A Data Structure for the Refactoring of Multimodal Knowledge

Jochen Reutelshoefer, Joachim Baumeister, Frank Puppe

Institute for Computer Science, University of Würzburg, Germany
{reutelshoefer, baumeister, puppe}@informatik.uni-wuerzburg.de

Abstract. Knowledge often appears in different shapes and formalisms, thus available as multimodal knowledge. This heterogeneity denotes a challenge for the people involved in today's knowledge engineering tasks. In this paper, we discuss an approach for refactoring of multimodal knowledge on the basis of a generic tree-based data structure. We explain how this data structure is created from documents (i.e., the most general mode of knowledge), and how different refactorings can be performed considering different levels of formality.

1 Introduction

In today's knowledge engineering tasks knowledge at the beginning of a project is often already available in various forms and formalisms distributed over multiple sources, for instance plain text, tables, flow-charts, bullet lists, or rules. We define this intermixture of different shapes of knowledge at different degrees of formalization as *multimodal knowledge*. However, current state-of-the-art tools are often constrained to a specific knowledge representation and acquisition interface for developing the knowledge base. In consequence, the tools are commonly not sufficiently flexible to deal with multimodal knowledge. While the evolution of the knowledge system based on a single formalism (e.g, ontology evolution) has been thoroughly studied, the evolution of multimodal knowledge has not yet been sufficiently considered. In this paper, we propose a data structure and methods, that support representation and refactoring of multimodal knowledge. We have implemented this data structure within a semantic wiki, since such systems proved to be well-suited to support collaborative knowledge engineering at different levels of formalization.

The rest of the paper is organized as follows: In the next section, we give a detailed introduction into multimodal knowledge, and we motivate why refactoring of this type of knowledge is necessary. Further, we provide a data structure, that helps to perform refactorings. In Section 3, we introduce categories of refactorings for multimodal knowledge and we show how they can be performed using the described approach. In this context, we discuss how semi-automated methods can help on the refactoring tasks. In Section 4, we discuss the overlapping aspects of refactoring with the related domains ontology engineering and software engineering. We conclude pointing out the challenges we face with this approach and give an overview of the planned future work.

2 Multimodal Knowledge

We introduce the idea of multimodal knowledge (MMK) and its advantages and challenges. Further, we present a data structure called *KDOM* to represent multimodal knowledge in an appropriate manner. An implementation of this approach within the semantic wiki KnowWE is also given.

2.1 Multimodal Knowledge and the Knowledge Formalization Continuum

Often, knowledge is available in different (digital) representations as we already motivated above. To gain advantage of the knowledge by automated reasoning, the collection of differently shaped knowledge pieces needs to be refactored towards an initial (formalized) knowledge base. During this process the entire knowledge base may contain a wide range of different degrees of formalization and distinct representations. The full range from unstructured plain text over tables or flowcharts to executable rules, or logics sketched as in Figure 1 is metaphorically called the *knowledge formalization continuum* (KFC) [1].

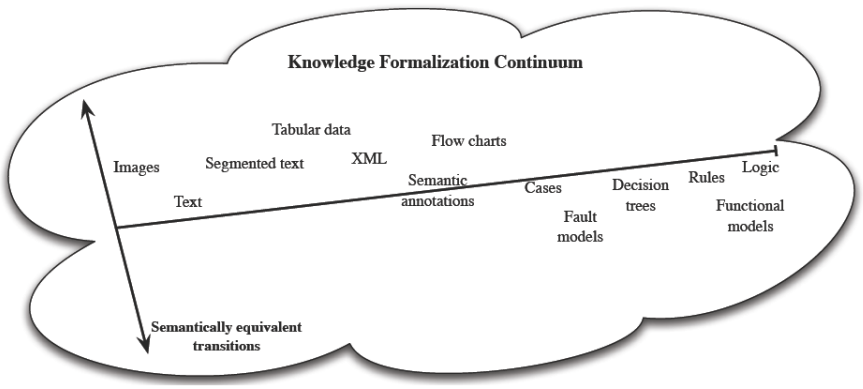


Fig. 1. Sketch of the Knowledge Formalization Continuum building the basis for multimodal knowledge

By turning knowledge into other representations, it allows for (not completely) seamless transitions in either direction - more formal or less formal. The most suitable degrees of formalization for a given project need to be carefully selected according to the cost-benefit principle. However, in any case refactorings towards the desired target knowledge base become necessary. *Refactoring* is defined as changing the structure of something without changing the semantics. For clarification, we comprehend refactoring in this context as changing structure

without changing the *intended* semantics as we are also dealing with non-explicit knowledge artefacts (e.g., plain text). This point of view also considers plain texts as first class knowledge items, which can (manually or semi-automated) be refactored to an executable representation.

Advantages of Working with Multimodal Knowledge:
User friendliness: The formats and representations the domain experts are used to (e.g., plain text in some cases) can be integrated in the knowledge engineering process. Thus, people can participate in the first step with a minimum of training efforts. Lowering the barriers of participation tackles an important problem of knowledge engineering in general.

Bootstrapping: Assuming that we can work with different representations of knowledge, the bootstrapping process shows up being extremely simple: Any documents relevant to the problem domain can just be imported into the system. The evolution of the knowledge driven by the knowledge engineering process will increase its value continuously.

Maintenance: Many (executable) knowledge bases that have been created in the past lack of maintainability. For example, in the compiled versions of large rule bases there is often no sufficient documentation attached to allow other people to further extend or modify the knowledge. Using the MMK approach — to keep the executable knowledge artefacts closely located next to original justifying less formal knowledge entities — provides knowledge engineers a context-sensitive comprehension of the formalized knowledge.

The Challenge of Working with Multimodal Knowledge:
The main challenge is to cope with the different forms of knowledge with respect to formality and syntactical shape. In the next section, we present an approach enabling the multimodal knowledge to be refactored (at the cost of some pre-engineering). However, in detail there are further challenges to be considered:

- handle redundancy of knowledge in different representations and degrees of formality
- tracing the original source of knowledge items (justification) while traversing the KFC
- keeping readability/understandability for humans (the flow of the content)

2.2 KDOM – a Data Structure for Multimodal Knowledge

The most important aspect of this approach is that free text is accepted as a fundamental representation of knowledge. The key idea is to have a self-documenting knowledge base, that can easily be understood by the domain experts. Further, explicitly formalized parts can be embedded into the free text paragraphs. Our approach, to cope with the problems of different knowledge formats described above, implies to break down the given data to (some) textual representation. Some non-textual structure like tables or flowcharts can be converted into text

(for example using cell delimiter signs or XML-formats). However, images, for instance, cannot be converted into a useful (in this context) textual representation. Thus, these items are considered as *knowledge atoms*. This approach treats such items as units, which can be refactored (merely moved) as a whole, but its internal structure cannot be changed. To apply refactoring methods, we build a detailed document tree from the given document corpus. We call this tree the *Knowledge Document Object Model* (KDOM) inspired by the DOM-tree of HTML/XML-documents. The difference is that the source data is not in XML syntax and that we have an explicit underlying semantics for (at least parts of) the content. Of course, one system cannot support every imaginable format of knowledge. Thus, some pre-engineering efforts are necessary to provide support for the formats required. These include the formats given in the startup knowledge and the target formats forming the 'goal' of the engineering task. In the pre-engineering step we define a kind of schema-tree merely forming the ontology of the syntactical structure of the content that is processed.

The KDOM Schema Tree We describe the possible compositions of syntactical entities in a multimodal knowledge document as a tree. At each level the expected occurring formats are listed as siblings. We call such a definition of a syntactical entities together with some intended semantics a *KDOM-type*. An example KDOM schema tree for documents containing text, tables, comments, and rules is shown in Figure 2[1].

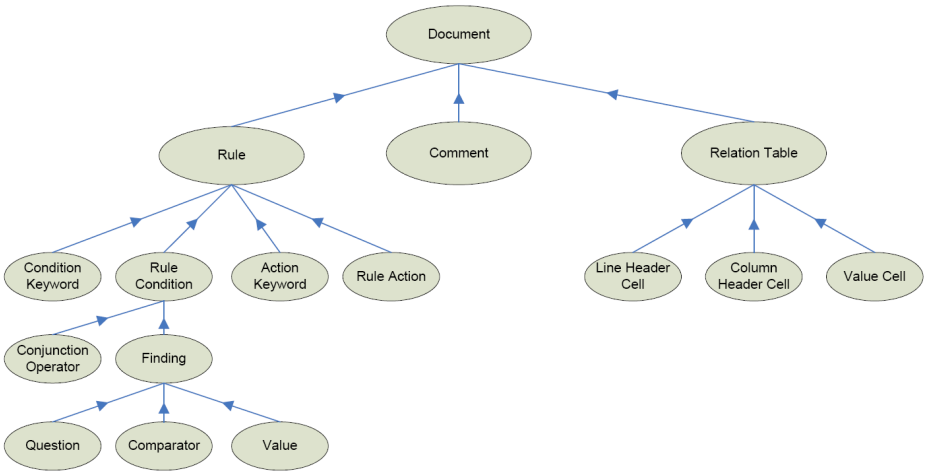


Fig. 2. Sketch of a KDOM schema tree for tables, rules, and comments.

[1] A KDOM schema is similar to an XML-Schema definition except that we do not have XML-Syntax, but an explicit definition of the syntactical shape (a parser) for each type.

This tree schema specifies the syntactical patterns, that can occur as sub-patterns of its parents. The type *Document* — which is always the root node in this KDOM schema — has three children types: *Rule*, *Relation Table*, and *Comment*. Thus, in the document rules, relation tables and comments are expected. Each of these first level types may specify children types for themselves denoting which sub-entities are expected. It does not specify any cardinalities or orders of appearances in the document.[2] In the next paragraph we outline how this KDOM schema is used to create a content tree from knowledge documents applying a *semi-parsing*-like approach. Semi-parsing denotes, that only specific parts of a document are processed in detail by parsers, while other parts remain as text nodes containing a (potentially large) fragment of plain text.

Building a Content KDOM Tree Having an appropriate schema tree of types including their parsers defined, one can start to create content trees from the documents. The following gives a short definition of the tree-based data structure:

Definition 1 (KDOM). *A* KDOM *is defined as set of nodes, where a node* $n \in KDOM$ *is defined as a tuple*

$$n = (id,\ content,\ type,\ parent,\ children).$$

Thus, each node stores a unique id, some textual content, a type (describing the role of the content), one parent (with exception of the root node having no parent), and an (ordered) list of children nodes. A valid KDOM of a document is given if:

1. *The text content of the root node equals the text content of the document.*
2. *The following constraints are true:*
   (a) *text-concatenation(n.children) = n.text for all* $n \in \{KDOM \setminus LEAFS\}$ *LEAFS being the subset of KDOM with an empty chilren set*
   (b) *n.type accepts n.text for all* $n \in \{KDOM\}$*, i.e., the text part of the node n can be mapped to the corresponding type.*

At each level in the schema tree the implicit type *PlainText* is allowed, catching arbitrary content, that is not covered by explicitly defined types (semi-parsing). This definition implies, that a concatenation of the leafs in depth-first search order results in the full document text string. We also provide types for concepts, concept values, conditions over concepts, and rules; further types can be easily added. The construction of a KDOM is sketched by pseudo code in Listing 1.1.

The root node of a document always refers to the full document — this is also the first step of the tree building algorithm. Then, in each level all children types are iterated and searched in the father's content. When one type detects

[2] The order of the siblings defines the order the entities are processed in the parsing algorithm (see Listing 1.1)

a text passage that is relevant (*findOccurences* i.e., matched by its parser), then it allocates this text fragment. Once some text fragment is allocated by a type it will only be processed by the children types of the former type (defined by the KDOM schema tree). If there is lots of unstructured text in MMK we expect that lots of text does not match any type and thus is not allocated by an (explicit) type in the tree (*createPlaintextNodes*).

Listing 1.1. A recursive algorithm to build up a KDOM tree

```
buildKDOMTree (fatherNode) :

  forall (type : childrenTypes(fatherNode))
    childrenNodes = findOccurrences(type, unallocatedTextOfFather)

    forall (childNode : childrenNodes)
      buildKDOMTree(childNode)


  forall (string : unallocatedTexts)
    createPlaintextNodes(string)
```

Figure 3 shows an example of a document that is parsed by the KDOM schema introduced in Figure 2. It shows a wiki system describing possible problems with cars. The particular article provides information on clogged air filters in form of plain text paragraphs, rules, and a table.

The first paragraph shows some describing text, followed by a comment line. Then, a rule (labeled number 3) is defined followed by plain text and so on. Rule and tables are labeled in detail hierarchically, e.g., (3.1) and (3.2) for the two parts of the rule. Given that the parser-components of the types of the schema tree are configured correctly to recognize the relevant text parts, we can use the proposed algorithm to create the KDOM content tree from the document. The resulting parse tree shown in Figure 4 has one node for each labeled section from the document.

As required already mentioned above any level in the tree contains the whole content of the document. The content can be considered/engineered at different levels of formality. Thus, also the refactoring methods in general can be applied at different levels.

2.3 Implementation in KnowWE

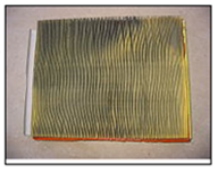
Semantic wikis have been successfully used in many different software engineering and knowledge engineering projects in the last years, e.g., KIWI@SUN [2]. Further, a semantic wiki is a suitable tool to capture multimodal knowledge as described. For this reason we implemented the introduced KDOM data structure in the semantic wiki KnowWE [3]. In fact, the KDOM tree is the main data structure carrying the data of the wiki pages. A unique ID is assigned to

General# (1)

The (combustion) air filter prevents abrasive particulate matter from entering the engine's cylinders, where it would cause mechanical wear and oil contamination.

Most fuel injected vehicles use a pleated paper filter element in the form of a flat panel. This filter is usually placed inside a plastic box connected to the throttle body with an intake tube.

Older vehicles that use carburetors or throttle body fuel injection typically use a cylindrical air filter, usually a few inches high and between 6 and 16 inches in diameter. This is positioned above the carburetor or throttle body, usually in a metal or plastic container which may incorporate ducting to provide cool and/or warm inlet air, and secured with a metal or plastic lid.



clogged air filter

Typical Symptoms#

Typical symptoms, which indicate a clogged air filter, are the following driving issues: unsteady idle speed, weak acceleration, starting pr (2) reased mileage (based on average mileage and the currently measured mileage or abnormal exhaust fumes.

Comment: *This rule should be moved somewhere else*

(3.1) (3.2.1) (3.2.2) (3.2.3) (3.2)

```
IF (Exhaust fumes = black AND Fuel = unleaded gasoline)
THEN Clogged air filter = Suggested
```

(3)

(3.3) (3.4)

A typical starting problem which is connected to this problem is a barely or not starting engine in combination with a starter that turns over.

A clogged air filter can cause black exhaust fumes which will turn the color of the exhaust pipe to sooty black.

Relevant symptoms:

- engine start
- exhaust fume color
- driving
- real mileage

(4)

(5) (5.1) (5.2) (5.3)

	Clogged Air Filter
Driving = un peed	+
Driving = weak acceleration	+
Starter = turns over	(...)
Exhaust pipe color evaluation = abnormal	+
Exhaust fumes = black	+

Comment: *The table above needs some verification* (6)

(7.1) (7.2)

```
IF Driving = weak acceleration
THEN Clogged air filter = Suggested
```

(7)

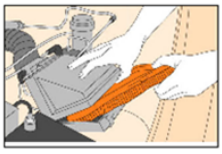
(7.3) (7.4)

Repair Instructions# (8)

A clogged air filter needs to be replaced by a new one. Therefor the air filter housing have to be found. It will be either square (on fuel-injected engines) or round (on older carbureted engines) and about 12 inches (30 cm) in diameter.

After locating the housing the screws or clambs on the top of it have to be removed. Now the old air filter can be removed. At this point the housing should be cleaned from any dirt and debris with a clean rag.

Finally the new air filter can be put in and the top of the housing can be screwed or clambed back on.



aif filter change

Fig. 3. An example document containing tables, rules, and comments.

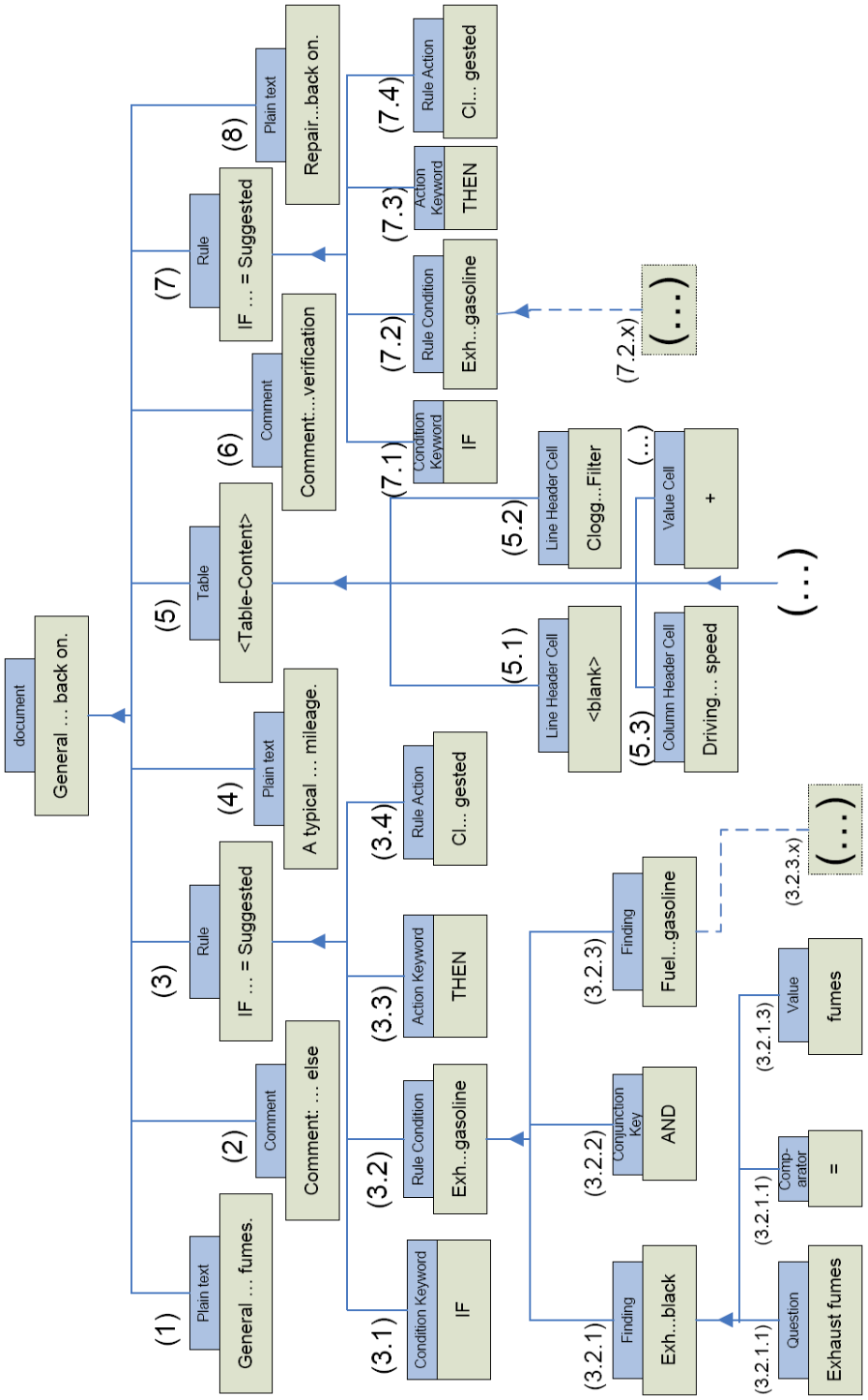


Fig. 4. Structure of the KDOM content tree for the given example document.

every content node of the tree, which allows precise referencing of specific parts of a document/wiki page. The types are integrated into the system by a plugin mechanism. For additional (groups of) types a plugin is added to the core system at system initialization time.

Figure 5 shows a class diagram with the core classes participating in the implementation of the KDOM approach in the system KnowWE.

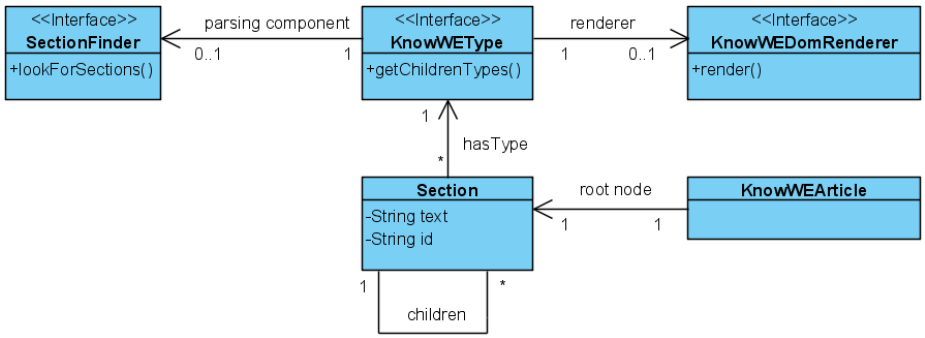


Fig. 5. A simple class diagram of the KDOM implementation in KnowWE.

For each *KnowWEType* a *SectionFinder* as parser component is specified, which is responsible to allocate the correct text areas for the given type. To generate the content tree the algorithm shown in Listing 1.1 is implemented. Of course, a big part of the pre-engineering workload is implementing parsers (SectionFinder) for defined types. For this reason, we provide a library of parser components for common formats (e.g., XML, tables, line/paragraph-based), that can be reused and extended. This allows for quick adaptation to new projects demanding specific knowledge formats. Some of the markups implemented in KnowWE can be found in [3].

3 Evolution of Multimodal Knowledge with Refactorings

The evolution of knowledge in wikis is typically performed by manual editing of the source texts of the wiki pages by the user community. Although, many systems already provide some editing assistance techniques (e.g., templates, auto-completion, tagging), the work of restructuring the knowledge already present is still accomplished by manual string manipulations in the wiki source editor.

Given the KDOM tree described in Section 2.2 the structure of the knowledge can be taken into account to develop further refactoring support. The text-based knowledge can be considered in context when examining the content *and* types of the surrounding nodes (father, siblings, cousins).

3.1 Refactorings

In the following we describe refactorings and how they can be performed with this approach:

1. Renaming of concept
2. Coarsen the value range of a concept

Rename Concept This is probably the operation used most frequently, and it is also simple to perform. The task is to identify each occurrence of the object in all documents and replace it by the new name specified. Precondition of course is, that the occurrences were captured correctly in the KDOM tree generated. Problems can arise when different objects have the same name. For example if different questions have equally named values. Overlapping value terms often occur for example on scaled feature values like *low*, *average/normal*, and *high*. Regarding the following two sketched rules the system needs to distinguish between *normal* as value for *Exhaust pipe color* and for *Mileage evaluation*, when performing a renaming task on the value ranges.

```
IF Exhaust pipe color = normal
THEN Clogged Air Filter = N3

IF Mileage evaluation = normal
THEN Clogged Air Filter = N2
```

As the text string *normal* will probably appear quite frequently in an average document base, it is necessary to identify the occurrences, when it is used as value of a specific concept. The renaming algorithm can solve these ambiguities by looking at the KDOM tree. Figure 6 shows the relevant KDOM subtrees of the two rules. Thus, the renaming algorithm can be configured to check whether a parent node of a value (*1*) is of type *Finding*(*2*). Further, it can look up the content the sibling node of type *Question* (*3*) to infer the context of the value for any occurrence.

Renaming of the occurrences in the formal parts in a consistent way is necessary for compiling executable knowledge. However, the occurrences in less formal parts cannot be identified that easily. But considering the whole knowledge corpus renaming of these occurrences is still desirable with respect to consistency of the multimodal knowledge base. We can provide all occurrences as propositions to the knowledge engineer as a simplest semi-automated approach. To improve this approach, we are planning to employ advanced NLP techniques on less detailed/formalized parts of the KDOM content trees to generate better propositions.

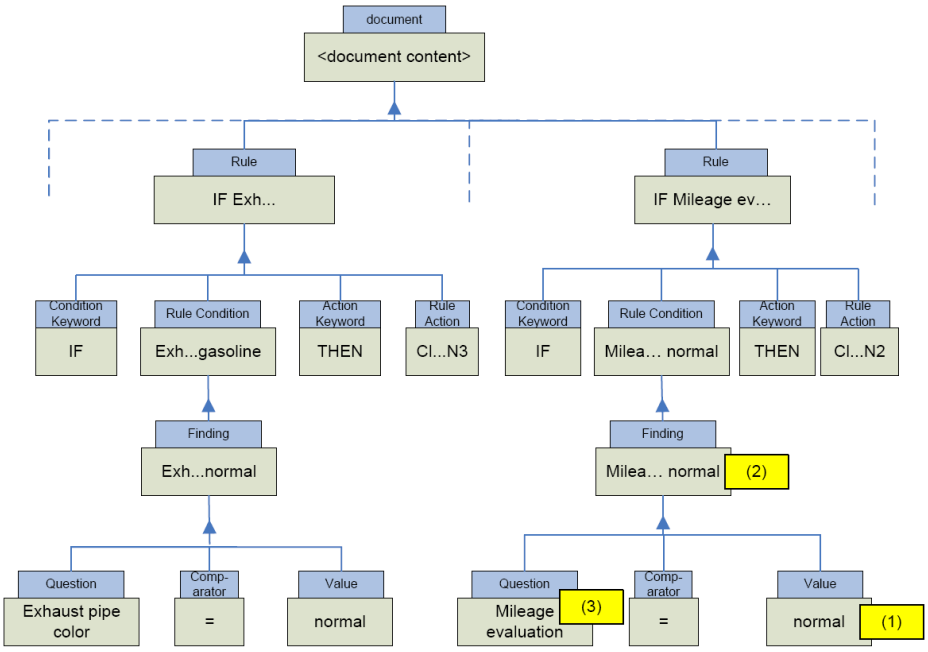


Fig. 6. KDOM subtrees for findings of the two rules listed above.

Coarsen Value Range Often, domain experts start implementing the ontological knowledge with choice questions providing detailed value ranges. During ongoing development the value range of some concepts turn out to be unnecessary precise, e.g., an over-detailed concept. In the car diagnosis scenario, the value range of *Mileage evaluation* is initially defined with the values given in the following:

```
Mileage evaluation
- decreased
- normal
- slightly increased
- strongly increased
```

During the development of the knowledge base it turns, that it is not suitable to have a distinction between *slightly increased* and *strongly increased mileage*. A reason could be, that the knowledge is not so detailed to take advantage of this distinction, resulting in an unnecessary high number of rules or disjunctive expressions. The solution is a mapping of *slightly increased* and *strongly increased* to a new value *increased*. To execute this it is necessary to find and adapt all knowledge items using the object. This operation can be performed as an iterated application of *Rename Concept* on the value range of the concept.

4 Related Work

The presented work is strongly related to refactoring in ontology engineering and techniques for refactoring in software engineering.

Refactoring in Ontology Engineering The benefit of refactoring methods has been recognized in ontology engineering, recently. Current research, however, only discuss the modifications of concepts and properties within an ontology, but does not consider possible implications of tacit knowledge that is neighbouring and supporting the ontology. For example, in [4] various deficiencies were presented that motivate the application of targeted refactoring methods. Here, the particular refactoring methods also considered the appropriate modifications of linked rule bases. In [5] an approach for refactoring ontology concepts is proposed with the aim to improve ontology matching results. The presented refactorings are mainly based on rename operations and slight modifications of the taxonomic structure. In the past, the approach *KA scripts* was presented by Gil and Tallis [6]. KA scripts and refactoring methods are both designed to support the knowledge engineer with (sometimes complex) modifications of the knowledge. More recently, a related script-based approach for enriching ontologies was proposed by Iannone et al. [7].

Refactoring in Software Engineering The presented parsing algorithm can also be compared to the parsing of formal languages (e.g., programming languages), which has been employed successfully for multiple decades. There, the parsers are often generated from a (context-free) grammar specification (e.g., ANTLR [8]) and can process the input in linear time [9]. The parse trees also are used for refactoring in development environments. However, in order to deal with multimodal knowledge one advantage of the KDOM approach is that it can also deal with non-formal languages (to some extend), for example, by employing text-mining or information extraction techniques to generate nodes. Additionally, this idea will be extended to a semi-automated workflow involving the knowledge engineer. Further, we can add new syntax as plugins, and we are able to configure the schema at runtime. Even though, we are working on the integration of parse trees generated by classical parsers to allow embedding of formal languages into the semi-parsing process at better performance.

5 Conclusion

In this paper, we introduced the generic data structure KDOM to support the representation of multimodal knowledge. We explained how given documents can be parsed into this data structure with some initial pre-engineering effort. Further, we explained how it serves as the basis for refactoring of the knowledge. We proposed a selection of refactorings and sketched how they can be performed automated or semi-automated using the KDOM. We further plan to apply semi-automated processes on the construction of the KDOM tree by

proposing detected objects or relations to the knowledge engineer, who then can confirm if a given type should be attached to some text fragment.

One of the key challenges in this approach is, that the system needs to be newly configured to each knowledge engineering task, its startup document structures and its target representations. This entails that the knowledge engineering tools needs to be agile and methods and tools for the quick definition of parser components are necessary.

References

1. Baumeister, J., Reutelshoefer, J., Puppe, F.: Engineering on the knowledge formalization continuum. In: SemWiki'09: Proceedings of 4th Semantic Wiki workshop. (2009)
2. Schaffert, S., Eder, J., Grünwald, S., Kurz, T., Radulescu, M.: Kiwi – a platform for semantic social software (demonstration). In: ESWC'09: Proceedings of the 6th European Semantic Web Conference, The Semantic Web: Research and Applications, Heraklion, Greece (June 2009) 888–892
3. Reutelshoefer, J., Lemmerich, F., Haupt, F., Baumeister, J.: An extensible semantic wiki architecture. In: SemWiki'09: Fourth Workshop on Semantic Wikis – The Semantic Wiki Web (CEUR proceedings 464). (2009)
4. Baumeister, J., Seipel, D.: Verification and refactoring of ontologies with rules. In: EKAW'06: Proceedings of the 15th International Conference on Knowledge Engineering and Knowledge Management, Berlin, Springer (2006) 82–95
5. Svab, O., Svatek, V., Meilicke, C., Stuckenschmidt, H.: Testing the impact of pattern-based ontology refactoring on ontology matching results. In: Third International Workshop On Ontology Matching (OM2008). (October 2008)
6. Gil, Y., Tallis, M.: A script-based approach to modifying knowledge bases. In: AAAI/IAAI'97: Proceedings of the 14th National Conference on Artificial Intelligence and 9th Innovative Applications of Artificial Intelligence Conference, AAAI Press (1997) 377–383
7. Iannone, L., Rector, A., Stevens, R.: Embedding knowledge patterns into OWL. In: ESWC'09: Proceedings of the 6th European Semantic Web Conference, The Semantic Web: Research and Applications. Springer (2009) 218–232
8. Parr, T.: The Definitive ANTLR Reference: Building Domain-Specific Languages. Pragmatic Bookshelf (2007)
9. Wilhelm, R., Maurer, D.: Compiler Design. International Computer Science Series. Addison-Wesley (1995) Second Printing.