Book Title
Book Editors
IOS Press, 2003

Validating Documentation With Domain Ontologies

Leonid Kof [1], Markus Pizka
Fakultaet fuer Informatik, Technische Universitaet Muenchen, Munich, Germany.

Abstract. Do we always use the same name for the same concept? Usually not. While misunderstandings are always troublesome, they pose particularly critical problems in software projects. Requirements engineering deals intensively with reducing the number and scope of misunderstandings between software engineers and customers. Software maintenance is another important task where proper understanding of the application domain is vital. In both cases it is necessary to gain (or regain) domain knowledge from existing documents that are usually inconsistent and imprecise.

This paper proposes to reduce the risk of misunderstandings by unifying the terminology of the different stakeholders with the help of an ontology. The ontology is constructed by extracting terms and relations from existing documents. Applying text mining for ontology extraction has an unbeatable advantage compared to manual ontology extraction: Text mining detects terminology inconsistencies before they are absorbed in the ontology. In addition to this, the approach presented in this paper also introduces an explicit validation of ontology gained by text mining.

1. Documents are Always Inconsistent

Usually, some kind of requirements document is written in the beginning of a software project. After requirements elicitation, one of the first tasks of the software developer is to understand the requirements document which includes trying to understand the terminology used. But practical experiences show that apart from being imprecise, requirements documents also use inconsistent terminology.

A simple steam boiler specification [1], written for a formal methods contest, for example, looked extremely precise at first glance. However, the document called the same measuring unit in different places “water level measurement device”, “water level measuring unit”, “device to measure the quantity of water”, ... Obviously, this unwanted obfuscation hampers understanding of the domain. The reader can not be sure whether there is just one unit or two or three different devices. And of course, real life specifications, not written for an academic formal methods contest, are very likely even less consistent. Furthermore, real life documents are usually much longer rendering manual detection and resolution of such inconsistencies virtually impossible.

[1]Correspondence to: Leonid Kof, Fakultaet fuer Informatik, Technische Universitaet Muenchen, Boltzmannstr. 3, D-85748, Garching bei Muenchen, Germany Tel.: +49 89 289-17834; Fax: +49 89 289-17307; E-mail: kof@in.tum.de.

The consequences of this confusion grow with the progress of a software system through the software life cycle. At the later maintenance stage not only the documentation is inconsistent, but also the terminology used in the code does not necessarily coincide with the documentation. Sneed concludes in [2] that in many systems "*procedures and data are named arbitrarily*". Clearly, this strongly contributes to the fact that software maintenance consumes 80% of all costs for software and 50% out of these 80% must be devoted to program comprehension [3].

The goal of the approach presented in this paper is to detect and eliminate terminology inconsistencies by building consistent ontologies. The ontology extracted from documents and code are themselves validated either via prototyping (in requirements engineering) or by comparison with the implemented domain model (in the case of re-engineering). After validation by the stakeholders, this ontology will then be used as a consistent conceptual basis in the development or maintenance process.

Outline

The reminder of this paper is organized as follows: Section 2 gives an overview of existing text analysis methods. Section 3 shows how ontologies can be extracted in general, whereas Section 4 introduces an ontology extraction approach based on text analysis. In 5 we will show how ontology extraction can be embedded in the process of requirements engineering (or software re-engineering) and how the extracted ontology becomes validated, though validation is performed differently for requirements engineering and re-engineering. Finally, Section 7 summarizes the results of the proposed approach.

2. Related Work on Text Analysis

Document analysis for itself is neither a completely new problem nor a new solution in software engineering. There have already been various attempts to apply text analysis to requirements documents. In [4] Ben Achour classifies the linguistic methods of requirements engineering as either syntactic, lexical, or semantic.

Before giving an overview of the related approaches, we want to pose a set of criteria making a text analysis approach suitable for requirements documents analysis:

- The approach should not rely on any firm expression patterns. This is necessary due to extremely poor quality of requirements documents and practical impossibility to enforce any writing style.
- The approach should be interactive and not completely automatic. This is necessary to detect inconsistencies in the analyzed document. As praxis shows, inconsistencies are inevitable in requirements documents, which makes a completely automatic approach unfeasible. As Aussenac-Gilles [5] and Goldin & Berry [6] state, completely automated technique is not desirable as it potentially results in wrong extraction or information loss. Obviously, the human interaction should be limited to validation activities and should be unnecessary, for example, for pure term extraction.
- The approach should not rely on any previous domain knowledge. It is mostly the case in requirements engineering, that software at project beginning engineers have little superficial knowledge about the application domain, which causes difficulties in understanding the customer.

- The approach should extract not only terms relevant for the application domain, but also relations between these terms (i.e., ontology extraction instead of glossary extraction).

Among the three classes of existing approaches (lexical, syntactical, and semantical), lexical approaches are the most robust ones. Lexical methods, as proposed by Goldin & Berry [6], extract terms on the basis of common character sequences occurring in different sentences: any character sentence appearing in at list two sentences is a potential domain term. The decision whether the character sequence is really a term is made manually by the analyst. This simplicity is also the reason for the robustness of the lexical techniques. However, lexical approaches are limited to term extraction, they do not provide any term classification.

Syntactic methods are the oldest and the best known ones: Abbott suggested in [7] a method of terminology extraction based on an analysis of substantives, verbs, etc. (substantives become classes, verbs become actions, etc.) A similar proposal was made by Chen in [8]. Abbott states,

> Although the process we follow in formalizing the strategy may appear mechanical, it is not (given the current state–of–the–art of computer science) an automatable procedure.

The techniques proposed by Abbott and Chen could certainly be automated, today, using modern "Part–of–Speech" taggers (see for example Ratnaparkhi [9]). However, even if they were automated, these techniques would not produce a complete ontology but only the bare terminology.

A number of syntactic approaches were proposed for ontology extraction was proposed for other (not software engineering) applications. For example, Hearst [10] suggests a heuristics for extraction of the "is-a" relation from text. Berland and Charniak [11] extend the idea of Hearst to the extraction of the "part-of" relation. Degeratu and Hatzivassiloglou [12] use ideas similar to Hearst to extract the 'terms and relations from legal documents. The three above approaches, although sensible in the domains they were developed for, have common drawbacks making them barely applicable to requirements engineering:

- they rely on certain firm expression patterns,
- they are completely automated and do not give the user a validation possibility.

Other existing syntactical techniques, like those by Lame [13] and Zhou et al. [14] do not rely on firm expression patterns, but they require a-priori knowledge of domain terminology to become applicable.

Semantic techniques like those by Fuchs [15], Gervasi and Nuseibeh [16] or Ambriola and Gervasi [17] translate every sentence into a logical formula. This is surely even more than ontology extraction, but they rely on firm predefined expression patterns, which makes these approaches barely applicable to real life requirements documents. Furthermore, they *use* a given ontology to work, but they do not produce one.

The summary of this overview on related work on text analysis is obvious: there is no ontology extraction approach *satisfying the above requirements* in software engineering, yet! One could object that text mining was not necessary for ontology extraction. But, the only alternative would be purely manual ontology design (as in [18]) which would imme-

diately lead into the troubles that we aim to overcome. Manual design does not reliably detect terminology inconsistencies but rather is a major source for inconsistencies.

The remainder of this paper describes our proposed ontology extraction approach and its embedding into software engineering process. In the context of the presented work an ontology is defined as a taxonomy enriched with associations. The taxonomy itself consists of a set of terms and the "is-a"–relation.

3. Ontology Construction Basics

The concept of an "ontology" was introduced in artificial intelligence as a means for communication between intelligent agents (see for example [19]). Today, it is regarded as a generally useful concept to communicate concept dependencies. As software development involves communication between "intelligent agents", here called software engineers and domain experts, an ontology can be a valuable instrument to establish a common language within a software project.

3.1. Do We Actually Need an Ontology?

At first it might seem that an ontology is unnecessary because a common language could also be established with a simple glossary. However, a brief example by Zave and Jackson [20] shows that a simple glossary quickly falls short in establishing a common understanding. The context of this example is a hypothetical university information system with a definition of the term "student", the binary relation "enrolled", and a conversation between the two agents Able and Baker:

> Able: Two important basic types are *student* and *course*. There is also a binary relation *enrolled*. If the basic types and relations are formalized as predicates, then it holds that
>
> $$\forall s\ \forall c\ (enrolled(s, c) \Rightarrow student(s) \wedge course(c)).$$
>
> Baker: Do only students enroll in courses? I don't think that's true.
> Able: But that's what I mean by *student*!

Although they *do* agree that the term "student" is an important domain concept, they disagree on the meaning of this concept.

3.2. Basics of Ontology Construction During Requirements Engineering

The usefulness of an ontology as a requirements engineering product has already been recognized. For example, Breitman and Sampaio do Prado Leite [18] regard an application ontology as one of the products of the requirements engineering activity. All methodologies for ontology construction listed in their work share the same basic steps as shown in figure 1. Apart from validation and verification, these steps include identification of information sources, identification of the list of terms, classification of the terms and their description.

Besides these common steps, the various methodologies listed by Breitman & Sampaio do Prado Leite remain rather abstract in the sense that they do not specify *how* to identify information sources, *how* to classify terms, and so on.

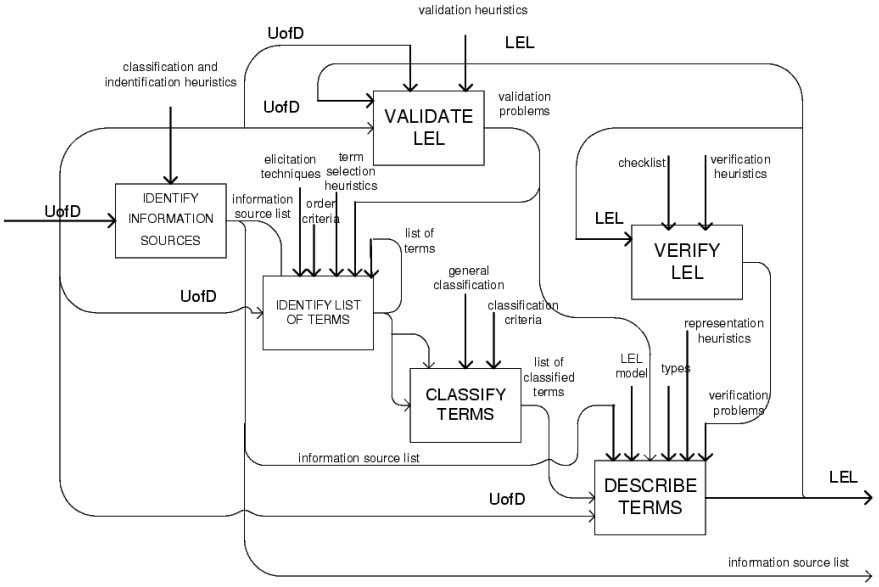

Figure 1. Ontology Construction Process (source: [18])

However, if we consider ontology construction as an activity within the requirements engineering process, the identification of the sources of information seems rather obvious: the primary source of information are the requirements documents. For our second example for employing ontologies in software engineering, i. e. re-engineering, two sources of information have to be regarded: documentation of the software system and its code itself.

The next section shows how we extract a domain ontology from textual documentation.

4. Ontology Extraction via Text Analysis

Text analysis can easily be introduced into the general ontology construction process presented above. Figure 2 depicts the activities needed to produce an ontology starting from some documents[1].

"Parsing and subcategorization frames extraction" corresponds to the "identification of the term list step" of figure 1, "term clustering & taxonomy building" as well as "association mining" correspond to "term classification". Term validation and verification will be addressed later (see sec. 5).

Note, that we define ontology as a taxonomy enriched with associations. The taxonomy itself consists of a set of terms and the "is-a"–relation. The overall process of ontology construction consists of four steps: term extraction, term clustering, taxonomy building (as cluster hierarchy) and relation (aka association) mining.

[1] see [21] for further details

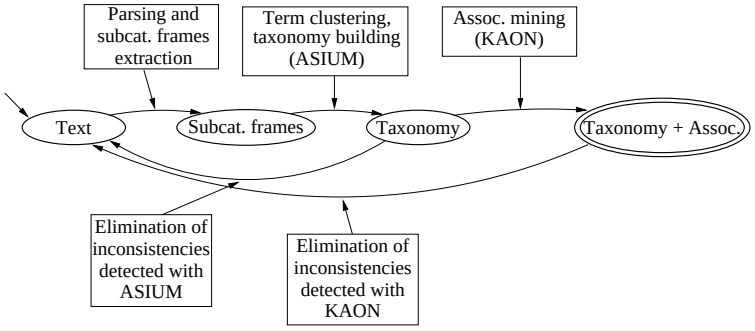

Figure 2. Ontology Extraction Procedure [21]

Extraction of terms from requirements documents: To extract terms, each sentence is parsed and the resulting parse tree is decomposed. Noun phrases that are related to the verb of the sentence are extracted as domain concepts. For example, from the sentence "`The control unit sends an alarm message in a critical situation`" "`send`" is extracted as the main verb, "`control unit`" as the subject and "`alarm message`" as the direct object.

Term clustering: The second step clusters related concepts. Two concepts are considered as related and put into the same cluster if they occur in the same grammatical context. I.e., two terms are related in the following cases:

- They are subjects of the same verb.
- They are direct objects of the same verb.
- They are indirect objects of the same verb and are used with the same preposition.

For example, if the document contains two sentences like

1. "The control unit sends an alarm message in a critical situation"
2. "The measurement unit sends measurements results every 5 seconds",

the concepts "`control unit`" and "`measurement unit`" are considered as related, as well as "`alarm message`" and "`measurements results`".

Taxonomy building: Concept clusters constructed in the previous step are used to build the taxonomy by joining overlapping concept clusters. The emerging larger clusters represent more general concepts. For example, the two clusters {`alarm message`, `measurements results`} and {`control message`, `measurements results`} are joined into the larger cluster

{`alarm message, control message, measurements results`}

because they share the common concept {`measurement results`}. The new joint cluster represents the more general concept of possible messages.
This step also aids in identifying synonyms[2] because synonyms are often contained in the same cluster. For example, if a cluster contains both "`signal`" and

[2] different names for the same concept

"`message`", the domain analyst performing the ontology construction can identify them as synonyms.
Since manual construction of the taxonomy would be both cumbersome and error-prone we use the tool ASIUM [22] for term clustering and taxonomy building.

Associations/relations mining: There is a potential association between two concepts if they occur in the same sentence. Each potential association then has to be validated by the requirements engineer before being recorded as an association between concepts.
Note, that the validation of the association proposed by the association mining tool automatically implies a validation of the requirements document. If the tool suggests an association that *can not* be valid (i.e., a pair containing completely unrelated concepts), then we have detected an evidence that the requirements document contains some inconsistent noise that must be eliminated (see [23] for an in–depth treatment of association mining).
We use the tool KAON [23] during this step.

All techniques introduced but term extraction were developed separately from each other for other purposes than requirements documents analysis. However, chaining these separate techniques into a proper process and enriching them with term extraction results is of great benefit for ontology extraction from requirements documents. The feasibility of the approach described above was proven on two case studies, presented in [21].

5. Applying Ontology to Document Validation

The method introduced above extracts an ontology from natural language documents. It is applicable to different kind of texts, such as requirements documents in new projects, or user and developer documentation of existing software.

5.1. Dealing with Inconsistencies

One key feature of the introduced text analysis method is its interactivity. In each step the analyst receives feedback allowing him to steer the construction. During the construction of concept clusters or overlaps the analyst may identify the content of the resulting clusters as inconsistent. This indicates that some unrelated concepts were put into the same cluster. Since such defects are not introduced by the extraction and clustering steps themselves, the inconsistencies detected can and should be corrected in the original text before the analysis continues, as illustrated in figure 2[3].

5.2. Terminology Validation After Extraction

The idea of iterative ontology extraction perfectly fits the Volere requirements engineering (RE) process [24]. According to [24] virtually every RE process is built in a similar way and runs through knowledge acquisition, writing the requirements document, requirements validation and prototyping. Figure 3 depicts a simplified Volere RE process. Solid arrows stem from the original process, dashed arrows are the transitions that we add in our approach to improve the validation phase.

[3] see [21] for details on inconsistency detection and correction

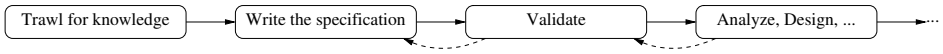

Figure 3. Volere Requirements Engineering Process

According to the Volere RE–Process, the RE process starts out with a brainstorming and refinement of the requirements until the requirements are ripe enough to be written down. Obviously, the results of the brainstorming sessions must be validated after writing them down. Robertson & Robertson [24] list several validation goals, such as

- Completeness of each requirement
- Traceability
- *Consistent terminology*

The ontology extraction described in the previous section facilitates checking for consistent terminology because inconsistent terminology becomes exposed in the text mining phase as implausible concept clusters and associations.

As stated above, the standard Volere–Process had to be extended to accommodate this kind of document validation: When inconsistent terminology is detected during text mining, it has to be corrected before the construction of the ontology continues. This feedback loop is marked by a dashed arc from “Validate” to “Write the specification” in figure 3. As a result of this iteration the requirements document will only contain consistent terminology by the moment ontology construction is finished. One could argue that this iteration could go on for a long time which is correct from an abstract point of view. In practice, applying this process has the pleasant side-effect to quickly teach writers of requirements documents how to use consistent terminology.

Obviously, the same idea of document analysis and validation can be applied in the case of software re-engineering. The only difference is that not the requirements documents but any existing documentation can be analyzed.

5.3. Validating the Taxonomy and Associations

Obviously, the extracted ontology itself must also be validated. In contrast to the validation of the terminology, this validation step depends on the goal (requirements engineering vs. re-engineering) of the analysis.

In the case of requirements engineering there is no reference domain model for the extracted domain ontology to be compared with. Thus, the ontology can only be validated via manual examination by a domain expert or via prototyping. Prototyping and validation in the case of requirements will be discussed in section 6.1.

In the case of re-engineering or software maintenance there are more sources of information than just the RE documentation. Another domain model can be extracted from the existing code. In virtually any realistic situation, the implemented and the documented domain model will differ. Section 6.2 shows how the domain model extracted from code can be compared with the ontology deduced from additional documentation and how the results of this comparison can be used to improve both documentation and code.

5.4. Ontology as a Document Validation Means: a Case Study

In one of our case studies we analyzed the steam boiler specification [1] to extract an ontology form it. This specification describes a control application whose aim is to support

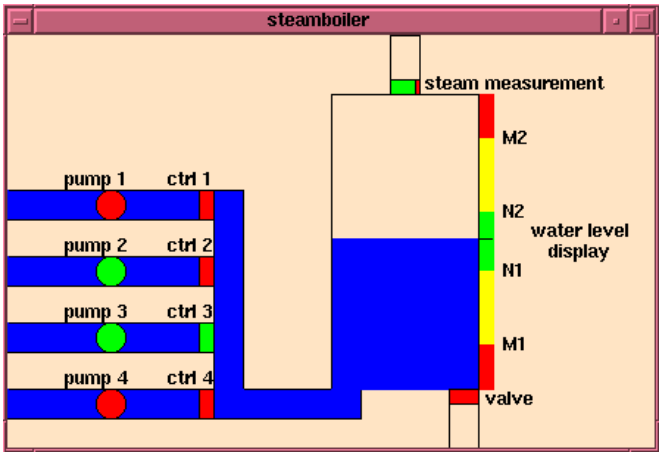

Figure 4. The steam boiler system [25]

required water level in a steam boiler. The steam boiler system consists of the following units (see also Figure 4, taken from [1]):

- the steam-boiler
- a device to measure the quantity of water in the steam-boiler ("water level display" in Figure 4)
- four pumps to provide the steam-boiler with water
- four devices to supervise the pumps (one controller for each pump) ("ctrl" in Figure 4)
- a device to measure the quantity of steam which comes out of the steam-boiler ("steam measurement" in Figure 4)
- an operator desk (missing in Figure 4)
- a message transmission system (missing in Figure 4)

The system should provide the required water level even despite failures of some components. Depending on the functioning components the system works in different operation modes.

Figure 5 shows a manually constructed ontology for the steam boiler system.[4] It shows the concepts introduced in the specification and a classification of these concepts. The classification is explicitly introduced in the requirements document as well. The concept classes are:

- sent messages (messages sent by the control program)
- received messages (messages received by the control program)
- failures
- operation modes
- physical units
- physical parameters

[4] Rectangular boxes represent concepts and hexagonal boxes represent relations between concepts. The very common "is-a"–relation is shown by means of variable-width lines, the thin end pointing to the more general concept and the thick end pointing to the less general one.

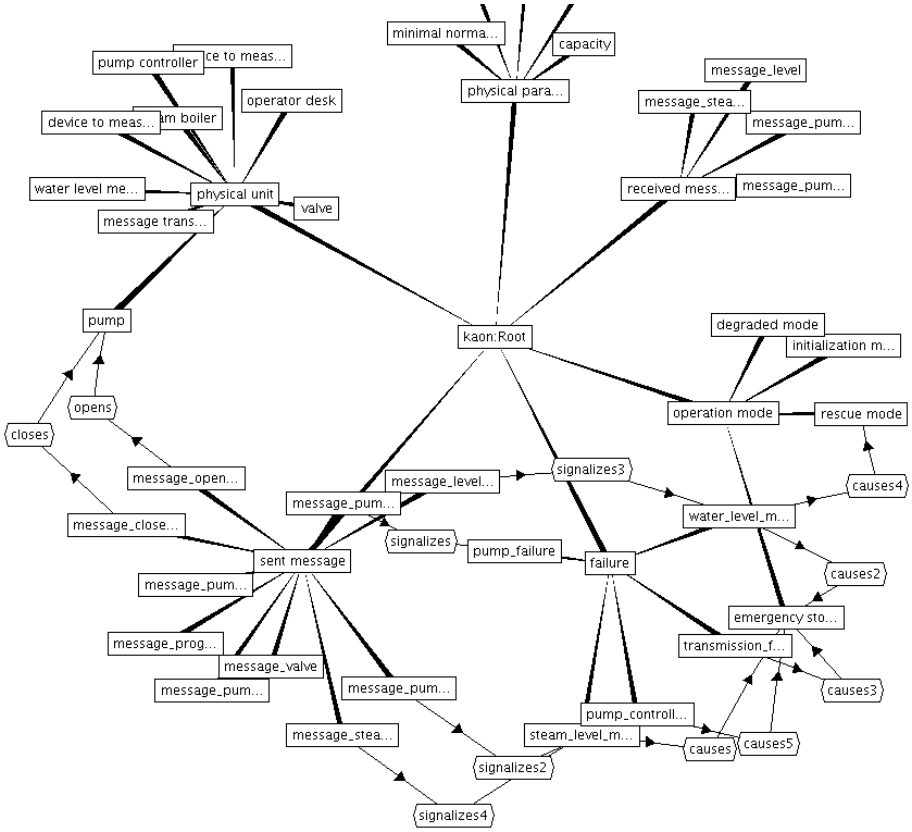

Figure 5. Steam boiler ontology, manually constructed

Only few associations are explicitly stated in the requirements document. Figure 5 shows three classes of them:

“signalizes” is an association between a hardware failure and a message signalizing this failure.

“causes” is an association between a hardware failure and the operation mode caused by this failure.

“opens/closes” are associations between messages controlling the pumps and the pumps themselves.

Figure 6 shows a part of the ontology produced by the means of text analysis. The diagram shows the ontology root (`kaon:Root`), four top-level concepts (`operation mode`, `failure`, `physical unit` and `message`) with some of their subordinate concepts and relations between them. For example, there are associations “Transmission_failure causes emergency_stop_mode” and “Rescue_mode is_caused_by water_level_measuring_unit_failure”.

When compared to the manually constructed ontology in Figure 5, the extracted ontology contains all the concept classes but “physical parameters”. The names of physical

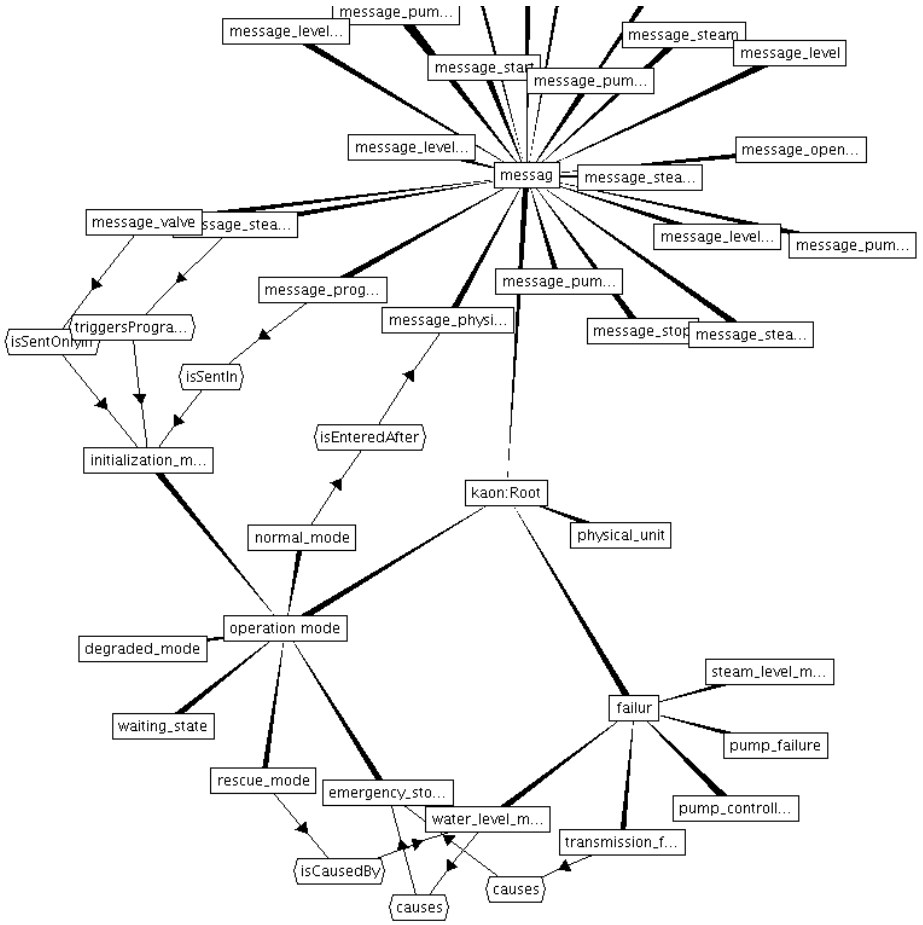

Figure 6. Steam Boiler: part of the produced ontology

parameters were not extracted as they occur solely in incomplete phrases (enumerations). Extraction of concepts from incomplete phrases is not possible yet. For the same reason two of the physical units were not extracted: “operator desk” and “message transmission system”. These concepts are mentioned only once in the document and their role is not further specified. A human reader would extract these two concepts, but would have to guess how they interact with other components. This point can be seen both as a weakness of the extraction technique and as an omission in the document: these two components are also missing in the steam boiler simulator (Figure 4), programmed for the participants of the formal methods contest, whose goal was to provide a formal steam boiler specification. As for other concept classes (messages, operation modes and failures), the approach succeeded in extracting all the concepts belonging to these classes.

Additionally to the concepts present in the original requirements document, operation mode “waiting state” was extracted from the revised document version. This concept was added during document revision, as the original document contained some abstract “state”, which was treated exactly in the same way as operation modes. The extracted

ontology contains also additional relations, like “message_program_ready isSentIn initialization_mode” and “message_steam_boiler_waiting triggersProgramStart_in initialization_mode”. These relations are sensible, but missing in the manually constructed ontology.

To summarize, the presented approach to ontology extraction by the means of text analysis is a powerful method, able to find flaws in requirements documents, to correct them, and then to produce an application domain ontology.

6. Use Cases for Ontology Validation

6.1. Requirements Engineering: Ontology Validation via Prototyping

Is ontology extraction sufficient for document validation? Not really, because the ontology itself could be flawed. It therefore has to be validated by a domain expert, first. For this purpose, we convert the ontology into a domain specific model. “Domain specific” means that the modelling technique is tailored to the needs of the application domain.

In one of our case studies [21] we used the formal method and tool AutoFOCUS [26] to build the model domain for a distributed embedded control systems. AutoFOCUS offers the following modelling concepts for distributed embedded systems specification:

- hierarchically structured components
- messages
- typed channels (for message exchange)
- automatons (as a component implementation) including states and state transitions

The mapping of the ontology extracted from documents onto an AutoFOCUS domain model consists of two steps:[5]

1. Complete subtrees of the extracted taxonomy are mapped on AutoFOCUS concepts. For example, one of the taxonomies extracted in the case study contained the subtree “hardware”. Each leaf concept of this subtree was mapped to an AutoFOCUS “Component”. The taxonomy also contained the subtree “failures”, which was mapped to AutoFOCUS “States”, etc.
   If two branches are non disjoint, one either has to map both of them onto the same modelling concept or to descend to finer subtrees. The decision on which concept to use for what subtree is context dependent and has to be made by the analyst.
2. The associations existing in the ontology are mapped onto the corresponding connection concepts available in the formal model. These are communication channels for components and transitions between states for subtrees that were mapped onto states.

The result of this modelling is a formal model of the system specified by the requirements document. Note, that due to the automated terminology extraction and the mechanical construction of the ontology, this formal model resembles the system specified by the requirements documents without being biased from misinterpretations of the software engineer or forgotten resp. ignored statements in the documents.

[5] mapping to AutoFOCUS is presented in more detail in [27]

In case of AutoFOCUS, this model is even executable representing a prototype of the target system. This prototype in turn is an excellent means to validate the ontology itself because it allows a domain expert to check whether all the intended states, components, transitions, etc. are in place and behave as expected. A failure of this validation represents a flaw in the ontology which is most likely due to an inconsistency or incompleteness of the original requirements document!

For example, consider Figures 7 and 8. The former one represents the components and channels generated from the ontology shown in Figure 6, while the latter represents the generated states and state transitions for the “control_unit”. The state transition diagram is obviously incomplete. This is due to the fact that the relations between corresponding states are missing in the ontology. The associations, in turn, are missing just because the state are never explicitly mentioned in the same sentence. Thus, missing state transitions in the resulting AutoFOCUS model indicate lack of explicitness in the requirements document.

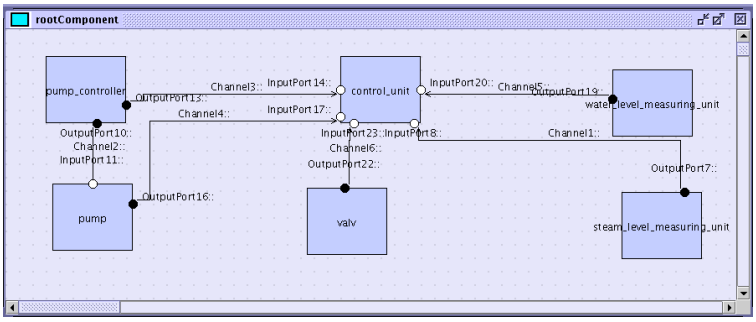

Figure 7. Component network, converted from the steam boiler ontology in Figure 6

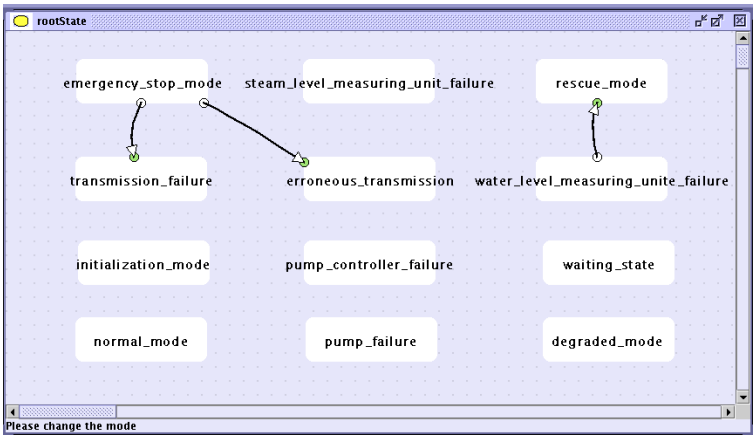

Figure 8. The automaton for “control_unit”, converted from the steam boiler ontology in Figure 6

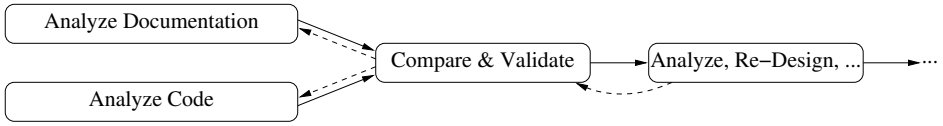

Figure 9. Knowledge acquisition process, adapted to several information sources

6.2. Re-Engineering: Ontology Comparison

Similar to requirements engineering, software re-engineering can usually only be accomplished after a sound domain knowledge has been gained. However, in opposite to requirements engineering there is more than one relevant source of information. In addition to the documentation, the program code is another, an even more important source of domain information. As the code and the documentation are frequently changed independently after roll-out, the documented and the implemented domain model are hardly ever consistent but differ from each other. Thus, to validate the ontology extracted from the documentation, it is necessary to compare it to the ontology implemented in program code or vice versa.

The availability of more than one source of information entails minor changes to the process of knowledge acquisition. Figure 9 shows ontology extraction from two different sources. Now, validation is done by comparing different ontologies and not via prototyping.

To compare the documented and the implemented domain model, it is necessary to extract a domain model from the program code. In the case of object-oriented (OO) code, a significant portion of the concept hierarchy is explicitly coded with the class hierarchy induced by inheritance and associations. In non-OO code other structural dependencies, such as modules, file and directory structure, and `#include` directives must be taken into concern. For simplicity, we assume an OO programming model, here.

In [28], we have shown, how concise and consistent naming within the program code can be achieved and preserved during software development and maintenance. Here, require concise naming of identifiers within the code and continue this line of thought by using the names of the identifier as the starting point for term extraction. The associations between the concepts – i.e. classes – are extracted from the inheritance hierarchy and associations among classes.

Note, that the ontology build from code will comprise domain concepts as well as technical concepts, as for example classes used for file or database access. To be able to compare this ontology with an ontology build from domain only documentation, the domain concepts must be separated from the technical concepts. Ontology comparison metrics, such as introduced by Maedche and Staab [29], can cater for such “add-ons” when measuring ontology similarities. The defined metrics are asymmetric, so it is possible to see whether the documented ontology is covered by the implemented ontology.

The following list defines ontology similarity measures adapted to the comparison of ontologies extracted from documentation and program code.

Lexical comparison: Lexical comparison is based on the edit distance [30] measuring the minimal number of insertions, deletions and substitutions necessary to convert one string into another. For example, the edit distance between “toy example” and

"toy-examples" is 2. On the basis of the edit distance Maedche and Staab define string similarity of the two strings L_i and L_j:

$$StrSim = \max\left(0, \frac{\min(|L_i|, |L_j|) - EditDistance(L_i, L_j)}{\min(|L_i|, |L_j|)}\right)$$

The string similarity measure $StrSim$ returns values between 0 and 1; 0 meaning completely different, 1 for match. Lexical ontology similarity of two ontologies built on lexicons $\mathcal{L}_1$ and $\mathcal{L}_2$ is defined as

$$LexSim(\mathcal{L}_1, \mathcal{L}_2) = \frac{1}{|\mathcal{L}_1|} \sum_{L_i \in \mathcal{L}_1} \max_{L_j \in \mathcal{L}_2} (StrSim(L_i, L_j))$$

The lexical similarity measure $LexSim$ is asymmetric, which is necessary for comparison of the documented and the implemented ontology:
$LexSim(DocumentedLexicon, ImplementedLexicon)$ measures whether every documented concept is implemented. Surely, one should expect that the implemented ontology contains more concepts. However, it is suspicious, if the documented concepts are missing in the implementation.

Taxonomy comparison: Taxonomy similarity measures whether the sub- and superconcepts of a certain concept coincide in two ontologies. Let $\mathcal{H}$ denote the taxonomic (hierarchical) part of the ontology and let $SupSub(L, \mathcal{H})$ be the set of sub- and superconcepts of L in the hierarchy $\mathcal{H}$. Taxonomic overlap of two hierarchies with respect to the term L is defined as

$$TO(L, \mathcal{H}_1, \mathcal{H}_2) = \frac{|SupSub(L, \mathcal{H}_1) \cap SupSub(L, \mathcal{H}_2)|}{|SupSub(L, \mathcal{H}_1) \cup SupSub(L, \mathcal{H}_2)|}.$$

The average value measures the extent to that the documented and the implemented hierarchies agree.

$$\overline{TO}(\mathcal{H}_1, \mathcal{H}_2) = \frac{1}{|\mathcal{L}_1|} \sum_{L \in \mathcal{L}_1} TO(L, \mathcal{H}_1, \mathcal{H}_2)$$

This metric is asymmetric, just like the lexical similarity metric.
$\overline{TO}(DocumentedHierarchy, ImplementedHierarchy)$ measures whether the documented concept hierarchy (extracted as cluster hierarchy) is correctly implemented.

Relation comparison: The relation similarity metric defined by Maedche and Staab [29] is rather complicated though this complexity is dispensable for ontologies extracted from code. It caters for relations with arbitrary domain and range but the relationships at the class level of OO code are restricted: Each relation connects just two classes. Note, that the relations extracted from the text are binary as well (see section 4 or [21] for details). Thus, to compare relations, a simpler metric can be used.
Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be two lexicons and let the lexicons be sorted in such a way that for all i the concept L_{1i} corresponds to the concept L_{2i}. This correspondence can be established for example by the means of the lexical similarity measure. Given the sets of non-taxonomic relations $\mathcal{R}_1$ and $\mathcal{R}_2$ as parts of the extracted ontologies,

it is easy to determine the coinciding relations, i. e. the relations connecting the corresponding terms. With this the relational overlap of two ontologies can be computed as follows:

$$\overline{RO}(\mathcal{R}_1, \mathcal{R}_2) = \frac{Number\ of\ coinciding\ relations}{|\mathcal{R}_1|}$$

Again, this metric is asymmetric.
$\overline{RO}(DocumentedRelations, ImplementedRelations)$ exposes whether all the documented relations are implemented.

These ontology similarity metrics deliver valuable information to software maintainers by giving an estimate on how large the discrepancies between documentation and code are. In other words, these numbers reflect the reliability of the information contained in the documentation.

In the case that the similarity measures are high enough (it is up to the analyst to define what "high enough" means) they also validate the extracted domain ontology.

7. Summary

The ontology based validation method presented in this paper tackles a crucial tasks in various software engineering activities: validation of the terminology. Two example scenarios that could benefit from this approach are requirements engineering and software maintenance.

The starting point of this ontology construction is an analysis of the available documents. However, as experience shows, the documents, especially the requirements documents, are usually inconsistent. Thus, detection of inconsistencies has predominant importance if the ontology is to provide any value.

The iterative text analysis approach presented in this paper performs both tasks: it extracts the terminology and gives feedback to the analyst, enabling him to discover inconsistencies. After elimination of these inconsistencies the domain ontology is built and can be used as *the* common language for all the stakeholders in further project phases.

In order that become actually useful, the ontology gets validated. In the case of requirements engineering the validation is performed via prototyping, in the case of reengineering the ontology extracted from documents is validated by comparing it with another ontology extracted from the actual program code.

Term extraction, ontology construction and validation all together are combined into an integrated process that delivers a valuable tool to counter inconsistencies in documentation and code which are a frequent source of misunderstandings during software development and subsequent errors in the resulting products.

References

[1] Abrial, J.R., Börger, E., Langmaack, H.: The steam boiler case study: Competition of formal program specification and development methods. In Abrial, J.R., Borger, E., Langmaack, H., eds.: Formal Methods for Industrial Applications. Volume 1165 of LNCS., Springer–Verlag (1996) `http://www.informatik.uni-kiel.de/~procos/dag9523/dag9523.html`.

[2] Sneed, H.M.: Object-oriented cobol recycling. In: WCRE '96, IEEE Computer Society (1996) 169
[3] Pigoski, T.M.: Practical Software Maintenance. Wiley Computer Publishing (1996)
[4] Ben Achour, C.: Linguistic instruments for the integration of scenarios in requirement engineering. In Cohen, P.R., Wahlster, W., eds.: Proceedings of the Third International Workshop on Requirements Engineering: Foundation for Software Quality (REFSQ'97), Barcelona, Catalonia (1997)
[5] Aussenac-Gilles, N.: Supervised Learning for Ontology and Terminology Engineering. In Kushmerick, N., Ciravegna, F., Doan, A., Knoblock, C., eds.: machine Learning for the Semantic Web, Dagstuhl seminar, Dagstuhl (Germany). (2005)
[6] Goldin, L., Berry, D.M.: AbstFinder, a prototype natural language text abstraction finder for use in requirements elicitation. Automated Software Eng. **4** (1997) 375–412
[7] Abbott, R.J.: Program design by informal English descriptions. Communications of the ACM **26** (1983) 882–894
[8] Chen, P.: English sentence structure and entity-relationship diagram. Information Sciences **1** (1983) 127–149
[9] Ratnaparkhi, A.: Maximum Entropy Models for Natural Language Ambiguity Resolution. PhD thesis, Institute for Research in Cognitive Science, University of Pennsylvania (1998)
[10] Hearst, M.A.: Automatic acquisition of hyponyms from large text corpora. Technical Report S2K-92-09 (1992)
[11] Berland, M., Charniak, E.: Finding parts in very large corpora. In: Proceedings of the 37th annual meeting of the Association for Computational Linguistics on Computational Linguistics, Morristown, NJ, USA, Association for Computational Linguistics (1999) 57–64
[12] Degeratu, M., Hatzivassiloglou, V.: Building automatically a business registration ontology. In: The Second National Conference on Digital Government (dg.o 2002), LA, CA. (2002)
[13] Lame, G.: Using nlp techniques to identify legal ontology components: Concepts and relations. In: Law and the Semantic Web. (2003) 169–184
[14] Zhou, L., Booker, Q., Zhang, D.: Toward rapid ontology development for underdeveloped domains. In: HICSS '02: Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS'02)-Volume 4, Washington, DC, USA, IEEE Computer Society (2002) 106
[15] Fuchs, N.E., Schwertel, U., Schwitter, R.: Attempto Controlled English (ACE) language manual, version 3.0. Technical Report 99.03, Department of Computer Science, University of Zurich (1999) `http://www.ifi.unizh.ch/attempto/publications/papers/ace3_manual.pdf`, accessed 21.05.2004.
[16] Gervasi, V., Nuseibeh, B.: Lightweight validation of natural language requirements: a case study. In: 4th International Conference on Requirements engineering, IEEE Computer Society Press (2000) 140–148
[17] Ambriola, V., Gervasi, V.: Experiences with domain-based parsing of natural language requirements. In Fliedl, G., Mayr, H.C., eds.: Proc. of the 4th International Conference on Applications of Natural Language to Information Systems. Number 129 in OCG Schriftenreihe (Lecture Notes) (1999) 145–148
[18] Breitman, K.K., Sampaio do Prado Leite, J.C.: Ontology as a requirements engineering product. In: Proceedings of the 11th IEEE International Requirements Engineering Conference, IEEE Computer Society Press (2003) 309–319
[19] Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach. 2nd edition edn. Prentice-Hall, Englewood Cliffs, NJ (2003)
[20] Zave, P., Jackson, M.: Four dark corners of requirements engineering. ACM Trans. Softw. Eng. Methodol. **6** (1997) 1–30
[21] Kof, L.: An Application of Natural Language Processing to Domain Modelling – Two Case Studies. International Journal on Computer Systems Science Engineering **20** (2005) 37–52
[22] Faure, D., Nédellec, C.: ASIUM: Learning subcategorization frames and restrictions of se-

lection. In Kodratoff, Y., ed.: 10th European Conference on Machine Learning (ECML 98) – Workshop on Text Mining, Chemnitz Germany (1998)
[23] Maedche, A., Staab, S.: Discovering conceptual relations from text. In W.Horn, ed.: ECAI 2000. Proceedings of the 14th European Conference on Artificial Intelligence, Berlin, IOS Press, Amsterdam (2000) 321–325
[24] Robertson, S., Robertson, J.: Mastering the Requirements Process. Addison–Wesley (1999)
[25] Abrial, J.R., Börger, E., Langmaack, H.: Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control. Volume 1165 of LNCS. Springer–Verlag (1996)
[26] The AutoFocus Homepage: (2004) `http://autofocus.in.tum.de/index-e.html`, accessed 21.02.2004.
[27] Klitni, A.: Textanalyse für Requirements Engineering: Konvertierung der Analyseergebnisse nach AutoFOCUS (2004) Technische Universität München, Fakultät für Informatik, Systementwicklungsprojekt.
[28] Deißenböck, F., Pizka, M.: Concise and consistent naming. In: Proceedings of the 13th International Workshop on Program Comprehension, St. Louis, Missouri, USA, IEEE CS Press (2005)
[29] Maedche, A., Staab, S.: Measuring similarity between ontologies. In: EKAW '02: Proceedings of the 13th International Conference on Knowledge Engineering and Knowledge Management. Ontologies and the Semantic Web, Springer-Verlag (2002) 251–263
[30] Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions, and reversals. Cybernetics and Control Theory **10** (1966) 707–710