2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)

Quality Improvement for UML and OCL Models through Bad Smell and Metrics Definition

Khanh-Hoang Doan
Computer Science Department
University of Bremen
doankh@informatik.uni-bremen.de

Martin Gogolla
Computer Science Department
University of Bremen
gogolla@informatik.uni-bremen.de

***Abstract*—Detecting and fixing software quality issues early in the design phase is indispensable for a successful project applying model-based techniques. This paper presents an extension of the tool USE (UML-based Specification Environment) with features for (a) reflective model queries and model exploration, (b) metric measurement, (c) smell detection, and (d) quality assessment with metrics[1]. The newly added functionalities can be fine-tuned by designers, are closely related and can be applied together interactively in order to help designers to achieve better models.**

***Index Terms*—Metrics, UML Metamodel, OCL, Model quality assessment.**

I. Introduction

Assuring the quality of software artifacts in the early phases of the development process has been widely accepted in software engineering as a good practice. Detecting and fixing issues occurring in the design phase, for example, can significantly prevent faults arising in later stages, e.g., the coding phase. Fixing issues reduces the cost and effort of the development process. Models specified by modeling languages such as the UML (Unified Modeling Language) [1] enriched by the OCL (Object Constraint Language) [2] play an increasingly important role and become vital elements in the software development process. Thus improving the quality of a UML and OCL model has a significant influence on the success of the software. This tool paper presents an extension of the tool USE (UML-based Specification Environment) [3], [4], an originally two-level modeling tool (allowing a model and model instantiation), which supports OCL parsing and evaluation as well as model testing, validation and verification. The proposed extension includes features for reflective model queries and model exploration, metric measurement, smell detection, and quality assessment with metrics. Beside metric measurement and smell (issue) detection, which have been introduced in several tools [5]–[7], in this extended version of USE, we also introduce a functionality for model quality assessment with pre-defined metrics. A metrics threshold configuration given by an experienced chief designer will be translated into OCL invariants and evaluated in order to provide qualified feedback to software designers reflecting the goals expressed by the metrics configuration. The newly added functionalities are closely related and can be applied interactively together in order to help designers to achieve better model quality.

[1]A demo video is available online at: https://youtu.be/_yvuPYdDfkw

The rest of the paper is structured as follows. In Sect. II, we present the extension in a nutshell. Section III introduces in detail the new features, including the extension for (a) model quality assessment, i.e., metrics measurement, and (b) quality assessment with smell detection combined with metrics. The contribution ends with related work in Sect. IV and concluding remarks and future work in Sect. V.

II. Extension Overview

Figure 1 presents an overview of the extension that we have made for USE [3], [4]. USE originally was a two-level modeling tool supporting the specification, visualization, validation and verification of information systems based on UML and OCL. The input is a textual specification containing the description of model elements found in UML class diagrams. The model is enriched by additional integrity constraints written in the Object Constraint Language (OCL). Developers can create a system state of the specified model and validate the defined constraints on this system state. USE offers various kinds of UML diagrams for model exploration, e.g., object diagrams and sequence diagrams. It also provides other features for model validation, verification and exploration. Details about applying USE and its options can be found in [3], [4], [8].

In the lower part of Fig. 1, we show new features that are now available, in particular for model quality assessment, i.e., metric measurement, smell detection, and quality assessment with metrics. These new features utilize the UML 2.4 metamodel (the OMG superstructure) [9], which is internally added as a type model for all UML user models. The UML metamodel is applied for the definition of metrics and design smells. In principle, the metrics and smells are defined as OCL expressions in the context of the UML metamodel, however, the evaluation of these OCL expressions is done on the level of a metamodel instantiation, i.e., a UML metamodel instantiation that corresponds to the UML user model. Metamodel instantiation generation is a functionality that runs in the background and takes responsibility for generating the metamodel instantiation corresponding to the input UML user model. As an example, a simple user model that has only one class and one attribute is depicted in Fig. 2; its

978-1-7281-5125-0/19/$31.00 ©2019 IEEE
DOI 10.1109/MODELS-C.2019.00121

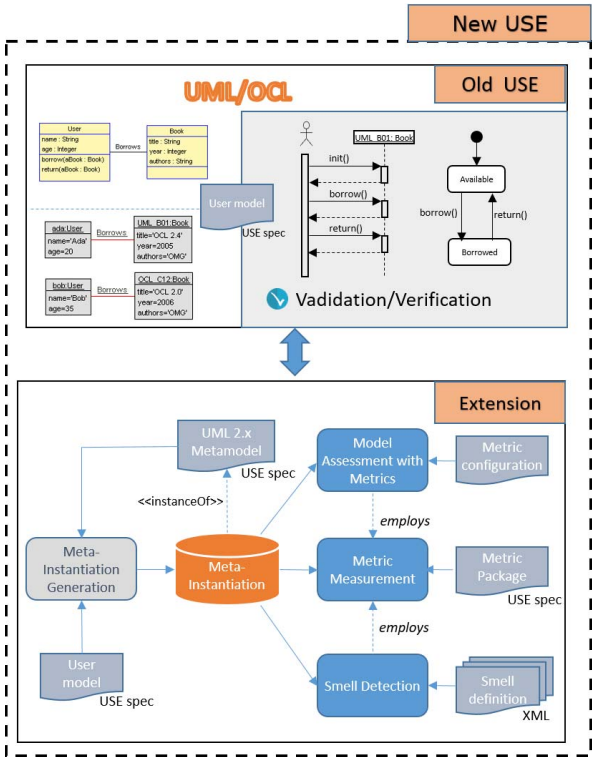

Figure 1. Overview.

corresponding and equivalent metamodel instantiation needs three UML metamodel objects.

In Fig. 2, the class Person is an instance of the metaclass Class, and the attribute name is an instance of the metaclass Property. The extended version of USE offers a three-level modeling representation: model instantiation, user model, and UML metamodel. Modelers can see at the same time a simplified UML metamodel at level M2 that only contains those metaclasses from the metamodel, that the user model needs for instantiation, as shown in the upper part of Fig. 2. The UML user model and the corresponding metamodel instantiation are at the level M1, and a possible USE system state would be at level M0 (not shown in Fig. 2). This ability to see only the relevant part of the UML metamodel provides a better

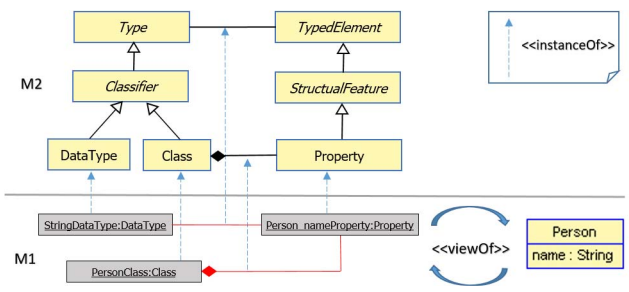

Figure 2. Example: UML user model instantiation by the UML metamodel.

understanding of metamodeling, especially when one wants to define new metrics or design smells, which are presented later. In the following part, we present newly added features that support modelers to assess a model design in detail.

III. Model Quality Assessment

A. Metrics Measurement

In order to measure class model metrics, we categorize metrics into two kinds: class scope metrics and model scope metrics. Two new classes, i.e., `ClassMetrics` and `ModelMetrics` will be added to the UML metamodel level in a separate package. These classes encapsulate metrics as their operations. Class scope metrics, e.g., Number of Children (NOC) [10], Depth of Inheritance (DIT) [10], Data Abstraction Coupling (DAC) [11], are defined as operations in the class `ClassMetrics`, and these operations are formulated as OCL expressions on the metamodel level. For example, the metric DAC is expressed as the following OCL expression:

```
ClassMetrics::DAC(): Integer =
    userdefinedTypeAttributes()→size()
```

where `userdefinedTypeAttributes()` is an auxiliary function that returns an OrderedSet of attributes whose type is a user-defined type. The model scope metrics are similarly defined as operations of the class `ModelMetrics`. Figure 3 shows an example how to retrieve a metrics measurement.

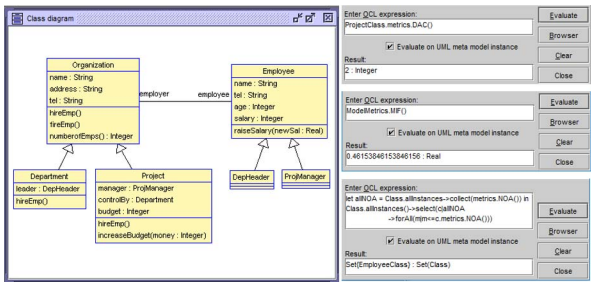

Figure 3. Example for metrics measurement.

Particularly, retrieving a metric of a class can be done by using the following OCL expression template: *<Name of class>*.**metrics**.*<Name of the metric>*, e.g., `ProjectClass.metrics.DAC()`, and the model metric can be computed by calling it: *ModelMetrics.<Name of the metric>*, e.g., `ModelMetrics.MIF()`. Within our tool, metrics can be utilized for querying the model itself. For example, one could formulate an expression (as depicted in the lower corner of Fig. 3) to find the classes which have the maximum values for NOA (number of attributes).

B. Quality Assessment with Metrics Thresholds

Design metrics measurement and evaluation can be used as an early indicator for software quality. In the previous section, we have shown how the metrics of a UML model can be retrieved. In this section, we present a new feature of USE that allows designers to evaluate the design based on *thresholds*

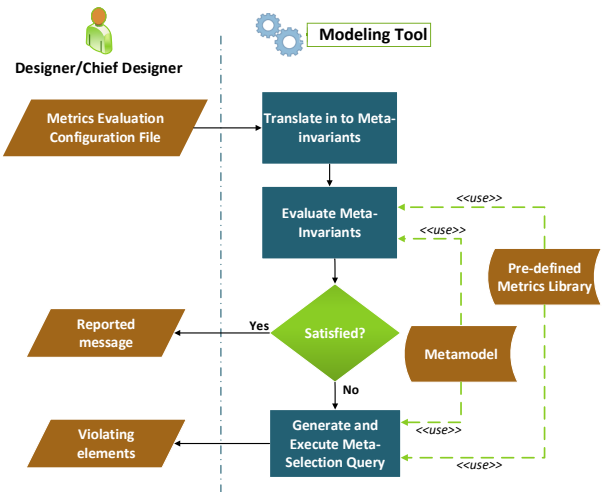

Figure 4. Workflow of model quality assessment with metrics.

for a set of metrics, both class and model scope metrics. For example, one could set the maximum value of the DAC metric in a project to be '4', which means every class in the model should not have more than 4 attributes that have another class as their type. Or the value of the MaxDIT metric for the overall design could be set to be between 1 and 5, i.e., at least one inheritance tree in the project is larger than or equal to 1, and there exists no inheritance tree that is larger than 5. Actually, the proposed metric threshold values could be collected from empirical studies (some of them have been indicated in the survey in [12]) or can be set by an experienced chief designer based on the requirement of the current project.

A metric interval configuration will be exploited for model quality evaluation following the workflow depicted in Fig. 4. The input of the process is a *metrics evaluation configuration*, which is defined in a separate text file. It stores a set of desired metrics threshold settings, and each setting contains the information of the metric scope, the name of the metric as well as the lower and upper thresholds. For example, `0 NOM 4 15` is a class scope metrics, which restricts the `NOM` metric of every class in the model to be between 4 and 15.

```
0 NOM 4 15
0 DIT -1 5
0 NOA -1 10
0 NOC -1 4
1 DSC 5 20
1 MIF 0.2 0.8
```

In the next step, each setting in the metric interval configuration file will be translated to a *meta-invariant* that is evaluated on the metamodel level. For example, setting `0 NOM 4 15` (applied to class scope metrics) is translated to the following meta-invariant:

```
Class.allInstances()→forAll(c|
  4 <= c.metrics.NOM() and c.metrics.NOM() <= 15)
```

The translated meta-invariant indicated below is for the setting `1 MIF 0.2 0.8` (applied to model scope metric):

```
0.2 <= ModelMetrics.MIF() and ModelMetrics.MIF() <= 0.8
```

The result of evaluating these automatically translated meta-invariants will tell the designer which metric threshold settings are satisfied and which ones are unsatisfied in the project. In case the evaluation of a class scope metric is unsatisfied, the designer might want to explore which classes do not satisfy the threshold setting. To do that, the unsatisfied setting is automatically translated into a corresponding query. Executing this query on the metamodel level will show the designer the violating classes. For example, the following meta-query is automatically generated from the setting `0 NOM 4 15`:

```
Class.allInstances()→select(c|
  not (4 <= c.metrics.NOM() and c.metrics.NOM() <= 15))
```

Figure 5 presents screenshots taken from USE that are applying design quality evaluation by utilizing metrics threshold settings. The class diagram of the model under consideration is presented on the left of the figure. Applying USE to evaluate this model with the metric configuration presented above, we get the evaluation result as shown in the upper right part of Fig. 5. Particularly, only the setting of the NOC (number of children) metric [10], i.e., `NOC -1, 4`, returns false. To find which classes cause the NOC metric setting to be unsatisfied, one can double click the row of the NOC metric setting. Then one obtains the violating elements as shown in the lower right part of Fig. 5: In this case the class `Employee`. Additionally, in order to provide better feedback to the developer, violating classes are highlighted in the class diagram of the model under consideration.

C. Smell Detection

Bad design decisions (*design smells*) might reduce the quality of the system under the construction. Therefore, detecting design smells and (if possible) resolving them during the design phase helps to improve the quality of the software. In this section, we introduce a smell detection method, which is added to USE. This method also utilizes the UML metamodel as well as generated metamodel instantiation (see Section II) for design smell definition and detection. As can be seen from Fig. 1, design smell definitions are stored in an external XML file. Each entry in this XML file defines a design smell, which includes several elements: name, description, type (Design guideline, Metric, Naming convention), severity (Critical, Medium, Low), definition (OCL expression), context (Class, Association, Generalization, Property, Operation). The following listing shows the structure of a design smell definition:

```
<DesignSmell id ="">
   <Name>. . .</Name>
   <Desc>. . .</Desc>
   <Type>. . .</Type>
   <Severity>. . .</Severity>
   <Definition>. . .</Definition>
```

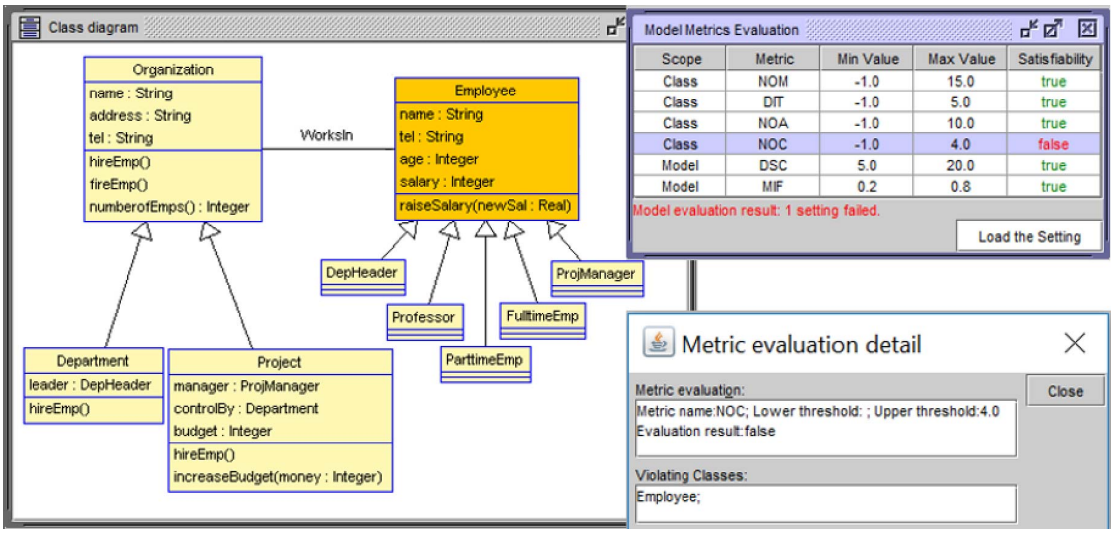

Figure 5. An example of model quality evaluation with metrics.

```
    <Context>. . .</Context>
</DesignSmell>
```

Note that if the severity of a design smell is critical, it means the smell must be fixed, otherwise, it is impossible to create a run-time system of the design under consideration. Medium severity level means the smell should be fixed, and low level means that considerations of the designer must be taken into account to decide whether the smell is really a issue or not. In our approach, the smells library does not contain:

- The constraints that are already specified in the UML metamodel as invariants, e.g., the constraint *"Generalization hierarchies must be directed and acyclical"* [9, p. 93].
- Trivial issues, that are considered as syntax errors and that are checked by our tool while compiling the user model, e.g,. *"A class has no name"* or *"An attribute has no type"*.

In summary, we have formulated 30 design smells as a library ready for the smell detection process. Each smell in the library will be checked on the model under consideration by checking a boolean OCL expression based on the following template:

```
<Context of the smell>.allInstances()→exists(e|
  <smell definition>)
```

For example, the smell *"An abstract class is a subclass of a concrete class"* has the context `Class`; therefore the OCL expression used to evaluate the smell is

```
Class.allInstances()→exists(e|
  e.isAbstract and e.superClass→exists(c|not c.isAbstract))
```

In the case there exists a smell on the model (the evaluation of the OCL expression on the metamodel level returns true), we provide feedback to the developer with a list of user model elements that cause the smell. We do so by executing an OCL query based on the following template:

```
<Context of the smell>.allInstances()→select(e|
  <smell definition>)
```

These violating elements are also highlighted in the class diagram of the model under consideration in order to offer better feedback to the developer. The right upper part of Fig. 6 shows the result when we utilize our tool to detect smells on the simple model shown on the left. As result, our tool detects four smells in the model and the violating elements of a smell, for example, the *"Redundant generalization paths"* smell, can be seen by opening a separate window and they are highlighted in the class diagram as presented in Fig. 6.

We also allow designers to add their own smells to the library (depending on the requirement of a particular project). The smell could be instantly utilized to check the model under consideration. Since defining a smell by an OCL expression on the metamodel level is absolutely non-trivial, one advantage of using our tool is that it offers abilities to explore and query the model itself through our three-level modeling representation (see Section II). Designers can also check the syntax of the OCL expressions and test the smell definition before adding it to the library.

IV. Related Work

Several tools have been presented recently aiming towards model quality assessment. The work in [5] has presented a unified method for the definition and checking of UML class diagram quality properties. For example, syntactic issues, best practices and naming issues are treated as a plugin for EMF. Another approach [6] has been using the *mmSpec* language, integrated into the tool *metaBest* to specify and check quality properties on UML models. [13] introduced an approach for quality assessment of modeling artifacts (metamodels, models, model transformations) by supporting hierarchical quality model definition using OCL and evaluated modeling artifacts based on metric measurements. Another

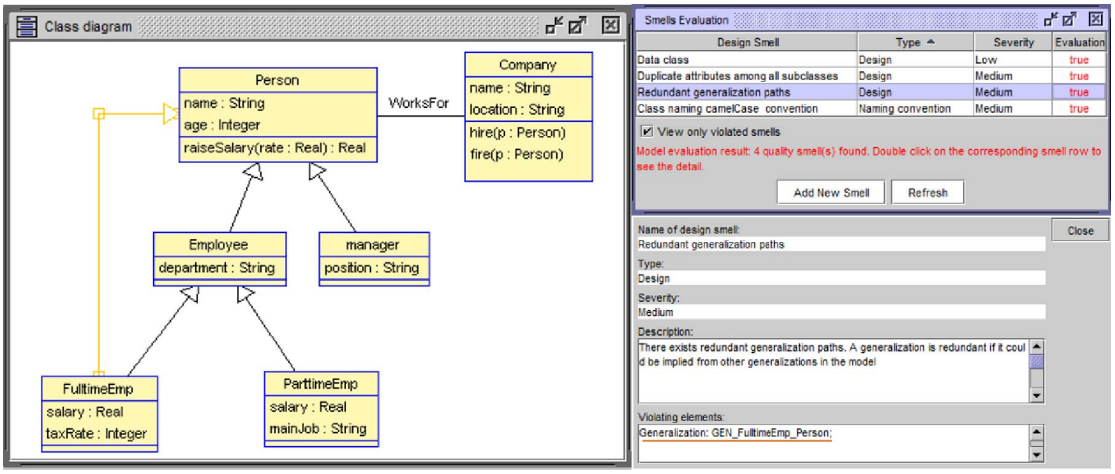

Figure 6. An example of a smell detection.

tool, SDMetrics [7], works with models stored in the XML Metadata Interchange (XMI) format, and metrics and design guidelines are defined in the form of XML-based specifications. Compared to these approaches, our present version of USE offers functionalilites not only for metrics measurement or smell detection but also for reflective model queries and for model exploration. In contrast to the mentioned approaches, developers can *interactively* work with these functionalities to achieve better model quality. In addition, quality assessment with the metrics method presented here does require fewer knowledge and expertise of the underlying modeling language from developres. They only need to define a set of metric thresholds for each particular project and can let the tool check the fulfillment of the model.

V. Conclusion and Future Work

In this contribution, we have presented an extension of the tool USE for model quality assessment. By adding the UML 2.4 metamodel and an automatically generated metamodel instantiation corresponding to the model under consideration, we now offer reflective queries, metric measurement and smell detection. Future work can be done in various directions. First of all, we intend to improve the usability aspect of the model assessment process. For instance, a functionality showing all pre-defined metrics together with recommended upper and lower threshold settings collected from the studies in the literature would be a good solution for the configuration file. Developing means for automatically refactoring will be future work as well. Another promising direction for future work would be to develop a prediction system for external software properties starting from internal quality indicators, i.e., metrics measurement.A further point for improving the applicability of our tool would be to define smells and metrics in terms of graphical patterns over the UML metamodel (UML metamodel class diagram fractions with particular requirements indicated) and to translate these patterns internally into the OCL expressions that our approach currently supports. For example, a particular bad smell could be represented with a generalization arrow between two class templates SUB and SUPER and two attributes SUB::'name'+SUB and SUPER::'name'+SUPER indicating two similar attribute names. Last but not least, larger case studies must give feedback on the applicability of the approach.

References

[1] Object Management Group – OMG, *Unified Modeling Language Specification, version 2.4.1*, 2011. [Online]. Available: http://www.omg.org/spec/UML/2.4.1

[2] J. Cabot and M. Gogolla, "Object Constraint Language (OCL): A Definitive Guide," in *Proc. 12th Int. School Formal Methods for the Design of Computer, Communication and Software Systems: Model-Driven Engineering*. Springer, Berlin, LNCS 7320, 2012, pp. 58–90.

[3] M. Gogolla, F. Büttner, and M. Richters, "USE: A UML-Based Specification Environment for Validating UML and OCL," *Journal on Science of Computer Programming, Elsevier NL*, vol. 69, pp. 27–34, 2007.

[4] M. Gogolla, F. Hilken, and K.-H. Doan, "Achieving Model Quality through Model Validation, Verification and Exploration," *Journal on Computer Languages, Systems and Structures, Elsevier, NL*, 2017, Online 2017-12-02.

[5] D. Aguilera, C. Gómez, and A. Olivé, "A Method for the Definition and Treatment of Conceptual Schema Quality Issues," in *Proc. 31st Int. Conf. ER 2012*, 2012, pp. 501–514.

[6] J. J. López-Fernández, E. Guerra, and J. de Lara, "Assessing the Quality of Meta-Models," in *Proc. 11th Workshop MoDeVVa@MODELS 2014.*, 2014, pp. 3–12.

[7] J. Wüst, "SD Metrics," https://www.sdmetrics.com/index.html, accessed: 2019-06-05.

[8] USETeam, "USE: A UML based Specification Environment," University of Bremen, Tech. Rep., 2007. [Online]. Available: http://www.db.informatik.uni-bremen.de/projects/USE/use-documentation.pdf

[9] Object Management Group, *OMG Unified Modeling Language(OMG UML), Superstructure, version 2.4.1*, 2011. [Online]. Available: http://www.omg.org/spec/UML/2.4.1/Superstructure

[10] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, Jun. 1994.

[11] W. Li and S. Henry, "Object-oriented metrics that predict maintainability," *J. Syst. Softw.*, vol. 23, no. 2, pp. 111–122, Nov. 1993.

[12] M. Genero, M. Piattini, and C. Caleron, "A survey of metrics for UML class diagrams," *Journal of Object Technology*, vol. 4, pp. 59–92, 2005.

[13] F. Basciani, J. D. Rocco, D. D. Ruscio, L. Iovino, and A. Pierantonio, "A customizable approach for the automated quality assessment of modelling artifacts," in *10th International Conference on the Quality of Information and Communications Technology*, 2016, pp. 88–93.