The design and implementation of a software infrastructure for IQ assessment

Morgan Ericsson*, Anna Wingkvist and Welf Löwe

School of Computer Science, Physics, and Mathematics,
Linnaeus University,
351 95 Växjö, Sweden
E-mail: morgan.ericsson@lnu.se
E-mail: anna.wingkvist@lnu.se
E-mail: welf.lowe@lnu.se
*Corresponding author

Abstract: Information quality assessment of technical documentation is an integral part of quality management of products and services. Technical documentation is usually assessed using questionnaires, checklists, and reviews. This is cumbersome, costly and prone to errors. Acknowledging the fact that only people can assess certain quality aspects, we suggest complementing these with software-supported automatic quality assessment. The many different encodings and representations of documentation, e.g., various XML dialects and XML Schemas/DTDs, is one problem. We present a system, a software infrastructure, where abstraction and meta modelling are used to define reusable analyses and visualisations that are independent of specific encodings and representations. We show how this system is implemented and how it

1 reads information from documentations
2 performs analyses on this information
3 visualises the results to help stakeholders understand quality issues.

We introduce the system, the architecture and implementation, its adaptation to different formats of documentations and types of analyses, along with a number of real world cases exemplifying the feasibility and benefits of our approach. Altogether, our approach contributes to more efficient information quality assessments.

Keywords: information quality assessment; software-based analysis; technical documentation.

Reference to this paper should be made as follows: Ericsson, M., Wingkvist, A. and Löwe, W. (2012) 'The design and implementation of a software infrastructure for IQ assessment', *Int. J. Information Quality*, Vol. 3, No. 1, pp.49–70.

Biographical notes: Morgan Ericsson is an Associate Professor in Computer Science at Linnaeus University, Växjö, Sweden. After completing his PhD in 2008, he was a Postdoctoral Researcher at the Department of Information Technology, Uppsala University, Sweden. His main research interest is how methods and tools from software technology and database research can be used

to assess quality. As a skilled programmer, he has contributed to the development of programming models and frameworks for software and information analysis.

Anna Wingkvist is an Associate Professor in Computer Science at Linnaeus University, Växjö, Sweden. Her academic background is in information systems development, methodological and research methods reasoning, and project management. Since completing her PhD in 2009, her scientific interest and publications are mainly in the information quality domain. In 2011, she was awarded a prestigious, multiyear research grant from the Swedish Governmental Agency for Innovation Systems, VINNOVA.

Welf Löwe is a Professor in Computer Science. He has held a Chair in Software Technology at Linnaeus University since 2002. From numerous projects in industry and research he has experiences in software and information analysis as well as software and information quality management in general. He is the co-founder of ARiSA, a spin-off company of his chair, that maintains and provides the open source software and information analysis tool VizzAnalyzer.

1 Introduction

Quality assessment and assurance is an important part of the documentation (technical information) production process (Hargis et al., 2009). A lack of quality reduces not only the value of documentation, but also that 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 (Smart et al., 1996).

Quality depends not only on the documentation 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 quality assessment and assurance are continuous processes. In this article we present a software infrastructure that can be used to define analyses and visualisations to assess and communicate the quality of documentation. The (software) infrastructure, VizzAnalyzer, is available as Open Source Software and can be downloaded from the ARiSA website (http://www.arisa.se/tools.php). VizzAnalyzer 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 visualisations, and mappings from real world documentations.

The article is organised as follows. We begin by providing examples of how information quality assessment can be supported by automatic analyses and visualisations. We discuss lessons learned from implementing these, such as the need to support different representations and encodings of documentation. We introduce abstraction and meta modelling as tools to help define reusable analyses and visualisations independent of encoding and representation. We define three levels of abstraction, and discuss mappings between these. We then present the architecture of our software infrastructure that is based on the three levels of abstraction. Further, we

discuss the use of incomplete models and model evolution. Finally, we discuss related work, present conclusion and outline future research directions.

2 Software-supported quality assessment in practice

In this section we present four real world examples where we use the software infrastructure to automatically assess the quality of documentations. In our case we had access to (non-classified parts of) a warship documentation and a mobile phone documentation. Each example discusses a potential quality issue, the analysis we used to detect it, how the results were visualised, and the outcome.

2.1 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 providing a context and thereby reducing the need for 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 comparing texts of the documentations' substructures (referred to as sections).

We analysed 913 XML documents (sections) from the warship documentation and found that only six 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 depicts the result of the clone detection. Each box corresponds to a section, the colour represents the degree of uniqueness of a section, and the edges indicate similarities between sections.

In order to perform clone detection, we need the individual sections of a documentation and the text these contain.

2.2 Reference analysis

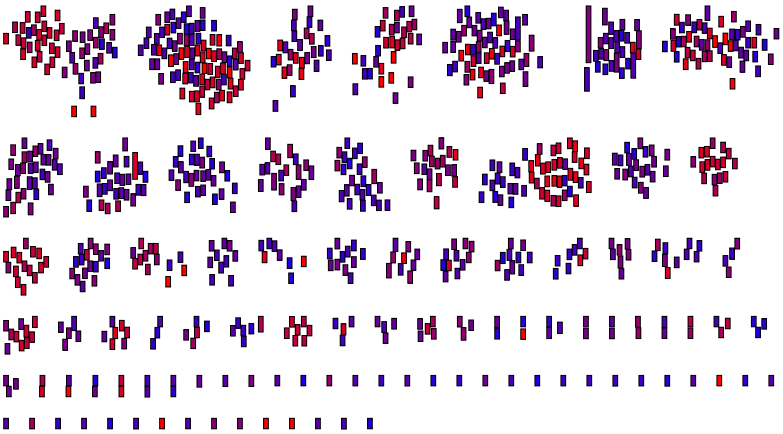
Cross-references are used to relate different parts of a documentation to each other. If too many cross-references are needed to understand the content, especially if non-local and forward references are used, information in the sections is not self-contained. Consequently, it can be difficult to read and understand the documentation. A high degree of cross-references indicates that the documentation structure might be suboptimal.

We applied a reference analysis to the warship documentation (which was also used for text clones). Figure 2 shows the outcome of this analysis. Each box corresponds to a section and the colour represents which substructure of the documentation (referred to as chapters) it belongs to. The distance between two boxes is proportional to the number of cross-references between them. The top figure shows the whole documentation while the bottom figure shows a part of the documentation with a large degree of cross-references across chapters. This part is problematic since it contains sections from

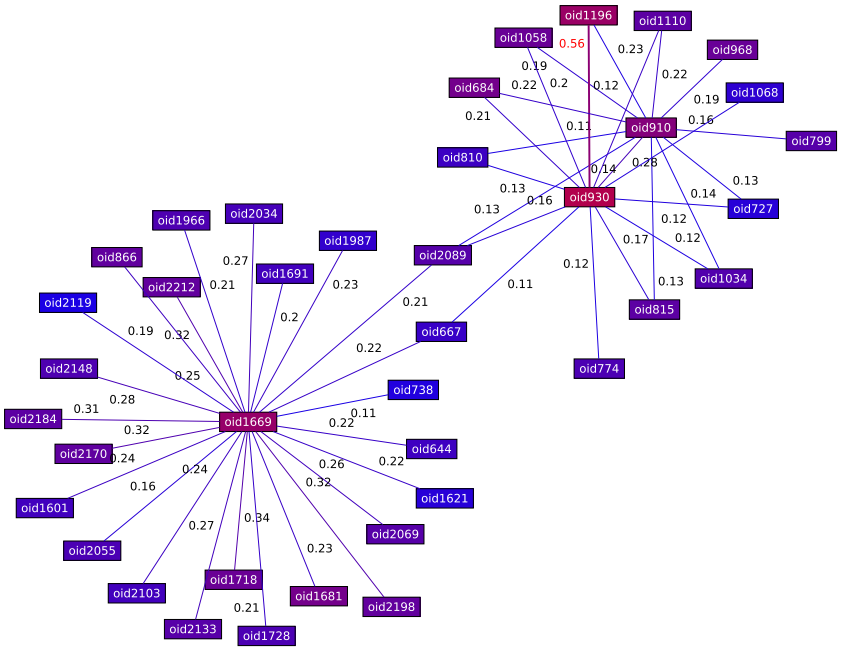
eight different chapters of the documentation (eight different colours of boxes) with many cross-references between them.

In order to perform a reference analysis, we need access to the individual sections of a documentation, the chapters these are contained in, and the cross-references.

Figure 1 (a) The result of a clone detection analysis of the technical information of a warship is visualised as clusters of similar documents (b) Each cluster can be viewed in more detail (see online version for colours)

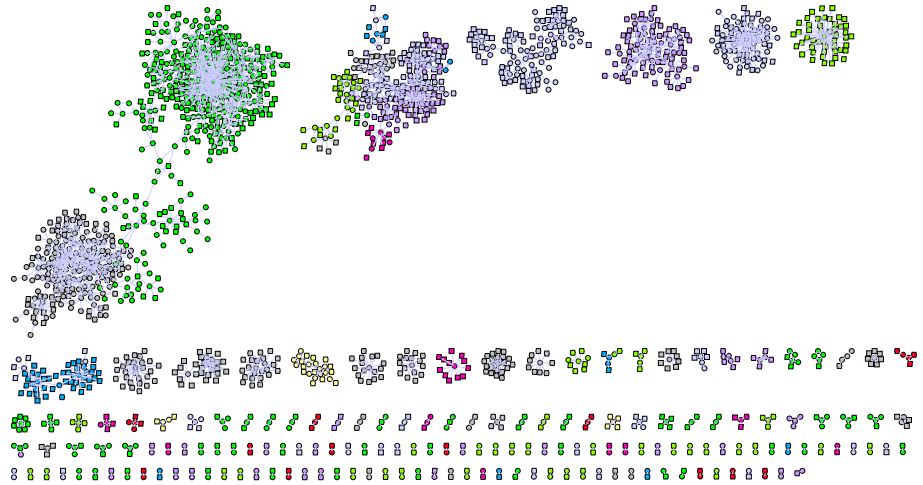


(a)

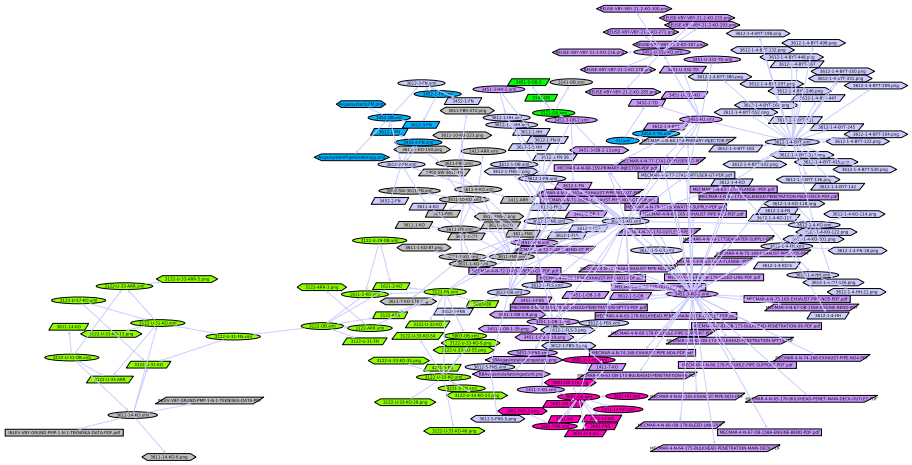


(b)

Figure 2 Visualisations of cross-reference analysis results, (a) cross-reference cluster overview and (b) detailed view zoomed into one problematic cluster (see online version for colours)



(a)



(b)

2.3 Use of meta-information

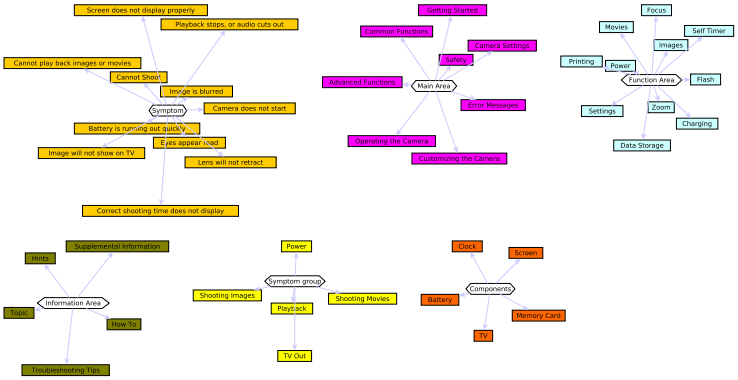
Meta-information such as tags and keywords can be used to improve usability and accessibility of documentations. However, improper definition of these, for example attaching a keyword to the wrong section of the documentation, can reduce usability.

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 text. Each tag belongs to a category. The analysis extracts all the applicabilities

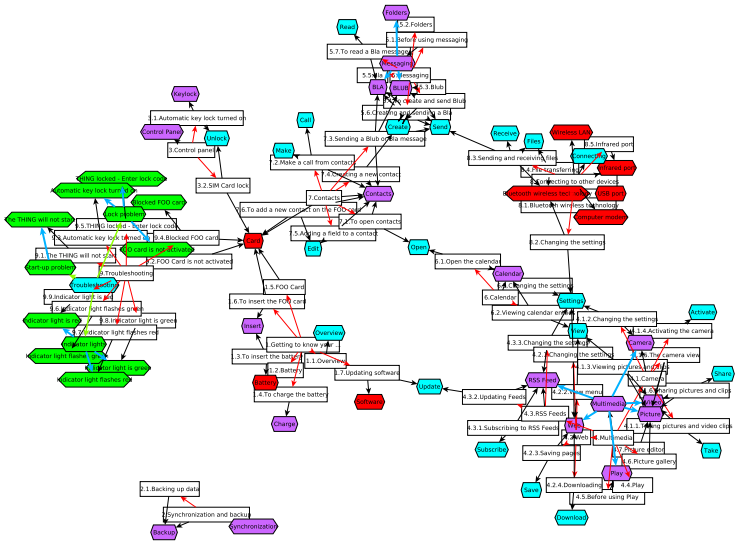
for each category, and the applicabilities (and categories) for each section of the documentation.

We analysed 12,286 XML documents from the mobile phone documentation and found that some categories were concentrated to certain chapters 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. Figure 3(a) shows the applicability tags and the categories, and Figure 3(b) shows applicability tags and sections of the documentation.

Figure 3 A meta-information analysis of a mobile phone technical documentation, (a) the visualisations show the structure of the meta-information and (b) the relation to information (see online version for colours)



(a)



(b)

Note: The colour of an applicability tag identifies the category it belongs to.

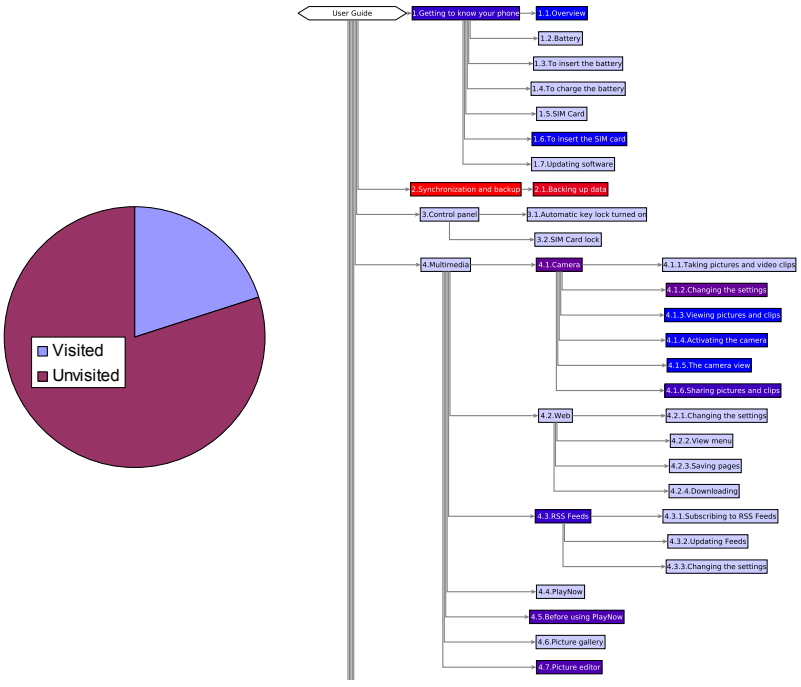
In order to perform an analysis of the meta-information use, we need access to the individual sections, the chapters they are contained in, and the tags attached to the sections and their categories.

2.4 Test coverage

Certain qualities are hard to analyse fully automatically. These require (human) test readers who approve sections of a documentation. The proofreading can be supported by analysis tools that analyse the coverage of the proofreading as well as how much time a test reader required to complete each of the sections of the documentation.

We implemented coverage analysis as an extension of a documentation browser; each section entry and exit is logged with a time stamp. This log is later combined with the documentation structure to form an understanding of how the test readers navigated the documentation. The combined information shows, for example, which documents are visited, for how long, in which order, and so on.

Figure 4 Visualisations of test coverage analysis, (a) statistics view and (b) graph view (see online version for colours)



(a) (b)

We applied a test coverage analysis to a subset of 70 documents of the mobile phone documentation where we asked the test readers to find information about a certain use case (how to take a picture and send it as an MMS). The results are depicted by Figure 4(b), which shows a test reader's behaviour during testing. It depicts the document structure (boxes correspond to individual sections belonging to chapters) and

encodes the time spent on individual section with different colours: light-blue boxes are sections not visited at all; the colour gradient from dark blue to red corresponds to how long the sections are visited (dark blue representing the minimum and red the maximum amount of time). In this example, the colour gradient represents the span from 1 to 20 seconds.

In a real testing situation it is possible to determine whether or not the documentation is suitable for a certain uses case (a question that a user might want to get answers to), whether or not a certain section is well-written and easy to grasp, or whether or not the structure of the sections are appropriate for navigating the documentation. However, any such conclusions about the whole documentation require that all intended use cases are covered and that the whole documentation is (potentially) proofread. Figure 4(a) shows the coverage of our use case for the mobile phone documentation. Only 17 of the 70 sections were visited, which results in 20% test coverage. Hence, 80% of the documentation is not proofread. In a real testing situation, the quality of this test would be considered insufficient due to low coverage of the documentation system. Alternatively, if all intended use cases are covered, the documentation contains redundant and 'useless' information. Figure 4(b) can be used to determine which parts of the documentation system should be covered by other test cases (and excluded from the system, respectively).

In order to support testing with the described analyses, we need access to the individual sections and the chapters they are contained in, along with the time stamps attached to the sections.

2.5 Lessons learned

We implemented the text clone, reference, meta-information, and coverage analyses to investigate how software can be used to help automate quality assessment [see Wingkvist et al. (2010a, 2010b) for a more detailed analysis]. We learned three important lessons during this implementation that are summarised below.

First, documentations can be represented and encoded in different ways. For example, the documentations we used in the examples use two different X(A)ML dialects. Analyses should be possible to reuse regardless of the many documentation encodings. We do not want to (re-)implement the analyses (and the visualisations) for each possible documentation representation and encoding.

Second, certain entities and relations are of common interest to more than one analysis (e.g., the sections and their containment in chapters) while others are unique to a specific analysis (e.g., the tagging or the time logging). The entities and relations extracted from the documentation (representation) that are of common interest can be re-used across a set of analyses.

Third, it is difficult to define a fixed set of entities and relations that provide sufficient information for all possible analyses. Consider the example in Section 2.1 as an initial quality analysis tool and the examples in the subsequent Sections 2.2 to 2.4 as a series of tool extensions. While clone detection only required the section entities and the text contained, we needed to add chapter entities and cross-reference relations for the reference analysis, then meta-information, and then time log information. We are certain that new analyses will require additional entities and relations. Hence, the set of entities and relations that are extracted from documentations to support analyses should be flexible and support future extensions.

The three lessons learned can be considered requirements on software to support quality assessment. We used these requirements to design and implement the software infrastructure that is presented in this article.

3 Abstractions, models and meta models

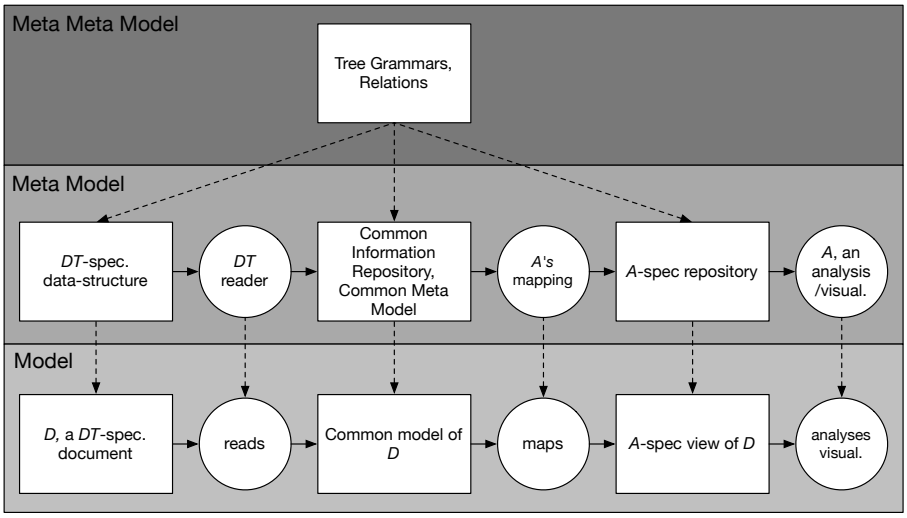
The example analyses that we implemented (cf., Section 2) show that there is an overlap of the parts of a documentation that we use to assess quality between different analyses. However, the examples also show that different encodings and representations used by different real world documentations make reuse difficult. For example, assume that we want to apply the cross-reference analysis (cf., Section 2.2) to two XML-compatible documents, one defined using a custom XML Schema and the other using XHTML. The analysis needs access to sections and cross-references, and these are represented in different ways in the two documents, so we cannot reuse the same implementation.

However, we can implement a reusable cross-reference analysis by defining what entities and relationships in the documentation that it needs, and implement mappings from the specific formats (the XML Schema and XHTML) to this 'abstract' documentation format that is specific to the analysis. We consider the abstract documentation format an abstraction – a model – of the actual documentation format. In most cases, documentation formats are already abstractions, for example, XML abstracts layout information, such as font sizes and styles. We extend this idea, and consider any documentation as a series of models, and mappings between these.

In order to be able to map between models, i.e., create new abstractions, there is a need to define what elements are valid in the model. We rely on meta modelling, and use higher level models, meta models, to define models on lower levels. For example, an XML document is often defined by a specific XML Schema or Document Type Definition (DTD). The XML document can be considered a model of a documentation, and the XML Schema or DTD is the meta model, that defines what elements the model can contain. Each model is defined by a meta model, so there is abstraction between models and meta models.

We define abstractions of meta models as well as models, so there is a need to map between meta models. Consequently, there is a need to define the valid elements of a meta model, i.e., there is a need for a *meta* meta model. This meta meta model defines what can be expressed by a meta model. For example, based on our experience from the four example analyses in Section 2 we define three document models with corresponding meta models. First, there is an actual document-specific model that corresponds to the actual format/schema used. This model is mapped to a more abstract, general model, that can represent (abstractions of) any document-specific model that we need to support. Finally, we use an analysis-specific model to keep analyses reusable, even if the 'common' model is changed to support other document-specific models. We refer to the latter as (meta) model evolution, and describe it in detail in Section 6. Each of the models represents documents that all have tree structures (a document contains a number of sections) with relationships between the nodes (cross-references). So, any meta model is defined by a specific tree grammar that determines which tree structures are valid and a set of relationships between nodes. The meta meta model consists of a definition of how a tree grammar is expressed and how relationships are defined. Figure 5 depicts the model layers and the levels of abstractions within the model and meta model layers.

Figure 5 The three layers of models (meta modelling) and the three abstraction levels of the model and meta model layers



Note: Models are depicted by rectangles and mappings are depicted by circles. D represents a documentation, DT a documentation type, and A an analysis.

The three meta models and the mappings between them are described in detail in Section 4. Section 6 describes the meta meta model and meta model evolution. An implementation based on these models is described in Section 5.

4 Meta models and mappings

This section discusses the Document-Specific, Common, and Analysis-Specific Meta Models, as well as the mappings between these in more detail.

4.1 A Document-Specific Meta Model

A documentation follows specific rules and conventions, specified either implicitly or explicitly. We refer to this as the *Document-Specific Meta Model*. In general, a documentation consists of entities structurally contained in each other and relations between them.

We define a Document-Specific Meta Model for a documentation type DT as $M^{DT} = (G^{DT}, R^{DT})$. G^{DT} is a tree grammar that specifies 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 defines 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 $T \subseteq T^{DT}$. Regular expressions are

either sequences $(t_1, \ldots, t_k)$, iterations (t^*), or selections $(t_1 | \ldots | t_k)$. R^{DT} denotes a set of relations over model entities, $R^{DT} = R_1^{DT}, \ldots, R_n^{DT}$, 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 Document-Specific Meta Model that corresponds to a specific XML *DTD*. The tree grammar $G^{DTD} = (T^{DTD}, P^{DTD}, document^{DTD})$ defines the containment structure trees of the DTD (i.e., the information set regardless of their XML encoding). Figure 7(a) depicts an example of a containment structure tree. G^{DTD} contains entity types (T^{DTD}) 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}. The productions P^{DTD} define the structural containment between the entitiy types (T^{DTD}) in structure trees with $document^{DTD}$ entities as root.

R^{DTD} contains a binary refers relation *refers*^{DTD}: *ref*^{DTD} $\times$ (*section*^{DTD} $\cup$ *subsection*^{DTD} $\cup$ *figure*^{DTD}) which states that a reference can refer to a section, a subsection, or a figure. R^{DTD} also contains 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.

4.2 The Common Meta Model

The *Common Meta Model* describes a family of models by abstracting document-specific details. We denote the Common Meta Model by $M = (G, R)$. The only major difference between the Common Meta Model and a Document-Specific Meta Model is that relations R must contain a 0th element, a numerical identifier to maintain the document order of relation tuples.

Example 2: Let $M = (G, R)$ be the Common Meta Model that we will use in our running example. $G = (T, P, \textit{document})$ defines the common containment structure trees that can contain entities for *document* and *section*. Figure 7(b) depicts an example of a such a tree. Productions P define the structural containment of these trees. R contains relations $refers : Num \times section \times section$, $caption : Num \times (document \cup section) \times String$ and $content : Num \times section \times String$. These relations use *String* to represent the character sequence of a caption or content, and *Num* to represent the numerical type of an identifier that encodes document order.

4.3 Mapping from a Document-Specific Meta Model to the Common Meta Model

The Common Meta Model is an abstraction of different Document-Specific Meta Models. For each such Document-Specific Meta Model, the abstraction is defined by a *Document-Specific Mapping*, α^{DT}. This mapping itself is defined by mapping the grammar G^{DT} to the Common Meta Model grammar G and relations R^{DT} to the Common Meta Model relations R.

α^{DT} is defined by mapping the entity types of a Document-Specific Meta Model to those of the Common Meta Model, $\alpha^{DT}: T^{DT} \rightarrow T$. The Documentation-Specific 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 (an onto mapping) 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., not surjective), and some Document-Specific Meta Model entity and relation types may be ignored (i.e., not complete).

Example 3: The mapping α^{DTD} maps entity and relation types of the Document-Specific Meta Model 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 $\alpha^{DTD}(content^{DTD}) = content$.

4.4 Analysis-Specific Meta Models and their mapping from Common Meta Models

Analyses might directly traverse the documents represented by the Common Meta Model, extract the required information, and perform computations. We introduce analysis-specific *views* on the Common Meta Model, i.e., abstractions of the model. A view provides the exact information required by a specific analysis. If a set of analyses uses the same information, they share the same view.

Views are abstractions of the Common Meta Model. Formally, a view is a meta model that is specific for an analysis, A, and is defined 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.

View model construction follows the same principles as the mapping and abstraction from Document-Specific to Common Meta Models. We ignore certain 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 Depth-First-Search (DFS) rank from the Common Meta Model 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, the 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 [cf., Löwe and Panas (2005) for applicable strategies].

Example 4: Define an analysis, *Coupling*, which computes the relative coupling of a section. The analysis computes the ratio of references within section s and its subsections, denoted $\mathit{localRefs}(s) = |\{\mathit{refers}(s1, s2)\colon\ \mathit{contains}^*(s, s1) \wedge \mathit{contains}^*(s, s2)\}|$, and all references incoming to and outgoing from s, denoted $allRefs(s) = |\{refers(s1, s2)\colon\ \mathit{contains}^*(s, s1) \vee \mathit{contains}^*(s, s2)\}|$. In short, $\mathit{Coupling}(s) = \mathit{localRefs}(s)/\mathit{allRefs}(s)$. An appropriate Analysis-Specific Meta Model $V^{Coupling}$ contains the entity types $\mathit{document}^{Coupling}$ and $\mathit{section}^{Coupling}$, and the relation type $\mathit{refers}^{Coupling}\colon \mathit{Void} \times \mathit{section}^{Coupling} \times \mathit{section}^{Coupling}$ with the mappings $\alpha^{Coupling}(\mathit{document}) = \mathit{document}^{Coupling}$, $\alpha^{Coupling}(\mathit{section}) = \mathit{section}^{Coupling}$, and $\alpha^{Coupling}(\mathit{refers}) = \mathit{refers}^{Coupling}$. Note that 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 $\mathit{refers}\colon \mathit{Num} \times \mathit{section} \times \mathit{section}$, along with other irrelevant entities and relations.

Figure 7 shows the Document-Specific Meta Model from Example 1, and how it is mapped to the Common Meta Model (Example 2-3) and an Analysis-Specific Meta Model for the Coupling analysis (Example 4). In this example, the Coupling is 0.5 for Section 2 and 0 for all other sections.

5 An infrastructure for quality assessment

In Section 3 we introduced a system where we use abstraction and meta modelling to to define analyses to assess and visualise quality. We defined meta models for the Document-Specific, Common and Analysis-Specific models, and introduced mappings between these in Section 4. In this section, we present a software infrastructure that is an implementation of this system.

The infrastructure consists of three major components:

1 the *Common Information Repository* that captures models of the documentations

2 *readers* that map documentations to the repository

3 *analyses* that assess and modify the repository and visualisations that present interactive views of the repository to stakeholders, e.g., technical writers, product managers, etc.

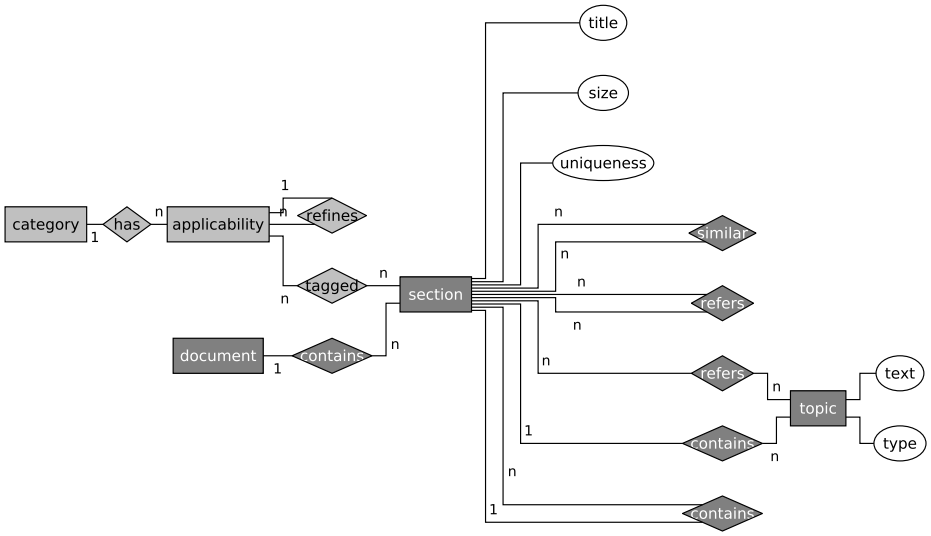
5.1 The Common Information Repository

The Common Information Repository is the centre of the infrastructure. It captures abstractions (models) of documentations, and any analysis or visualisation uses data stored in it. As discussed in Section 3 there exist several different models of a documentation. The Common Information Repository captures models of documentations according to the Common Meta Model. Models that are described by various Analysis-Specific Meta Models are computed from the repository (cf., Section 4 for details).

The meta models consists of entities and relationships, and we use Entity/Relationship (E/R) diagrams to describe them (cf., Section 6 for details). Figure 6 depicts the Common Meta Model for the analyses described in Section 2. The implementation (classes and methods in Java) is automatically generated from

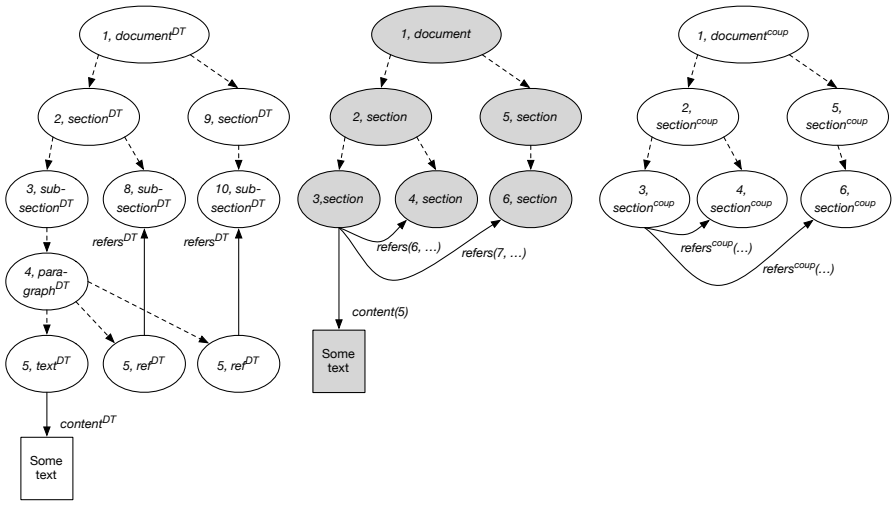
such E/R diagrams. We use the Common Meta Model as a configuration parameter to instantiation of the infrastructure. This way, the meta model can be adapted to the actual documentation types that we want to analyse.

Figure 6 Entity/relationship diagram of the Common Meta Model used in Section 2



Note: 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)

Figure 7 Mapping of Document-Specific to Common (grey) and Analysis-Specific models



(a) (b) (c)

Note: Entities of the three models are labelled with their *id* (DFS rank) and *type*. Dashed edges display structural *contains*, solid edges *content* and *refers* relations.

The Common Information Repository can be considered as a database that captures entities and relations between these entities (including the containing tree structure). The Analysis-Specific Meta Model views are similar to views in SQL.

5.2 Readers that map documentations to the repository

To apply predefined analyses and visualisations, there is a need to map documentations that are captured in a specific format, such as XML, to the Common Meta Model. This process can be considered a two-stage process, where we first create a model of the documentation according to the Document-Specific Meta Model. We then apply a mapping to map this model to entities and relationships that exist in the Common Meta Model. The mapping of the containment trees is defined recursively. Starting at the root, we traverse the document-specific containment tree in DFS order. We create new Common Meta Model entities for Document-Specific Meta Model entities of types that have a mapping defined. These are referred to as the relevant entities. Any irrelevant Document-Specific entities are ignored.

A generic event-based interface between Document-Specific and Common Meta Models and an abstract algorithm to map model entities are given in Procedures 1, 2, and 3. A tree-walker (cf., 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.

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 1

```
1: procedure generateTreeEvents(n = (id; t))
2:     startNode(n)
3:     for c ∈ childrenOfNode(n) do
4:         generateTreeEvents(c)
5:     end for
6:     finishNode(n)
7: end procedure
```

Procedure 2

```
1: procedure startNode(n = (id; t^DT))
2:     if α^DT(t^DT) is defined then
3:         create new entity n' ← (id; α^DT(t^DT))
4:         append n' to children of Stack.top
5:         Stack.push(n')
6:     end if
7:     map(n) ← Stack.top                    ▷ defines closest relevant ancestor of n
8: end procedure
```

Procedure 3

```
1: procedure finishNode(n = (id; t^DT))
2:     if α^DT(t^DT) is defined then
3:         Stack.pop
4:     end if
5: end procedure
```

A Document-Specific relation is a set of tuples $R_i^{DT}(n_1, \ldots, n_k)$ 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 $R_i^{DT}\colon t_1^{DT} \times \ldots \times t_k^{DT}$ 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 $R_j^{DT}(n_1, \ldots, n_k)$ will have a correspondence in the Common Meta Model and R_i can be defined over those entities. However, the following three situations make this mapping more complex.

First, if α^{DT} is not defined for a type of an entity n_i in $R_i^{DT}(n_1, \ldots, n_k)$, 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 $R_i\colon Num \times t_1 \times \ldots \times t_k$ is a numerical identifier which we implicitly generate. Each tuple $R_i^{DT}(n_1 = (id, t^{DT}) \ldots n_k)$ is mapped to $R_i(id, map(n_1), \ldots, map(n_k))$, 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

```
1: procedure generateRelationEvents(R^DT)
2:     for all R_i^DT ∈ R_S do
3:         if α^DT(R_i^DT) is defined then
4:             for all R_i^DT(n_1, ..., n_k) do
5:                 newRelationTuple(a^DT(R_i^DT), (n_1, ..., n_k))
6:             end for
7:         end if
8:     end for
9: end procedure
```

Procedure 5

1: **procedure** newRelationTuple($R_i, (n_1, \ldots, n_k)$)
2: **for all** $n_j \leftarrow (id, t_j^{DT}) \in n_1, \ldots, n_k$ **do**
3: **if** j = 1 **then**
4: $r0 \leftarrow id$
5: **end if** ▷ create 0th element of target relation
6: **if** $(t_j^{DT} = String \vee t_j^{DT} = Num) \wedge t_j = Void$ **then** ▷ drop string or numerical value
7: $r_j \leftarrow Void$
8: **else if** $t_j^{DT} = t_j = String \vee t_j^{DT} = t_j = Num$ **then** ▷ copy string or numerical value
9: $r_j \leftarrow n_j$
10: **else** ▷ lift the relation to the closes relevant ancestor
11: $r_j \leftarrow map(n_j)$ ▷ map captures the closest relevant ancestor of n_j as defined in Procedure 2
12: **end if**
13: **end for**
14: add tuple $r_0, r_1, \ldots, r_k$) to relation R_i
15: **end procedure**

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.

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. For many common formats, such as XML, there exist predefined processing libraries and frameworks. In such cases, it is trivial to define a reader.

5.3 Reusable analyses and visualisations

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 use visualisations to communicate information to stakeholders about the documentation and the analyses we perform. A visualisation defines a mapping from the information contained in the repository to a visual domain. The visual domain contains objects with various attributes, such as shapes, colours, patterns, and positions. The mapping between the two brings meaning and context to the visual objects in terms of documentation and quality.

A visualisation is a specific kind of analysis that operates on specific views of the repository. The mappings created by these analyses can be configured. We map repository entities (of different types) to visual objects (of different shape or colour) and map their quality attributes to visual attributes of these objects.

6 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 (and analyses) that only provide (relies on) entities and relations contained in the current Common 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 an analysis A cannot be applied to 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 Analysis-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 to 5 in Section 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, there is a 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 Procedures 1 and 4, no additional programming is needed when reusing an existing documentation type. We need to specify the missing entity and relation types as relevant in α^{DT}, and the Common Information Repository and models are generated automatically. New documentation types require specific implementations of Procedures 1 and 4.

Example 5: Assume the Common Meta Model from Example 2 that contains *document* and *section* entities and *refers* and *contains* relations, and the Analysis-Specific Meta Model from 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 $section ::= (section|paragraph)^*$.

We can define a new Analysis-Specific Meta Model $V^{Complexity}$ with a corresponding Analysis-Specific Mapping $\alpha^{Complexity}$ from the new Common Meta Model. 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 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 paragraphs are now the relevant ancestors of *ref* entities in the Common Meta Model. However, when applying $\alpha^{Coupling}$ to the 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 $refers^{Coupling} \square Void \times section^{Coupling} \times section^{Coupling}$ 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.

The effects of changing the Common Meta Model are often filtered by subsequent Analysis-Specific Meta Models. 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 only 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.

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 $X ::= \ldots$ 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 introduced to the 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.

7 Related work

Data and information quality are commonly considered multi-dimensional (Klein, 2001) and context-dependent concepts. A common definition of (information) quality provided by Juran (1998) is 'fitness for use'.

Hargis et al. (2009) discuss quality of 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 (1992) present a framework to assess the 'adequacy' of software maintenance documentation. 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. (2010b) 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 (Basili et al., 1994) approach from (software) quality assessment.

There exist several quality frameworks for data and information quality, for example, Wang and Strong (1996), Stvilia et al. (2007), Naumann (2002) and Chidamber and Kemerer (1994). Knight and Burn (2005) provides an overview of some of these, and Ge and Helfert (2007) 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 (1996) 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 (2005), 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. (1998) and Lee et al. (2002). The framework discussed by Arthur and Stevens (1992) is assessed using a checklist approach, which is also used by Stvilia et al. (2007). Hargis et al. (2009) suggests a combination of surveys and checklists. Wingkvist et al. (2011) 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 (Wingkvist et al., 2010a).

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 (Bär et al., 1999), and the Dagstuhl Middle Meta (DMM) model (Lethbridge et al., 2004). Strein et al. (2007) presented the ideas of software meta model definition and evolution. It provides a sound foundation that information quality assessment can relate to.

8 Conclusions and future work

This article 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 visualisations 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 visualisations resilient to changes to the readers and repository. As a proof of concept, we successfully applied instantiations of the infrastructure to analyse 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 developing new analyses based on real world requirements in collaboration with industry partners. We are also investigating how analyses that operate on syntactic and semantic levels of language can be used to complement analyses that operate on the structure of the information (e.g., XML). We have integrated natural language processing toolkits in the readers and developed (meta) models that support natural language structures. 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.

Acknowledgements

The research presented in this article was in part funded by the Knowledge Foundation (Grant No. 2011-0203) and Swedish Governmental Agency for Innovation Systems (VINNOVA) (Grant No. 2011-01351). We would also 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.

A version of this article was presented at the 16th International Conference on Information Quality.

References

Arthur, J.D. and Stevens, K.T. (1992) 'Document quality indicators: a framework for assessing documentation adequacy', *Journal of Software Maintenance*, September, Vol. 4, No. 3, pp.129–142. ISSN 1040-550X.

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. (1999) 'The famoos object-oriented reengineering handbook' [online] http://www.iam.unibe.ch/ famoos/handbook/ (accessed October).

Basili, V.R., Caldiera, G. and Rombach, H.D. (1994) 'The goal question metric approach', in *Encyclopedia of Software Engineering*, Wiley.

Chidamber, S.R. and Kemerer, C.F. (1994) 'A metrics suite for object-oriented design', *IEEE Transactions on Software Engineering*, Vol. 20, No. 6, pp.476–493.

Ge, M. and Helfert, M. (2007) 'A review of information quality research – develop a research agenda', in *Proceedings of the 12th International Conference on Information Quality*, November.

Hargis, G., Carey, M., Hernandez, A.K., Hughes, P., Longo, D. and Rouiller, S. (2009) *Developing Quality Technical Information? A Handbook for Writers and Editors*, Pearson Education, Upper Saddle River, NJ, ISBN 0131477498.

Huang, K-T., Wang, Y.R. and Lee, W.Y. (1998) *Quality Information and Knowledge*, Prentice Hall, Upper Saddle River, NJ, ISBN 0-130101-41-9.

Juran, J. (1998) *Juran's Quality Control Handbook*, 5th ed., McGraw-Hill, New York, NY, USA.

Klein, B.D. (2001) 'User perceptions of data quality: Internet and traditional text sources', *Journal of Computer Information Systems*, Vol. 41, No. 4, pp.5–15.

Knight, S.A. and Burn, J. (2005) 'Developing a framework for assessing information quality on the World Wide Web', *Informing Science*, Vol. 8, No. 1, pp.159–172.

Lee, Y.W., Strong, D.M., Kahn, B.K. and Wang, R.Y. (2002) 'Aimq: a methodology for information quality assessment', *Information & Management*, Vol. 40, No. 2, pp.133–146.

Lethbridge, T.C., Tichelaar, S. and Ploedereder, E. (2004) 'The dagstuhl middle metamodel', in *Proceedings of the International Workshop on Meta-Models and Schemas for Reverse Engineering (ateM 2003)*, May, Vol. 94, pp.7–18.

Löwe, W. and Panas, T. (2005) 'Rapid construction of software comprehension tools', *International Journal of Software Engineering and Knowledge Engineering*, Vol. 15, No. 6, pp.995–1026.

Naumann, F. (2002) *Quality-driven Query Answering for Integrated Information Systems*, Springer-Verlag, Berlin, Heidelberg, ISBN 3-540-43349-X.

Smart, K., Madrigal, J. and Seawright, K. (1996) 'The effect of documentation on customer perception of product quality', *IEEE Transactions on Professional Communication*, September, Vol. 39, No. 3, pp.157–162, ISSN 0361-1434, doi: 10.1109/47.536264.

Strein, D., Lincke, R., Lundberg, J. and Löwe, W. (2007) 'An extensible meta-model for program analysis', *IEEE Trans. Software Eng.*, Vol. 33, No. 9, pp.592–607.

Stvilia, B., Gasser, L., Twidale, M.B. and Smith, L.C. (2007) 'A framework for information quality assessment', *JASIST*, Vol. 58, No. 12, pp.1720–1733.

Wang, R.Y. and Strong, D.M. (1996) 'Beyond accuracy: what data quality means to data consumers', *J. Manage. Inf. Syst.*, Vol. 12, No. 4, pp.5–33, ISSN 0742-1222.

Wingkvist, A., Ericsson, M., Löwe, W. and Lincke, R. (2010a) 'Information quality testing', *Lecture Notes in Information Processing (LNBIP)*, September, Vol. 64, pp.14–26.

Wingkvist, A., Ericsson, M., Löwe, W. and Lincke, R. (2010b) 'A metrics-based approach to technical documentation quality', in *Proceedings of the 7th International Conference on the Quality of Information and Communications Technology*, pp.476–481.

Wingkvist, A., Ericsson, M. and Löwe, W. (2011) 'Making sense of technical information quality – a software-based approach', *Journal of Software Technology*, Vol. 14, No. 3, pp.12–18.