A SOFTWARE INFRASTRUCTURE FOR INFORMATION QUALITY ASSESSMENT

Morgan Ericsson, Anna Wingkvist, and Welf Löwe
Linnaeus University, School of Computer Science, Physics and Mathematics
Växjö, Sweden
(morgan.ericsson | anna.wingkvist | welf.lowe)@lnu.se

Abstract: Information quality assessment of technical documentation is nowadays an integral part of quality management of products and services. These are usually assessed using questionnaires, checklists, and reviews and consequently work that is cumbersome, costly and prone to errors. Acknowledging the fact that only humans can assess certain quality aspects, we suggest complementing these with automatic quality assessment using a software infrastructure that (i) reads information from documentations, (ii) performs analyses on this information, and (iii) visualizes the results to help stakeholders understand quality issues. We introduce the software infrastructure's architecture and implementation, its adaptation to different formats of documentations and types of analyses, along with a number of real world cases exemplifying feasibility and benefit of our approach. Altogether, our approach contributes to more efficient and automatic information quality assessments.

Key Words: Information Quality Assessment, Technical Documentation, Software-based Analysis

INTRODUCTION

Quality assessment and assurance is an important part of production of documentation (technical information) [6]. A lack of quality reduces not only the value of documentation, but also of the product (service and/or process) it is attached to. A lack of quality can reduce the perceived quality of the brand or company associated with it [15].

Quality depends not only on the technical information provided, but also the product and the context it is used within. A change to the product will reduce the quality of the documentation if it is not updated to reflect the change. Hence, it is important that the quality assessment and assurance is a continuous process.

In this paper we present a software infrastructure that can be used to define analyses and visualizations to assess and communicate the quality of documentation. The (software) infrastructure is open source[1] and originally developed for software quality assessment. It creates models of documentations according to a common meta model that describes a family of documentations. This common meta model is used to define analyses and visualizations, and mappings from real world documentations.

The paper is organized as follows. We begin by discussing examples of how information quality assessment can be supported by automatic analyses and visualizations. Based on these examples, we present the architecture of our infrastructure. We continue by discussing the use of models and meta models to make the infrastructure general and reusable even when new documentation formats or analysis questions need to be integrated. We then revisit the practical examples and shows how the analyses and visualizations can be implemented using the infrastructure. Finally, we discuss related work, present conclusion and outline future research directions.

[1] The infrastructure, implemented in Java, can be downloaded from http://arisa.se.

SOFTWARE-SUPPORTED QUALITY ASSESSMENT IN PRACTICE

This section presents three real world examples where we used the software infrastructure to automatically assess the quality of documentations. Each example discusses a potential quality issue, the analysis we used to detect it, how the results were visualized and the outcome. Figures 1 to 3 are included to show examples of visualizations; the text nodes are not meant to be legible.

Text Clone Detection

A text clone is a block of text that is repeated, in various degrees of similarity across the documentation, i.e., redundant text. Redundant text is not necessarily an indication of poor quality, since it can make the text easier to read and understand by increasing the cohesion and reducing the number of cross-references, for example. But, redundant text can also increase the cost of storing, maintaining, and translating the documentation.

In order to investigate the degree of text clones, we implemented a clone detection analysis and used it to assess (the non-classified parts of) a warship documentation. The clone detection analysis determines how similar two parts of the documentation are by first comparing text and then documentation structure. Figure 1 depicts the result of the clone detection.

We analyzed 913 XML documents from the warship documentation and found that only 6 of them were unique. 20 documents were exact clones of another document, and the remaining were similar to some extent. On average, a document was 54% unique.

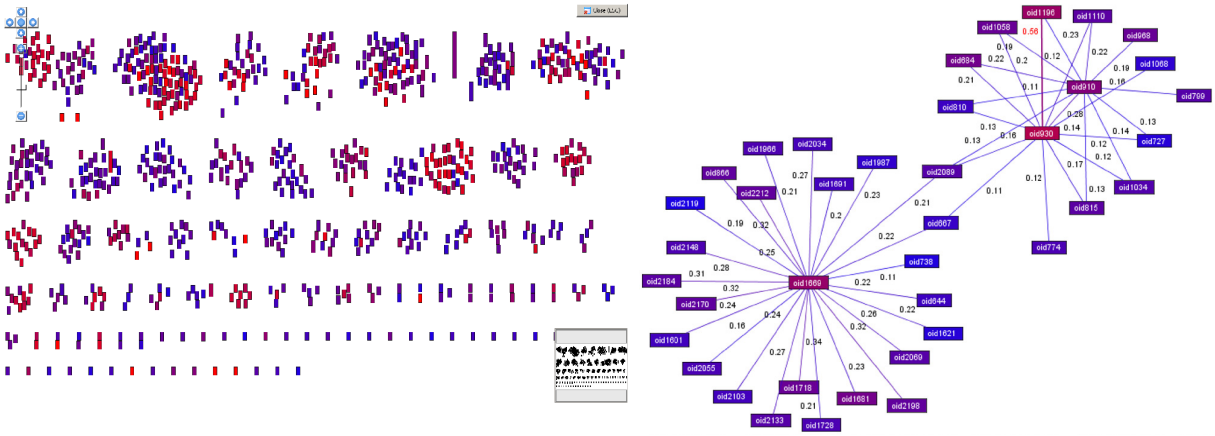

Figure 1: The result of a clone detection analysis of the technical information of a warship is visualized as clusters of similar documents (left). Each cluster can be viewed in more detail (right). The color of a box represents the degree of similarity from low (blue) to high (red).

Reference Analysis

Cross-references are used to relate different parts to each other in a documentation. If too many cross-references are needed to understand the content, especially, if non-local and forward references are used, it can be hard to follow the narrative. Consequently, information in the paragraphs and sections is not self-contained. This can indicate a suboptimal documentation structure, which might make it hard to read and to find the relevant parts.

We applied a reference analysis to the warship documentation (same as text clones). Figure 2 shows the outcome of this analysis. Each box corresponds to an XML document and the color represents which part

of the documentation it belongs to. The distance between two boxes represents the degree of cross-references. The left figure shows the whole documentation while the right figure shows a part of the documentation with a large degree of cross-references.
Figure 2 (right) displays a problematic set of documents with many cross-references (cluster) since it contains documents from eight different parts of the documentation (eight different colors of boxes). This can indicate a quality issue and it might be wise to reconsidered if corresponding documents should be moved closer to each other.

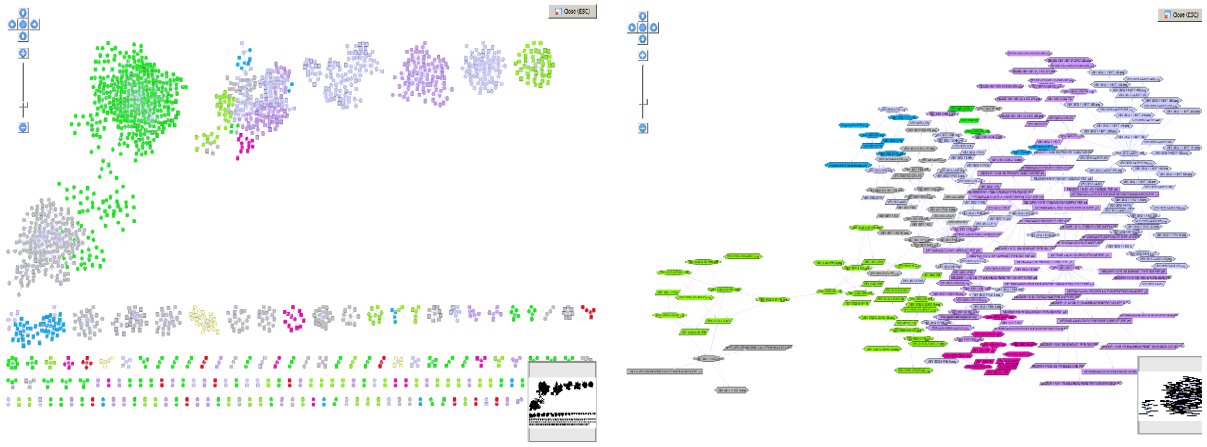

Figure 2: Visualizations of cross-reference analysis results, cross-reference cluster overview (left) and detailed view zoomed into one problematic cluster (right).

Use Of Meta-Information

Meta-information such as tags and keywords can be used to improve usability and accessibility of documentations. However, improper definition of these, for example attaching a keyword to the wrong part of the documentation, can reduce usability.

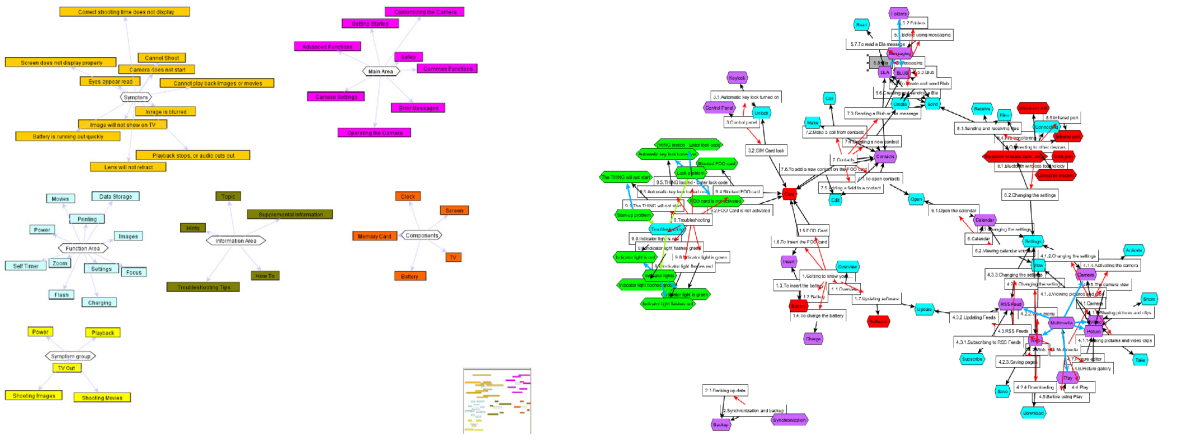

Figure 3: A meta-information analysis of a mobile phone technical documentation. The visualizations show the structure of the meta-information (left) and the relation to information (right). The color of an applicability tag identifies the category it belongs to.

In order to investigate how well meta-information is used, we implemented a meta-information analysis and applied it to the documentation of a mobile phone. In this documentation, a section can have a number of applicability tags that describe the content. Each tag belongs to a category. The analysis extracts all the applicabilities for each category, and the applicabilities (and categories) for each section of the documentation. Figure 3 (left) shows the applicability tags and the categories, and Figure 3 (right) shows applicability tags and sections of the documentation.
We analyzed 12,286 XML documents from the mobile phone documentation and found that some categories were concentrated to certain parts of the technical information, while other categories were spread out. This imbalance of tagging can indicate quality issues, but in this case the technical writers confirmed that it was intentional and part of the design.

AN INFRASTRUCTURE FOR INFORMATION QUALITY ASSESSMENT

In this section we present an infrastructure that supports the implementation and execution of analyses and visualizations of information (quality). This infrastructure was defined and used to implement the analyses and visualizations previously described.
The infrastructure consists of three major components that are centered on a common information repository that captures (abstractions) of the documentations. The three components are: (i) readers that map documentations to the repository, (ii) analyses that assess and modify the repository, and (iii) visualizations that present interactive views of the repository to stakeholders, e.g., technical writers, product managers, etc.

The Common Information Repository

The repository is the heart of the infrastructure. It captures abstractions of documentations, for example by removing presentation details such as layout, typeface, etc. We refer to such abstractions of the documentation as models (of a documentation). Each model consists of a number of data entities, such as documents, sections, paragraphs, and their relations, such as cross-references and structural containments. These data entities and relations are described by a common meta model, that can describe any model that can be captured by the repository.
For instance, if we want to capture a documentation consisting of sections and paragraphs, our common meta model should contain a document entity, a section entity and a paragraph entity, together with a structural containment relation to express the relation between documents, sections and paragraphs.
Many data entities and relations are general and common for all documentations, but our meta model will never be complete, i.e., it will never be able to describe all possible documentations. However, this incompleteness is an advantage of our approach as we can keep the meta model as small as sufficient for our analysis questions and grow it later on demand when new questions arise. We use the common meta model as a configuration parameter for an instantiation of the infrastructure, i.e., the meta model can be adapted to the actual documentation types that we want to analyze. We use an Entity-Relationship (ER) style diagram (cf. Figure 5) to describe the common meta model, and we can automatically generate an implementation (in Java) based on such a diagram.

Readers That Map Documentations To The Repository

To apply predefined analyses and visualizations, we need to map documentations that are captured in a specific format, such as XML, to the common meta model. This process identifies data entities and relations in the documentation, and maps these to corresponding entities of the common meta model, i.e., it creates a model of the documentation.
The readers are defined using a program language, and anything that can be accessed by software, e.g.,

text files, databases, web servers, can be mapped to the repository. We map a documentation by traversing it in document order and instancing the corresponding classes of the repository. Note that the classes and their constructors are automatically generated from (and documented by) the common meta model specification. For many common formats, such as XML, there exist predefined processing libraries and frameworks. In such cases, it is trivial to define a reader.
Suppose our documentation consists of sections, subsections, and paragraphs in an ordinary text format. The walker that traverses the documentation starts by invoking a start action that triggers the creation of a document entity. For each section, the walker invokes an action to construct a section entity and adds a structural containment relation to the document entity. Paragraphs and subsections are constructed and connected in a similar manner.

Reusable Analyses

Analyses such as the clone detection and reference analysis read from and write to the repository. Since the repository and the models of the documentation it contains are independent of the document format, analyses can easily be reused for several documentations.
We previously stated that our common meta model will never be complete, and that the advantage is that it can be reuses and thus changed on demand. Analyses should be robust to change in the underlying repository, in order to increase reusability and reduce maintenance, for example. We achieve this by defining analyses on views of the repository rather than the actual repository. A view contains only the entities and relations necessary for a particular analysis or class of analyses, which makes it robust to change. Views are specified in the same way as the common meta model, and the transformations required to provide views are automatically generated from these specifications.
Assume we extend our running example with a simple analysis that counts the number of subsections per section. A naive implementation would access every section in the repository, and then count each child that is a subsection. Assume we extend the common meta model so that each section contains a header paragraph and a body, which in turn contains the subsections. As a consequence, our analysis would not find any subsections, and return zero for each section. If we instead define a view that contains only sections and subsections, and define our analysis on that view, it would not have been affected by the change.

Visualizations

We use visualizations to communicate information to stakeholders about the documentation and the analyses we perform. A visualization defines a mapping from the information contained in the repository to a visual domain. The repository contains a model of the documentation and different views on it, which are annotated by the different analyses. The visual domain contains objects with various attributes, such as shapes, colors, textures, and positions. The mapping between the two brings meaning and context to the visual objects in terms of documentation and quality.
A visualization is a specific kind of analysis that operates on specific views of the common (information) repository. The mappings created by these analyses can be configured. We map repository entities (of different types) to visual objects (of different shape or color) and map their quality attributes to visual attributes of these objects. As displayed and discussed by the several examples provided.

MODELS AND META MODELS

In this section we present the design of the common repository and how it can be automatically generated from a specification. We also discuss the specification in detail. We previously discussed how we create abstractions of documentations, i.e., models that are stored in the common information repository. The

repository, i.e., a set of similar models, is described by a common meta model. On the (document) model level, we consider concrete entities and relations, such as sections and paragraphs, which exist in a particular documentation. On the common meta model level, we consider abstract entities and relations, that might exist in a certain (document) model. In this section, we introduce a new formalism, the meta meta model, that is used to specify the common meta model. And, we use tree grammars to describe the main document structure and relational algebra to describe the relations.

A Document-specific Meta Model

A documentation follows specific rules and/or conventions, specified either implicitly or explicitly. We refer to this as the documentation-type-specific, in short *document-specific*, meta model. For example, if we use XML to represent our documentation, we use an XML Schema or a Document Type Definition (DTD) to specify the structure of the document. The DTD or XML Schema specifies a document-specific meta model.

In general, a documentation consists of entities structurally contained in each other and relations between them, defined by a document-specific meta model $M^{DT} = (G^{DT}, R^{DT})$. We use the following meta meta model to describe document-specific meta models: G^{DT} is a tree grammar specifying the set of model entities and their structural containment and R^{DT} is a set of relations over model entities. Formally, $G^{DT} = (T^{DT}, P^{DT}, document^{DT})$ where T^{DT} is a set of model entity types, P^{DT} is a set of productions in Extended Backus-Naur Form (EBNF) that define the containment tree structures, and $document^{DT} \in T^{DT}$ is the root entity type of the structural containment trees. The EBNF productions $p \in P^{DT}$ are of the form $t ::= expr$ where $t \in T^{DT}$ and $expr$ is a regular expression over $T \subseteq T^{DT}$. Expressions are either sequences $(t_1, \dots, t_k)$, iterations (t^*), or selections $(t_1 | \dots | t_k)$. R^{DT} denotes a set of relations over model entities, $R^{DT} = R_1^{DT}, \dots, R_n^{DT}$, where each R_i^{DT} is defined over T^{DT}, and *String* and *Num* entities. The latter represent general string and numerical attributes, respectively.

Example 1: Let $M^{DTD} = (G^{DTD}, R^{DTD})$ be the meta model corresponding to a specific DTD, hence a specific document type DT. $G^{DTD} = (T^{DTD}, P^{DTD}, document^{DTD})$ defines the containment structure trees of the DTD (i.e., the information set regardless of their XML encoding). An example of such a tree is depicted in Figure 4 (left). It contains entity types for documents ($document^{DTD}$), (sub-)sections ($section^{DTD}$ and $subsection^{DTD}$), paragraphs ($paragraph^{DTD}$), and figures ($figure^{DTD}$). Paragraphs contain text entities of type $text^{DTD}$ and reference entities of type ref^{DTD}. Productions P^{DTD} define the structural containment in structure trees with $document^{DTD}$ entities as root. R^{DTD} contains a binary $refers$ relation of type $refers^{DTD} : ref^{DTD} \times (section^{DTD} \cup subsection^{DTD} \cup figure^{DTD})$ and caption and content text relations $caption^{DTD} : (document^{DTD} \cup section^{DTD} \cup subsection^{DTD} \cup figure^{DTD} \cup ref^{DTD}) \times String$ and $content^{DTD} : text^{DTD} \times String$. $String$ represents the character sequence of a caption or content.

A Common Meta Model

The common meta model, M, describes a family of models by abstracting document-specific details. As discussed previously, the meta model is not complete, and will hence evolve over time, as new document-specific meta models are added to an instantiation of the infrastructure. We can use the same formalism, the same meta meta model, to describe the common meta model, since at any point in time, it can be defined by a tree grammar and a set of relations.

We denote the common meta model by $M = (G, R)$. The only major difference between the common meta model and the document-specific meta model is that relations R must contain a 0^{th} element, a numerical identifier for maintaining the document order of relation tuples as we will detail in Example 2.

Example 2: Let $M = (G, R)$ be the common meta model, which we will use in our running example. $G = (T, P, document)$ defines admissible common structure trees containing entities for documents and

sections. An example of such a tree is depicted by Figure 4 (middle). Productions P define the structural containment in these trees. R contains relations $refers: Num \times section \times section$, $caption: Num \times (document \cup section) \times String$ and $content: Num \times section \times String$. $String$ represents the character sequence of a caption or content, and Num the numerical type of an id encoding document order.

Mapping from Document-specific to Common Meta Models

The common meta model is an abstraction of different document-specific meta models. For each such document-specific meta model, this abstraction is defined by a document-specific mapping α^{DT}. This mapping itself is defined by mapping grammars G^{DT} to the common meta model grammar G and relations R^{DT} to the common meta model relations R. The mapping, in turn, induces a reader implementation, which reads documentations of a document-specific meta model and creates models of the common meta model. Readers can be generated automatically from the mapping specifications.

For a grammar, α^{DT} is defined by mapping document-specific to common meta model entity types $\alpha^{DT}: T^{DT} \rightarrow T$; the documentation meta model entity type $document^{DT}$ is always mapped to the common meta model document entity type, i.e., $\alpha^{DT}(document^{DT}) = document$. For selected relations $R_i^{DT} \in R^{DT}$, we define mappings to corresponding relations $R_i \in R$, i.e., $\alpha^{DT}: R^{DT} \rightarrow R$.

In general, we do not require α^{DT} to be surjective or complete, since this would be unnecessarily restrictive. Some common meta model entity and relation types do not correspond to entity and relation types in every document-specific meta model (i.e., the mapping is not surjective), and some document-specific meta model entity and relation types may be ignored (and not complete).

Example 3: Our mapping α^{DTD} maps entity and relation types of M^{DTD} (Example 1) to the common meta model M (Example 2) as follows: $\alpha^{DTD}(document^{DTD}) = document$, $\alpha^{DTD}(section^{DTD}) = section$, and $\alpha^{DTD}(subsection^{DTD}) = section$. Entities of other types are dropped. The relation types are mapped as follows: $\alpha^{DTD}(refers^{DTD}) = refers$, $\alpha^{DTD}(caption^{DTD}) = caption$, and $\alpha^{DTD}(content^{DTD}) = content$.

The mapping α^{DT} is specified on the meta model level and implies a mapping map for concrete model entities in the following way. First, the document-specific structural containment tree is mapped to the corresponding common structure. Then, the mapped relations are attached to the common structure. The mapping of the containment trees is defined recursively. Starting at the root, we traverse the document-specific containment tree in Depth-First-Search (DFS) order. We create new common model entities for document-specific entities of types that have a mapping defined in α^{DT}. These are referred to as the *relevant* entities. The other, *irrelevant* document-specific entities are ignored.

A generic event-based interface between specific and common meta models and an abstract algorithm to map model entities are given in Procedures 1, 2, and 3. A tree-walker (see Procedure 1), initially called with the root entity of the specific model, traverses the containment tree in DFS order and generates *startNode*-events on traversal downwards and *finishNode*-events on traversal upwards, respectively. Entities of the structural containment tree are pairs *(id, t)*, with $id \in Num$ and $t \in T$ as the entity identifier (a DFS rank representing the document order) and type, respectively.

Procedure 1:

```
generateTreeEvents(n=(id; t))
   call startNode(n)
   for each c ∈ childrenOf(n) do
      call generateTreeEvents(c)
   end for
   call finishNode(n)
```

The common model data structure is created by the corresponding event-listener, *startNode* (see Procedure 2) and *finishNode* (see Procedure 3). They preserve the tree structure but filter out irrelevant entities.

Procedure 2:
startNode ($n = (id;\ t^{DT})$)
  if $\alpha^{DT}(t^{DT})$ is defined then
    create new entity $n' = (id;\ \alpha^{DT}(t^{DT}))$
    append n' to children of Stack.top
    Stack.push(n')
  end if
  $map(n)$ = Stack.top *// defines closest relevant ancestor of n required in Procedure 4*

Procedure 3:
finishNode ($n = (id;\ t^{DT})$)
  if $\alpha^{DT}(t^{DT})$ is defined then
    Stack.pop
  end if

A document-specific relation is a set of tuples $R_i^{DT}(n_1, \dots, n_k)$ over structural containment tree entities (and possibly string and numerical attributes). When constructing the common model, we ignore the relations that are not mapped by α^{DT}. Let $R_i^{DT}: t_1^{DT} \times \dots \times t_k^{DT}$ be a document-specific relation with a mapping $\alpha^{DT}(R_i^{DT}) = R_i$.
Assume that each entity type t_j^{DT} is mapped by α^{DT}. Then, each entity in tuples $R_j^{DT}(n_1, \dots, n_k)$ will have a correspondence in the common meta model and R_i can be defined over those entities. However, the following three situations that make this mapping more complex.
First, if α^{DT} is not defined for a type of an entity n_i in $R_i^{DT}(n_1, \dots, n_k)$, we will "lift" the relation to n_i's closest relevant ancestor entity. That is the entity in the common model corresponding to the closest transitive parent of n_i, which is relevant. It is captured by $map(n_i)$, defined in Procedure 2 and used in Procedure 5.
Second, the 0th element of any common relation $R_i: Num \times t_1 \times \dots \times t_k$ is a numerical identifier which we implicitly generate. Each tuple $R_i^{DT}(n_1 = (id, t^{DT}) \dots n_k)$ is mapped to $R_i(id, map(n_1), \dots, map(n_k))$, i.e., we make the entity identifier id of the first tuple element n_1 in the target relation R_i explicit. This is necessary, as n_1 might be irrelevant and the relation can get lifted to a parent p of n_1. Another relation tuple with first element $n' = (id', t^{DT})$, with $id < id'$, may get lifted to the same parent p, but we still can sort the target tuples according to the document order of their origins.
Third, if n_i^{DT} is a *String* or a *Num* instance then n_i is either the same *String* or *Num* instance, or is explicitly dropped using a special entity called *Void*. It is never transformed in any other way. The mapping of a specific to a common relation is performed in a second pass after the creation of the common model structure tree. It uses the event generator *generateRelationEvents* (see Procedure 4) and the corresponding event listener *newRelationTuple* (see Procedure 5).

Procedure 4:
generateRelationEvents (R^{DT})
  for each $R_i^{DT} \in R^{DT}$ do
    if $\alpha^{DT}(R_i^{DT})$ is defined then
      for each $R_i^{DT}(n_1, \dots, n_k)$ do
        call newRelationTuple$\left(\alpha^{DT}(R_i^{DT}), (n_1, \dots, n_k)\right)$

end for
end if
end for

Procedure 5:
newRelationTuple$(R_i, (n_1, \dots, n_k))$
for each $n_j = (id, t_j^{DT}) \in n_1, \dots, n_k$ do
if j == 1 then r_0 = id end if *// create 0th element of target relation*
if $(t_j^{DT} == String \vee t_j^{DT} == Num) \wedge t_j == Void$ then *// drop string or numerical value*
$r_j = void$
else if $t_j^{DT} == t_j == String \vee t_j^{DT} == t_j == Num$ then *// copy string or numerical value*
$r_j = n_j$
else *// lift the relation to the closes relevant ancestor*
$r_j = map(n_j)$ *// map captures the closest relevant ancestor of n_j as defined in Procedure 2*
end if
end for
add tuple $(r_0, r_1, \dots, r_k)$ to relation R_i

Note that the abstract event generation (algorithm schema) and the event handlers work independently of different real world documentation types and their mappings to the current common meta model. The abstract event generation and the event handling do not change when any of these components change. However, a concrete implementation of the abstract event generation, i.e., the implementation of Procedures 1 and 4, are document-specific and must obey its specific meta model implementation. Note that we need not map all entities and relations that a documentation provides to the common model. We may do that lazily when required by analyses.

Analysis-Specific Meta Models and their Mapping from Common Meta Models

Analyses might directly traverse the common models, extract required information, and perform computations. However, we introduce analysis-specific views on the common meta model, i.e., further abstracting from the common meta model, providing exactly the information required by a specific analysis. Several analyses can share a view, and hence, a view factors out the information useful for a set of analyses.

Views are further abstractions of the common meta model. Formally, a view is a meta model that is specific for an analysis A (or a set of analyses) described as $V^A = (G^A, R^A)$. G^A is a tree grammar that specifies the set of view entity types and their structural containment required by A. R^A is a set of relations over view entity types required by A. The views are specified using the meta meta model.

View model construction follows the same principles as the mapping and abstraction from document-specific to common meta models. We ignore some entity types, which results in filtering of the corresponding nodes. We propagate relevant descendants of filtered entities to their ancestors by adding them as direct children. Moreover, we ignore some relation types and attach remaining relations defined over filtered entities to the relevant ancestors of those entities. The numerical DFS rank from the common models may be copied or dropped depending on whether document order plays a role or not for a specific analysis A. As in our mapping from document-specific to common meta models, construction of a view is defined using a mapping specification denoted α^A, where A is a specific set of analyses.

Finally, analyses access the corresponding view and perform computations. We deliberately skip a discussion on how to capture analysis results as part of the model and display them; we refer to Löwe and Panas [13] for applicable extensions.

Example 4: Let *Coupling* compute the relative coupling of a section via references, i.e., the ratio of the

number of references within section s and its subsections, denoted $localRefs(s) = |refers(s1, s2) : contains^*(s, s1) \wedge contains^*(s, s2)|$, and all references incoming to and outgoing from s, denoted $allRefs(s) = |refers(s1, s2) : contains^*(s, s1) \vee contains^*(s, s2)|$. In short, $Coupling(s) = localRefs(s)/allRefs(s)$. An appropriate view meta model $V^{Coupling}$ would contain the entity types $document^{Coupling}$ and $section^{Coupling}$ along with the relation type $refers^{Coupling} : Void \times section^{Coupling} \times section^{Coupling}$ with the mappings $\alpha^{Coupling}(document) = document^{Coupling}$, $\alpha^{Coupling}(section) = section^{Coupling}$, and $\alpha^{Coupling}(refers) = refers^{Coupling}$. Note that the coupling ignores the order in which references occur and therefore drops the corresponding DFS *id* entity of type *Num* from the common meta model relation $refers : Num \times section \times section$, along with other irrelevant entities and relations.

Figure 4 shows an example mapping a document-specific model to a common model (cf. Examples 1–3), and to a *Coupling* view model. Note that the order of the two refers tuples in the common meta model is maintained due to the 0th element in each relation tuple mapped from the DFS rank of the source node of the 1st element in the corresponding document-specific model relation tuple (6 and 7, respectively). In this example, the *Coupling* of section 2 is *1/2*; for all other sections it is *0*.

META MODEL EVOLUTION

The initial common and analysis-specific meta models are usually designed to be suitable for a set of documentation types and analyses. New documentation types (analyses) that only provides (relies on) entities and relations contained in the current meta model are trivial to add. In the general case, when a new documentation type or a new analysis is added, the meta model needs to evolve. In this section we show how to control and reduce the effect these meta model changes have on existing analyses.

Assume that analysis A cannot be applied on entity and relation types captured in any of the existing analysis-specific meta models, but the required types are already captured in the common meta model. In this case a new analysis-specific meta model V^A and a new mapping α^A from the common to this specific meta model needs to be specified. There is no additional implementation effort since event-generators and event-handlers that populate the new meta model are generated automatically; see Procedures 1–5.

In the more general case, a new analysis also requires an extension of the common meta model, that in turn, implies that the common model creation is affected. Furthermore, the documentations are either able to provide these extended types of entities and relations, or else a new documentation type — along with the corresponding document specific mappings — needs to be integrated. In both cases, we need to extend the common meta model *M* and the document-specific mapping(s) α^{DT}. Given that structure tree and relation event generators work according to our schemas in Procedures 1 and 4, no additional programming is needed when reusing an existing documentation type. Then, we specify the missing entity and relation types as relevant in α^{DT}, and the common repository and model creation are generated automatically. New documentation types require specific implementations of the Procedures 1 and 4 schemas.

Example 5: Assume the common meta model of Example 2 that contains *document* and *section* entities and *refers* and *contains* relations, and the analysis-specific meta model of Example 4 for computing coupling between sections. We can now add a new *Complexity* analysis that counts paragraphs by introducing *paragraph* entities to the common meta model; We change the grammar productions accordingly to capture the new structural containment: *document* ::= *section** and *section* ::= (*section* | *paragraph*)*.

On top of this new common meta model, we can define a new analysis-specific meta model $V^{Complexity}$ with a corresponding analysis-specific mapping $\alpha^{Complexity}$. Based on $V^{Complexity}$, we can implement our *Complexity* analysis.

Note that our analysis-specific meta model $V^{Coupling}$ and its mapping $\alpha^{Coupling}$ as well as the whole

$Coupling$ analysis from Example 4 remain unchanged. The original common meta model relation $refers : Num \times section \times section$, becomes the new common meta model relation $refers : Num \times paragraph \times section$, since $paragraph$s are now the relevant ancestors of *ref* entities in the common meta model. However, when applying $\alpha^{Coupling}$ to this new common meta model, $V^{Coupling}$ remains unchanged, i.e., its grammar productions are still: $document^{Coupling}$ ::= $section^{Coupling*}$ and $section^{Coupling}$ ::= $section^{Coupling*}$ and the $refers^{Coupling}$ relation type is still $refers^{Coupling} : Void \times section^{Coupling} \times section^{Coupling}$ as before. This is because $section^{Coupling}$ entities are the relevant ancestors of $paragraph$ entities in the analysis-specific mapping $\alpha^{Coupling}$. Hence, the old $Coupling$ analysis can be reused and is applicable without any change.

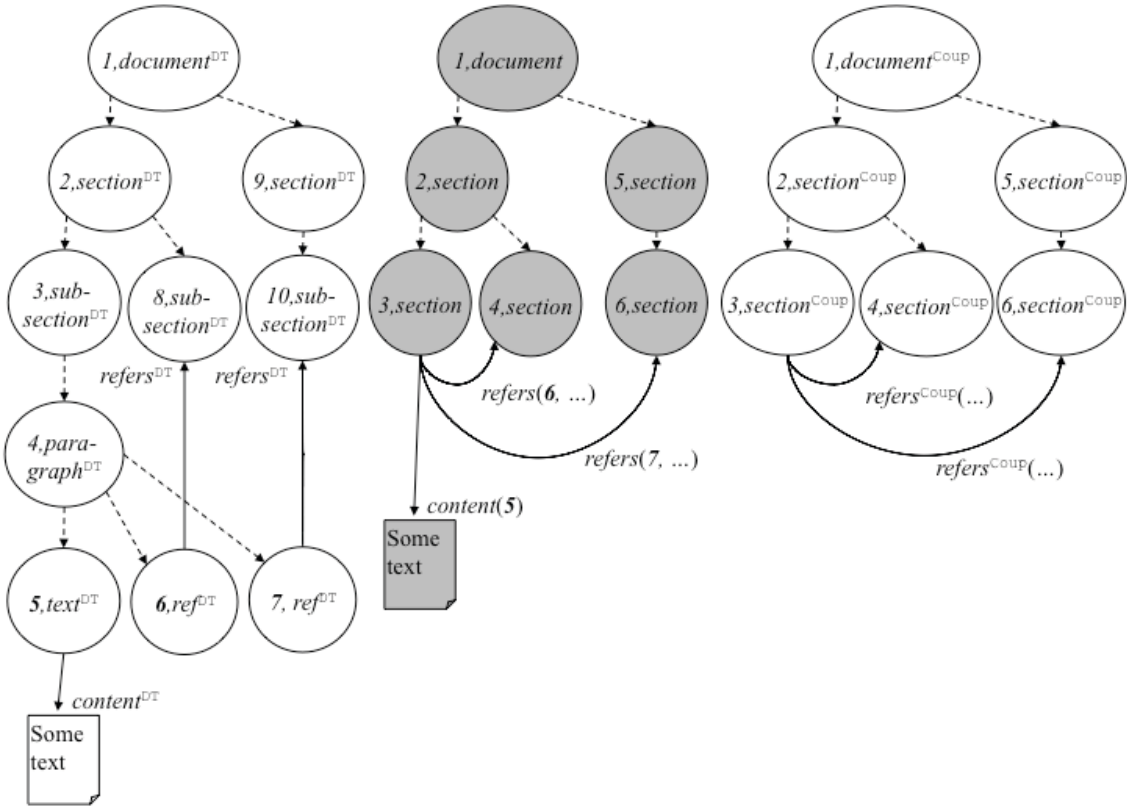

Figure 4: Mapping of document-specific to common (gray) and analysis-specific models (left to right). Entities of the three models are labeled with their *id* (DFS rank) and *type*. Dashed edges display structural *contains*, solid edges *content* and *refers* relations.

What we learned from Example 5 is that effects of changing the common meta model are often filtered by subsequent analysis-specific view abstractions. This should not comes as a surprise since the analysis-specific mapping is defined by explicitly declaring relevant entity and relation types; newly introduced types were not known back then and, hence, could not have been declared relevant in already existing analysis-specific mapping specifications. As long as changes just extend the common model trees, analysis-specific mappings would compensate for the change and (re-)produce the original analysis-specific model for the existing analyses.

However, by changing the common meta model and, thereby, relevant entity and relation types, we can experience reuse problems. If formerly irrelevant entities become relevant, mappings may create relations that no longer have the same type as before. Practically, this would mean that a relation *R* that used to be attached to an entity of type *X* is now attached to a descendant of that entity of type *Y*. This, in turn, could lead to situations where analyses cannot work as before. For instance, an analysis iterates all entities of type *X* and counts the number of times that these entities occur in a relation *R*. Model extension changes the result of this analysis: the count is now always *0* since only *Y* entities occur in *R* (and no *X* entity anymore). Hence, analyses cannot be reused without (programmed) adaptation. Fortunately, the effect of changes in the common meta model is often not visible in the existing analysis-specific models and, hence, many analyses can be applied without changes (as Example 5 shows).

There are safe changes to the common meta model guaranteed not to affect an analysis *A*:

- Adding a new type to a sequence expression on the right-hand side of a production.
- Adding an existing type *X* to a sequence if no other type relevant for *A* can transitively be derived from *X*.
- Introducing a new production *X*::= ... if no type relevant for *A* can transitively be derived from *X*.
- Adding a new relation *R*.

In all these cases, the entities and relations newly introduced to a common model will be filtered by existing analysis-specific mappings and the relations will be attached to the original entity types in the analysis-specific models (proofs are omitted here). Conversely, if a meta model change is not safe for an analysis *A*, we should check and potentially adapt *A*.

SOFTWARE-SUPPORTED QUALITY ASSESSMENT REVISITED

In this part of the paper we return to the real world examples presented previously to exemplify meta model definition and evolution. Figure 1 shows the result of a clone detection analysis. The analysis requires the documentation text. We define the specific meta model to contain a *section* and a *topic* entity type. The latter contains the actual *text* as an attribute (unary relation). A *section* entity can contain further *section* and *topic* entities. The common meta model will contain a *document* entity and the *topic* and *section* entity types of the specific meta model. When mapping a concrete documentation to the common meta model, each paragraph is mapped to a *topic* with the corresponding *text* as an attribute. Each (sub-)section is mapped to a *section* entity, and each *section* and *topic* is structurally contained in either a *section* or the *document*.

Figure 2 shows the use of cross-references. Visualization are special analyses and require specific meta models. The cross-reference visualization requires the *section* entity and a new *refers* relation type between *topics*. The *topic* entity is not required and will be filtered, and the relation is lifted. The visualization shows the format type of the topics, which can either be text (as in the examples) or different image formats. Text topics are visualized as ellipsoid shapes, and various image formats using diamonds and parallelograms. The format type requires a new *type* relation for topics.

Figure 3 (left) visualizes meta-information: applicabilities, i.e., tags used to classify sections, and their tag categories. The corresponding specific meta model and, hence the common meta model, requires *applicability* and *category* entity types and a relation type modeling that an applicability belong to a certain category. Figure 3 (right) relates meta-information to an actual documentation: applicabilities, colored according to their categories, are related to sections and subsections (white boxes). Applicabilities are refined, which requires an *is refined by* relation type in this specific meta model and the common meta model. Mapping adds an *applicability* (*category*) entity for each unique tag (tag category) found in the documentation. It adds the corresponding *has a* relation between category and applicability entity types. Finally, it adds the *is tagged by* (*is refined by*) relations between applicabilities and sections (applicabilities).

Altogether, to capture the quality related information displayed in Figures 1–3 our common meta model

needs the following entity types: document, section (with uniqueness and title attributes), topic (with type and text attributes), applicability, and category. In addition to the unary relations referred to as attributes and the structural containment relation *contains* implicitly encoded in the structure trees, it also needs the following relation types between these entities: *similar to*, *cross reference*, *is tagged by*, *has a*, and *is refined by*. The ER diagram, which describes the (current) complete meta model is depicted in Figure 5.

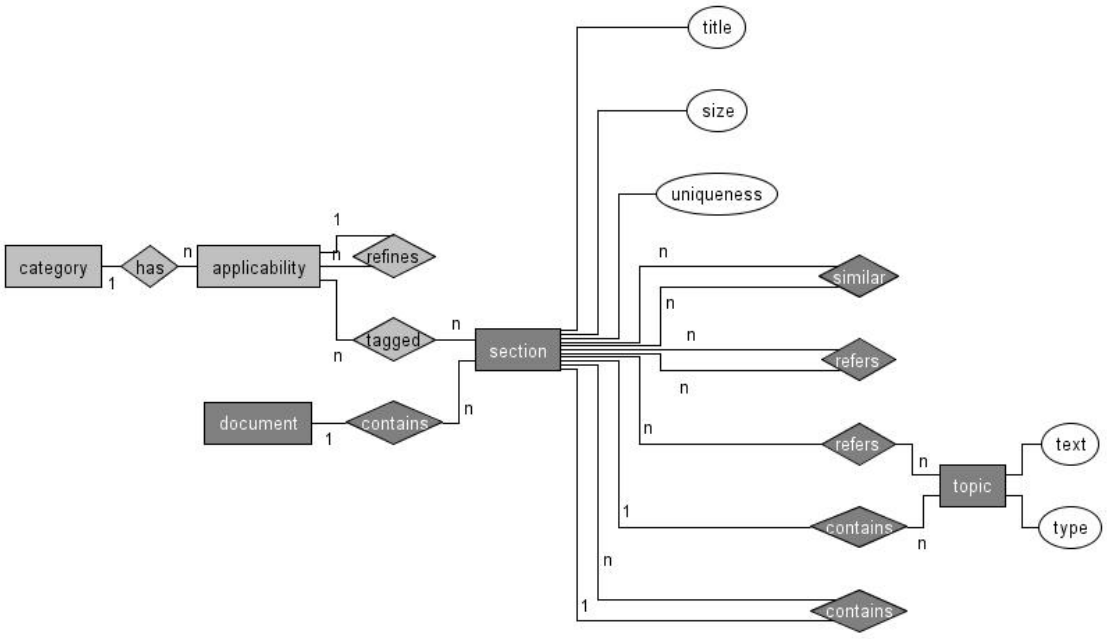

Figure 5: ER diagram of the current common IQ meta model: entities (boxes) with attributes (ellipsoids) and relations (diamonds) for capturing documentation data (dark grey with white labels) and meta-data (light grey with black labels).

RELATED WORK

Data and information quality are commonly considered multi-dimensional [9] and context-dependent concepts. A common definition of (information) quality provided by Juran [8] is "fitness for use".

Hargis et al. [6] discuss quality of technical information and define nine quality characteristics, for example, accuracy and clarity. These nine quality characteristics are divided into three categories: easy to use, understand, and find. The characteristics are quite broad, e.g., clarity does include concepts as conciseness and consistency.

Arthur and Stevens [1] present a framework to assess the "adequacy" of software maintenance documentation (i.e., technical information). Their framework defines quality as four Document Quality Indicators (DQIs): accuracy, completeness, usability, and extendibility. The DQIs are decomposed into Factors that refine a Quality, and Factors are decomposed into Quantifiers that are used to measure a Factor. A Factor of accuracy is consistency, which includes the Quantifiers conceptual and factual consistency. Wingkvist et al. [19] present a similar model for technical documentation, which is based on the idea of Key Performance Indicators that are considered an application of the more general Goal-Question-Metric [3] approach from (Software) Quality assessment.

There exist several quality frameworks for data and information quality, for example Wang and Strong [18], Stvilia et al. [17], Naumann [14], Chidamber and Kemerer [4]. Knight and Burn [10] provides an

overview of some of these, and Ge and Helfert [5] provide a review of the research in information quality (including frameworks). Most of the frameworks are hierarchical, and group information quality dimensions. For example, the framework by Wang and Strong [18] groups 16 quality dimensions, such as accuracy and relevance, into four dimensions (i.e., Intrinsic, Accessibility, Contextual, and Representational IQ). Many of the quality dimensions are common to several frameworks. Knight and Burn [10], for example, discovered that out of 12 surveyed frameworks, eight included accuracy and seven included consistency.

In many cases, information quality is assessed using surveys, e.g., Huang et al. [7] and Lee et al. [11]. The framework discussed by Arthur and Stevens [1] is assessed using a checklist approach, which is also used by Stvilia et al. [17]. Hargis et al. [6] suggests a combination of surveys and checklists. Wingkvist et al. [21] suggest to complement surveys and checklists with quality metrics (derived from software quality metrics) to automate the quality assessment. They acknowledge that the process cannot be fully automated, due to qualities such as ease of understanding, and suggests information testing as a complement [20].

The tool set and meta models discussed in this paper are inspired by research on software quality assessment. It relies on high-level abstractions, i.e., meta models, of software that contains enough information to support analyses. Meta models relevant in the software quality assessment community include the object-oriented FAMIX meta model developed in the European Esprit Project FAMOOS [2], and the Dagstuhl Middle Meta (DMM) model [12]. Strein et al. [16] presented the ideas of software meta model definition and evolution. It provides a sound foundation that information quality assessment can relate to.

CONCLUSION AND FUTURE WORK

This paper presents a software infrastructure for quality assessment and assurance of (technical) documentation. The infrastructure is based on a common information repository and readers that map to it, analyses that access and update it, and visualizations that help understanding the results it contains. We show how the use of meta models and meta meta models allows us to automatically generate the readers and the repository, and help make analyses and visualizations resilient to changes to the readers and repository. As a proof of concept, we successfully applied instantiations of the infrastructure to analyze and assess the quality of real world documentation, including the documentation of a mobile phone and a warship.

We are currently adding new analyses from existing quality models, such as the ones discussed in related work. We are also investigating how analyses that operate on syntactic and sematic levels of language can be used to complement analyses that operate on the structure of the information (e.g., XML). We are conducting evaluations of how accurately our analyses can be used to detect quality defects, and if a continuous quality assessment and assurance based on our infrastructure has any impact on the perceived quality of the documentation.

ACKNOWLEDGMENT

We would like to extend our gratitude to Applied Research in System Analysis AB (ARiSA AB, http://www.arisa.se) for providing us with the VizzAnalyzer tool and to Sigma Kudos AB, (http://www.sigmakudos.com) for providing us with their Content Management System (DocFactory) and raw data.

REFERENCES

[1] Arthur, J. D. and Stevens, K. T. Document Quality Indicators: A Framework for Assessing Documentation Adequacy. Journal of Software Maintenance, 4:129–142, September 1992.

[2] Bär, H., Bauer, M., Ciupke, O., Demeyer, S., Ducasse, S., Lanza, M., Marinescu, R., Nebbe, R., Nierstrasz, O., Przybilski, M., Richner, T., Rieger, M., Riva, C., Sassen, A., Schulz, B., Steyaert, P., Tichelaar, S. and Weisbrod, J. The FAMOOS Object-Oriented Reengineering Handbook, October 1999. Accessed via http://scg.unibe.ch/archive/famoos/handbook/hyperhandbook.pdf, June 2011.

[3] Basili, V.R., Caldiera, G. and Rombach, H. D. The goal question metric approach. Encyclopedia of Software Engineering. Wiley, 1994.

[4] Chidamber, S. R. and Kemerer, C. F. A Metrics Suite for Object-Oriented Design, IEEE Transactions on Software Engineering, 20(6):476–493, 1994.

[5] Ge, M. and Helfert, M. A Review of Information Quality Research: Develop a Research Agenda. Proceedings of the 12th International Conference on Information Quality, November 2007.

[6] Hargis, G., Carey, M., Hernandez, A. K., Hughes, P., Longo, D. and Rouiller, S. Developing Quality Technical Information: A Handbook for Writers and Editors. Upper Saddle River, NJ: Pearson Education, 7th printing, 2009.

[7] Huang, K. T., Wang, Y. R. and Lee, W. Y. Quality Information and Knowledge. Prentice Hall, Upper Saddle River, NJ, 1998

[8] Juran, J. Juran's Quality Control Handbook. McGraw-Hill, 5th edition, 1998.

[9] Klein, B. D. User Perceptions of Data Quality: Internet and Traditional Text Sources. Journal of Computer Information Systems, 41(4):5–15, 2001.

[10] Knight, S. A. and Burn, J. Developing a Framework for Assessing Information Quality on the World Wide Web. Informing Science, 8:159–172, 2005.

[11] Lee, Y. W., Strong, D. M., Kahn, B. K. and Wang, R. Y. AIMQ: A Methodology for Information Quality Assessment. Information & Management, 40 (2):133–146, 2002.

[12] Lethbridge, T. C., Tichelaar, S. and Ploedereder, E. The Dagstuhl Middle Metamodel: A Schema For Reverse Engineering. Proceedings of the International Workshop on Meta-Models and Schemas for Reverse Engineering (ateM 2003). Electronic Notes in Theoretical Computer Science, 94:7–18, May 2004.

[13] Löwe, W. and Panas, T. Rapid Construction of Software Comprehension Tools. International Journal of Software Engineering and Knowledge Engineering, 15(6):995–1026, 2005.

[14] Naumann, F. Quality-driven Query Answering for Integrated Information Systems. Springer-Verlag, Berlin, Heidelberg, 2002.

[15] Smart, K., Madrigal, J. and Seawright, K. The Effect of Documentation on Customer Perception of Product Quality. IEEE Transactions on Professional Communication, 39(3):157–162, September 1996.

[16] Strein, D., Lincke, R., Lundberg, J. and Löwe, W. An Extensible Meta-model for Program Analysis. IEEE Transactions on Software Engineering, 33(9):592–607, 2007.

[17] Stvilia, B., Gasser, L., Twidale, M. B. and Smith, L. C. A Framework for Information Quality Assessment. Journal of the American Society for Information Science and Technology, 58(12):1720–1733, 2007.

[18] Wang, R. Y. and Strong, D. M. Beyond Accuracy: What Data Quality Means to Data Consumers. Journal of Management of Information Systems, 12(4):5–33, 1996.

[19] Wingkvist, A., Ericsson, M., Löwe, W. and Lincke, R. A Metrics-based Approach to Technical Documentation Quality. Proceedings of the 7th International Conference on the Quality of Information and Communications Technology (QUATIC'2010), pp. 476–481, September 2010.

[20] Wingkvist, A., Ericsson, M., Löwe, W. and Lincke, R. Information Quality Testing, Lecture Notes in Information Processing (LNBIP), 64:14–26, September 2010.

[21] Wingkvist, A., Ericsson, M. and Löwe, W. Making Sense of Technical Information Quality – A Software-based Approach, Journal of Software Technology, 14(3):12–18, 2011.