id
INNER JOIN SwArchitecture a ON ra.a_id=a.id
INNER JOIN SwArchitecture_SwIntegrationTest ai
  ON a.id=ai.a_id
INNER JOIN SwIntegrationTest i ON ai.i_id=i.id
INNER JOIN SwIntegrationTest_SwIntegrationTestResult it
  ON i.id=it.i_id
INNER JOIN SwIntegrationTestResult t ON it.t_id=t.id
WHERE t.status='passed'
GROUP BY r.name;
```

Figure 11. SQL equivalent to query of Figure 4.

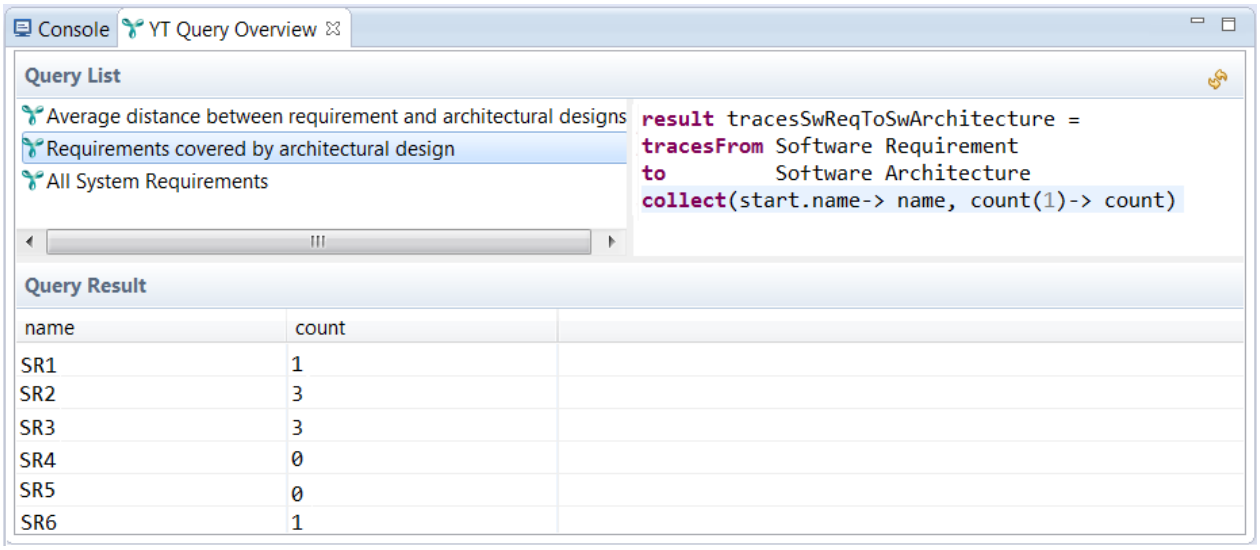

Figure 12. Screenshot of the analysis language interpreter.

E. Performance

Within the prototypical implementation, traceable artifacts from custom traceability information configuration models as shown in Figure 1 can be used for query, metrics, and rule definitions. Due to an efficient implementation used by the *tracesFrom... to...* function, analysis are re-executed immediately when an analysis is saved or can be triggered from a menu entry. The efficiency of the depth-first algorithm implementation was verified by interpreting expressions using TIMs ranging from 1,000 to 50,000 artificially created traceable artifacts. The underlying TICM was build according to the traceable artifact definitions of the A-SPICE standard [19].

TABLE V. DURATION OF TAL EVALUATION.

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

Table V shows the duration for interpreting the analysis expression from Figure 4 against TIMs of different sizes. The first column shows the overall number of traceable artifacts and links in the TIM. The second column gives the number of start artifacts for the depth-first algorithm implementation, i.e., the number of *Software Requirement*s for the exemplary analysis expression. The third column contains the execution time on an 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, sufficient for real-world applications to regular reporting and ad hoc analysis purposes.

The efficient implementation of the depth-first algorithm allows for re-execution on every save. However, it also add the constraints to all custom functions to be highly efficient. Additionally implemented functions with long execution times will have a negative effect on user experience and finally on the TAL at all.

V. CASE STUDY

Besides theoretical usage scenarios for the TAL, first experiences in real-world projects were gained with a tier one automotive supplier. The Traceability Analysis Language was used in five projects with TIMs ranging from 30,000 to 80,000 traceable artifacts defined in accordance to the Automotive SPICE standard.

To demonstrate the feasibility of the designed TAL and perform flexible evaluations of traceability information models, a prototype was developed. The analysis language is based on the aforementioned Xtext framework and integrated in the integrated development environment Eclipse using its plug-in mechanism [31]. The introduced interpreter evaluates rule, metric, and query expressions whenever the respective expression is modified and saved in the editor.

Currently, both components are integrated in Yakindu Traceability (YT) a software solution for creating, maintaining and analyzing traceability information models [32]. Therefore, the analysis language is configured to utilize the proprietary TIMM from which traceability information configuration models and concrete TIMs are defined. At runtime, the expression editor triggers the interpreter to request the current TIM from the underlying software solution and subsequently perform the given analysis.

The Eclipse view in Figure 12 shows the integration of the TAL into YT for creating and maintaining traceability information models. The view consist of three parts that all can be used individually in other views within an Eclipse application. First, the upper left part lists all available analyses in the current workspace. Second, on the upper right side the details of a selected analysis are shown in an embedded editor. The user can edit the expressions and is supported by syntax highlighting, code completion, and live validations. Finally, the bottom half of the view contains a dynamic table that lists the results from an executed analysis language expression. In Figure 12, the column headings “name” and “count” are defined from the query’s *collect* statement. The values in the rows represent the values from the interpreted analysis expression. In this

case the table shows the result of a coverage analysis showing all software requirements and the number of related software architecture artifacts.

At the tier one automotive supplier, multiple project roles used the Eclipse integrated TAL editor. The types of users can be divided into three groups, namely "power user", "direct user" and "indirect user". The main characteristic of power users is that they create queries. These might be used personally or shared with team members. Within the group of power users, there are two project roles involved. On the one hand, the process owners who are responsible for the traceability process within the automotive supplier company. Process owners define analyses that are used as basis for reports and quality gates throughout the process. On the other hand, there are experts from the YT vendor, which do consulting and customization. While process owners create analyses with the given functionality of the TAL, YT experts additionally develop and enhance YT itself. Although functions can be introduced by anyone, the case study has shown that it is done most efficiently by YT experts.

The second user group consists of so called direct users. They differ from power users by only using predefined analyses. Further, the user group consists of more project roles, namely architect, developer, and hardware tester. The case study has revealed that all three project roles have different requirements that need to be covered by the TAL. While architects are mainly concerned with analyzing the coverage between requirements, software architecture, and software units, developers use traceability data to ease development and maintenance tasks. Therefore, impact analyses play a superior role for developers, to estimate the impact of a particular change. Hardware testers have a very strong focus on coverage analysis in order to plan and execute testing tasks.

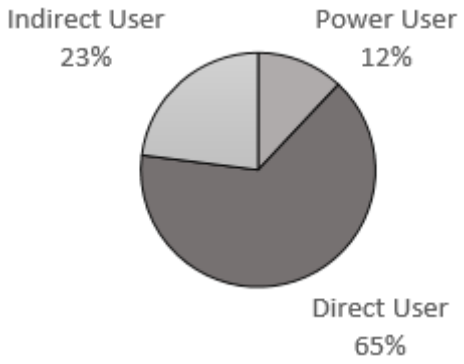

Figure 13. Users per user group.

The third group consists of users that do not interact directly with the TAL. It is therefore called indirect users. In addition to executing analyses within the Eclipse view, the underlying infrastructure is also used to aggregate data on a regular basis. This aggregated data is integrated into reports that are sent to different project roles such as requirements engineers or project managers. While the first two user groups use the TAL for short-term or immediate tasks, indirect users justify medium and long term decisions from the aggregated traceability analyses. Requirements engineers are responsible for a consistent implementation. Thus, they are interested in analyses such as "Are all requirements implemented?". In addition, project managers require information about the overall implementation and testing status of their project.

Within the case study, all three user groups were involved in using the TAL and the introduced editor. Based on the project role, different analyses were defined in order to satisfy the aforementioned information needs. The predefined analyses have replaced complex SQL statements that included up to seven joins to follow the links trough the traceability information model. Because the *tracesFrom... to...* function encapsulates the graph traversal, the analyses are more resilient to changes of the traceability configuration model.

While users were able to collect their data, especially the coverage analyses required by the hardware testers introduced some challenges. In contrast to the rather simple example in Section III-A2, the specification of requirement coverage as used by the tier one automotive supplier is more than two pages long. The extensive definition is due to the following two factors. First, compared to the sample TICM in Figure 1 the TICM contains more artifacts and links between them. Second, there are subsequently more ways to "cover" a requirement. However, not all paths through the traceability information model create valid coverage. Although it would have been possible to define the analysis statement using the existing TAL functionalities, it was decided to implement a custom function. Encapsulating the data aggregation and the majority of rules from the requirements coverage specification in a custom function increased the overall readability and maintainability of the analysis. Introducing a custom function minimized the length of the analysis statement by 85%.

Since other traceability solutions often struggle with performance issues, when it comes to large models, a main focus of the case study was on the runtime of analysis execution. The case study has shown that executing the far more complex real world functions including the custom function took nearly the same time as the aforementioned artificial analysis. Because the functions implement efficient graph traversal algorithms, they can be executed in real time. Therefore, the TAL solution can be integrated seamlessly.

The overall feedback from users involved in the case study was positive. They especially emphasized the easy to learn and powerful query language. Further, the traceability analysis language played an important role during an official A-SPICE Level 3 assessment. By using the TAL, all relevant data could be retrieved and reported in order to pass the assessment [33]. All in all, the TAL was used successfully on a daily basis as well as in a more official setting.

VI. Discussion

The discussion is divided into three parts. It starts with discussing different scenarios, in which the TAL could be applied. The second part elaborates on the findings of the case study executed at a tier one automotive supplier. Finally, the limitations of the introduced approach are explained.

A. Applying the Analysis Language

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 most basic metric expression reads like *the proportion of artifacts of type A that have no trace to artifacts of type*

B. Some generic scenarios focused on impact, coverage, and consistency analyses have been exemplified in Section III-A. However, there are more specific metrics that are applicable and reasonable for a particular industry sector, a specific project organization, or a certain development process as demonstrated by the case study.

In addition to the case study findings, industry-specific metrics, e.g., in the banking sector, could focus on the impact of a certain change request regarding coordination and test effort estimation. Project-specific management rules may for instance highlight components causing a high number of reported defects to indicate where to perform quality measures, e.g., code reviews. Moreover, the current progress of a software development project can be exposed by defining a staggered analysis relating design phase artifacts (e.g., *Software Requirement*s that are not linked to a *Software Architecture*) and implementation artifacts (e.g., *Software Architecture*s without trace to a *Software Unit*) in relation to the overall number of *Software Requirements*. Analysis expressions could also be specific to the software development process. In agile projects for example the velocity of an iteration could be combined with the number of bugs related to the delivered functionality. Thereby, it could be determined whether the number of bugs correlates with the scope of delivered functionality. These use cases emphasize the flexibility of the analysis language — in combination with an adaptable configuration model — for applying traceability analyses to a variety of domains, not necessarily bound to programming or software development in general. For example, a TIM for an academic paper may define traceable artifacts such as *authors*, *chapters*, and *references*. An analysis on such a paper could find all papers that cite a certain author or the average number of citations per chapter. It is therefore possible to execute analyses on other domains with graph-based structures that can benefit from traceability information.

B. Case Study

The case study executed at a tier one automotive supplier revealed that there are different user groups using the TAL. While nearly 80% of the users interact directly with the TAL editor, the remaining 20% use information gathered using the TAL. The reasons that only 12% of the users wrote queries are mainly organizational. On the one hand, traceability analysis results are used internally for reporting the project status or identifying change impacts. On the other hand, a sophisticated analysis capability is an important basis for external process audits such as A-SPICE level 3. In order to ensure high quality, error free queries, only a few well trained employees write and verify analysis definitions. After being verified, these analyses are shared with the different project teams and their results become binding. Although only power users directly use the TAL to define queries, the language statements need to be as readable as possible. Thereby, direct users can quickly understand the purpose of a particular analysis. While users from both groups have stated that TAL is easy to understand, additional research is required to support this claim.

Moreover, the case study has shown that YT experts were faster to implement custom functions. In addition, power users from the tier one automotive supplier showed no interest in writing custom functions. It has to be analyzed whether this is due to the fact that YT experts were available or because the custom function interface is to complex. Moreover, it has to be analyzed, what needs to be done to enable TAL users to specify their own analysis more easily.

In contrast to the aforementioned user groups, the interaction with the traceability information models has not changed for the indirect users. However, the underlying infrastructure to retrieve and aggregate the data has been changed. By introducing the TAL, complex SQL statements (cf. Figure 11) could be replaced by shorter analysis statements. Although the TAL definitions are less verbose, the impacts on the maintainability need to be studied. While the statements become shorter by hiding the complex graph traversal, the traversal algorithms need to be tested thoroughly. If an analysis result is wrong because of an error in the underlying traversal algorithm, failure detection and fixing becomes very difficult. During the early stages of the case study, every TAL analysis was cross-checked manually by a domain expert. The findings from this testing approach were two-fold: first, failures in the algorithm implementation or the analysis expressions were identified and fixed. Second, the TAL definitions revealed errorneous and inconsistent data in the traceability information model.

Since the concept provides that analyses are executed on every save, the underlying algorithms need to be very fast. The case study has shown that the execution times achieved with artificial models could be confirmed within an industry setting. Yet, both traceability configuration models are based on the A-SPICE standard. Therefore, additional research is required with other, more complex TICM.

In general all required information for the different user groups and project roles could be retrieved using the TAL. However, when managing and analyzing multiple versions of an artifact and its combination with others, as explained by software product line engineering, further investigation is required. Yet, the challenges are not solely within the analysis but also in the creation and maintenance of the underlying TIM.

C. Limitations

The approach presented in this paper is bound to limitations regarding both technical and organizational aspects. Regarding the impact of the developed DSL on software quality management practices, first investigations have taken place. However, more are needed to draw sustainable conclusions.

Although all analyses required by the participants could be satisfied, the case study at a tier one automotive supplier has shown that additional analysis capabilities are required. One main requirement is to evaluate, how much of an expected trace path is available in a certain TIM. If there is no complete path from a *System Requirement* to a *Software Integration Test Result*, it would be beneficial to show partial matches and to list missing artifacts such as a missing *Software Integration Test Result* or *Software Integration Test Specification*. Extending the result of an analysis in accordance to this requirement would enhance the information about the progress of a project.

From a language-user perspective, the big advantage of being free to configure any query, metric or rule expression is also a challenge. A language user has to be aware of the traceable artifacts and links in the TIM and how this trace information could be connected to extract reasonable measures. In addition, these analyses are safety critical and therefore need to be implemented and tested by domain experts. Moreover, the

context-dependent choice of suitable metrics in terms of type, number, and thresholds is subject to further research. These limitations do not impede the value of this work, though. In fact, in combination with the discussed application scenarios they provide the foundation for future work.

VII. Conclusion

This work describes a textual domain-specific language to analyze existing traceability information models. The TAL is divided into query, metric, and rule parts that are all implemented with the state-of-the-art framework Xtext. The introduced approach goes beyond existing tool support for querying traceability information models. By closing the gap between information retrieval, metric definition, and result evaluation, the analysis capabilities are solid ground for project- or company-specific metrics. Since the proposed analysis language reuses the artifact-type names from the traceability information configuration model, the expressions are defined using well known terms. Additionally, the newly introduced custom functions ensured short and concise analysis statements.

On the one hand, the introduced approach is based on an Eclipse Ecore model and is thereby completely independent of the specific type of traced artifacts. On the other hand, it is well integrated into an existing TICM and IDE using Xtext and the Eclipse platform. All parts of the TAL are fully configurable regarding analysis expression, limit thresholds, and query statements in an integrated approach to close the gap between *querying* and *analyzing* traceability information models. Subsequently, measures for TIMs can be specific to a certain industry sector, a company, a project or even a role within a project. The scenarios described in Section III-A propose areas, in which configurable analyses provide benefits for project managers, quality managers, and developers. These claims were supported by findings from the conducted case study.

Using the implemented interpreter for real-time execution of expressions, first project experiences within the automotive industry have shown that the TAL analyses are evaluated efficiently and are more resilient than other approaches, e.g., SQL-based analyses. Further, it has been shown that the analysis statements are mainly created by a small group of users, while the majority consumes the results either directly within the TAL editor or indirectly through higher level reports. Therefore, it can be concluded that the language itself needs to offer powerful mechanisms to enable power users to write queries efficiently. Additionally, the language has to be easy to understand so that also direct users recognize the purpose of a query.

While all information needs could be satisfied by the TAL, there was one situation during the case study where a custom function was required. In general, it can be concluded that within every analysis statement the majority of data retrieval and aggregation should be done by the query functions. In addition to previous work, it has been shown that the approach benefits from the additional configuration options in the query language. By introducing custom functions for complex data retrieval, the TAL statements remained small and concise. Subsequently, the metrics and rule statements are used solely for final aggregation and presentation.

Future work could focus on further assessing the applicability in real world projects and defining a structured process to identify reasonable metrics for a specific setting. Such a process might not only support sophisticated traceability analyses but could also propose industry-proven metrics and thresholds. Additionally, it could be investigated, how many custom functions are required throughout different projects and whether there are any shared patterns between them. These patterns might motivate the extension of the standard query functions provided by the TAL.

Some advanced features such as metrics comparisons over time using TIM snapshots to further enhance the analysis are yet to be implemented. Moreover, creating traceability information models for software product lines remains a challenge, which also affects the analysis capabilities.

In addition to evaluating the metrics against static values, future work might also focus on utilizing statistical methods from the data mining field. Classification algorithms or association rules for example could be used to find patterns in traceability information models and thus gain additional insights from large-scale TIMs.

All in all, the paper has introduced a highly customizable approach to specify traceability analyses in order to utilize the extensive insight contained in traceability information models. In addition, the implemented interpreter was used successfully at a tier one automotive supplier to satisfy the information needs of different user groups.

References

[1] H. Bünder, H. Kuchen, and C. Rieger, "A model-driven approach for evaluating traceability information," in SOFTENG 2017, The Third International Conference on Advances and Trends in Software Engineering. IARIA, 2017, pp. 59–65.

[2] O. C. Z. Gotel and C. W. Finkelstein, "An analysis of the requirements traceability problem," in Proceedings of IEEE International Conference on Requirements Engineering, 1994, pp. 94–101.

[3] J. Cleland-Huang, O. Gotel, J. Huffman Hayes, P. Mäder, and A. Zisman, "Software traceability: Trends and future directions," in Proceedings of the on Future of Software Engineering. ACM, 2014, pp. 55–69.

[4] E. Bouillon, P. Mäder, and I. Philippow, "A survey on usage scenarios for requirements traceability in practice," in Requirements Engineering: Foundation for Software Quality. Springer, 2013, pp. 158–173.

[5] J. I. Maletic and M. L. Collard, "Tql: A query language to support traceability," in ICSE Workshop on Traceability in Emerging Forms of Software Engineering, 2009, pp. 16–20.

[6] P. Mäder and J. Cleland-Huang, "A visual language for modeling and executing traceability queries," Software and Systems Modeling, vol. 12, no. 3, 2013, pp. 537–553.

[7] P. Rempel and P. Mäder, "Estimating the implementation risk of requirements in agile software development projects with traceability metrics," in Requirements Engineering: Foundation for Software Quality. Springer, 2015, pp. 81–97.

[8] M. Völter, DSL engineering: Designing, implementing and using domain-specific languages. CreateSpace Independent Publishing Platform, 2013.

[9] J. Cleland-Huang, M. Heimdahl, J. Huffman Hayes, R. Lutz, and P. Maeder, "Trace queries for safety requirements in high assurance systems," LNCS, vol. 7195, 2012, pp. 179–193.

[10] P. Mader, O. Gotel, and I. Philippow, "Getting back to basics: Promoting the use of a traceability information model in practice," 7th Intl. Workshop on Traceability in Emerging Forms of Software Engineering, 2013, pp. 21–25.

[11] A. Graf, N. Sasidharan, and Ö. Gürsoy, "Requirements, traceability and dsls in eclipse with the requirements interchange format (reqif)," in Second International Conference on Complex Systems Design & Management. Springer, 2012, pp. 187–199.

[12] P. Mäder and A. Egyed, "Do developers benefit from requirements traceability when evolving and maintaining a software system?" Empirical Softw. Eng., vol. 20, no. 2, 2015, pp. 413–441.

[13] A. Begel and T. Zimmermann, “Analyze this! 145 questions for data scientists in software engineering,” in 36th International Conference on Software Engineering. ACM, 2014, pp. 12–23.

[14] N. Anquetil et al., “A model-driven traceability framework for software product lines,” Software & Systems Modeling, vol. 9, no. 4, 2010, pp. 427–451.

[15] A. Marques, F. Ramalho, and W. L. Andrade, “Trl: A traceability representation language,” in Proceedings of the 30th Annual ACM Symposium on Applied Computing. ACM, 2015, pp. 1358–1363.

[16] H. Schwarz, Universal traceability. Logos Verlag Berlin, 2012.

[17] M. Mernik, J. Heering, and A. M. Sloane, “When and how to develop domain-specific languages,” ACM Comput. Surv., vol. 37, no. 4, 2005, pp. 316–344.

[18] N. Drivalos, D. S. Kolovos, R. F. Paige, and K. J. Fernandes, “Engineering a dsl for software traceability,” in Software Language Engineering. Springer, 2009, vol. 5452, pp. 151–167.

[19] Automotive Special Interest Group, “Automotive spice process reference model,” 2015, URL: http://automotivespice.com/fileadmin/software-download/Automotive_SPICE_PAM_30.pdf [retrieved: 14.8.2017].

[20] R. S. Arnold and S. A. Bohner, “Impact analysis-towards a framework for comparison,” in ICSM, vol. 93, 1993, pp. 292–301.

[21] C. Ingram and S. Riddle, “Cost-benefits of traceability,” in Software and Systems Traceability, J. Cleland-Huang, O. Gotel, and A. Zisman, Eds. Springer London, 2012, pp. 23–42.

[22] N. Kececi, J. Garbajosa, and P. Bourque, “Modeling functional requirements to support traceability analysis,” in 2006 IEEE International Symposium on Industrial Electronics, vol. 4, 2006, pp. 3305–3310.

[23] J. Cleland-Huang, O. Gotel, and A. Zisman, Eds., Software and Systems Traceability. Springer London, 2012.

[24] H. U. Asuncion, F. François, and R. N. Taylor, “An end-to-end industrial software traceability tool,” in 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering. ACM, 2007, pp. 115–124.

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

[26] The Eclipse Foundation, “Xtext documentation,” 2017, URL: https://eclipse.org/Xtext/documentation/ [retrieved: 14.8.2017].

[27] ——, “Xtend modernized java,” 2017, URL: http://eclipse.org/xtend/ [retrieved: 14.8.2017].

[28] L. Bettini, Implementing domain-specific languages with Xtext and Xtend. Packt Pub, 2013.

[29] M. Verbaere, E. Hajiyev, and O. d. Moor, “Improve software quality with SemmleCode: An eclipse plugin for semantic code search,” in 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion. ACM, 2007, pp. 880–881.

[30] P. Klint, T. van der Storm, and J. Vinju, “Rascal: A domain specific language for source code analysis and manipulation,” in 9th IEEE International Working Conference on Source Code Analysis and Manipulation. IEEE Computer Society, 2009, pp. 168–177.

[31] The Eclipse Foundation, “PDE/user guide,” 2017, URL: http://wiki.eclipse.org/PDE/User_Guide [retrieved: 14.8.2017].

[32] itemis AG, “Yakindu traceability,” 2017. [Online]. Available: https://www.itemis.com/en/yakindu/traceability/

[33] itemis AG, “Kostal finalises aspice assessment successfully with yakindu traceability,” 2017, URL: https://www.itemis.com/en/yakindu/references/kostal/ [retrieved: 14.8.2017].