Refactoring Methods for Knowledge Bases

Joachim Baumeister, Frank Puppe, and Dietmar Seipel

Department of Computer Science
University of Würzburg, 97074 Würzburg, Germany
Phone: +49 931 888-6740, Fax: +49 931 888-6732
email: {baumeister, puppe, seipel}@informatik.uni-wuerzburg.de

Abstract. The manual development of large knowledge systems is a difficult and error-prone task. In order to facilitate extensions to an existing knowledge base the structural design of the implemented knowledge needs to be improved from time to time. However, experts are often deterred even from important design improvements since some restructurings are too complex to handle.
In this paper, we introduce a framework that allows for automated refactorings. Refactoring methods are well-defined and are executed in a semi-automated way. In this manner, the developer is supported during the process of restructuring of even large knowledge bases. Refactoring methods are usually applied to improve the design of the knowledge base; in this paper, we sketch some design anomalies that identify poor design of the knowledge base.

1 Introduction

Although knowledge systems have been established in many domains over the last years, the development and maintenance of such systems is still a costly and time-consuming task. Large knowledge systems are commonly build in a manual way, i.e., domain specialists are modeling the knowledge bases by hand and formalize the knowledge using specialized tools. Also the maintenance and extension of the knowledge bases is usually done by the specialists themselves.

In the last years we have gained experience in projects with building large diagnostic knowledge systems, mainly in the medical domain. For example, the HEPATOCONSULT system [4] is a publicly available consultation and documentation system for the diagnosis of liver diseases. A subsystem of HEPATOCONSULT is the SONOCONSULT system [10], which is in routine use in the DRK-hospital in Berlin/Köpenick, Germany. The practical installation of further systems, such as the ECHODOC system [13] (formerly QUALITEE), in a hospital environment is currently implemented.

An important aspect of knowledge system development is the continuous adaptation and extension of already deployed knowledge bases. Often, these modifications consider the improvement of the knowledge base design, i.e., the simplification and generalization of certain aspects of the available knowledge. However, changing existing and often large knowledge bases is not a simple problem, since unstructured modifications can cause unexpected deficiencies and errors.

Engineering Knowledge in the Age of the Semantic Web:
14th International Conference, EKAW 2004, LNAI 3257, pp. 157-171,
Springer Verlag , 2004

1.1 Refactoring, Restructuring and Refinement

We present refactoring methods for the save modification of diagnostic knowledge bases. Such restructurings are initiated by the domain specialist in order to improve the design of the knowledge base. The main focus of the presented approach is the improvement of the knowledge base design. Since even simple changes of knowledge objects can imply subsequent and complex adaptations of the remaining knowledge base the domain specialist should be supported by specialized tools.

The introduction of such restructurings for a step-wise and algorithmic modification of the knowledge base was inspired by *refactoring methods* introduced for software engineering [5,14]. Here, refactoring considers the improvement of existing software code without changing the external behavior of the program. The implementation of refactorings is quite risky, especially for large systems. Therefore, refactoring methods are strongly connected with appropriate test methods. However, when performing a refactoring method the accompanying tests (e.g., unit tests) need to be adapted, and this is mostly done manually. In contrast to software engineering we can see that refactoring methods for knowledge bases often take advantage of the declarative nature of knowledge and test knowledge, respectively. Thus, refactoring methods for knowledge systems often can propagate their changes to the corresponding test knowledge, e.g., by modifying the respective objects in test cases.

The work on refactoring methods for knowledge bases is related to the concept of *KA scripts* of Gil and Tallis [7]. KA scripts and refactoring methods are both designed for supporting the knowledge engineer with complex changes of the knowledge. However, refactoring methods are focussing on improving the design of the knowledge base, whereas KA scripts mainly try to assist the user during extensions of knowledge bases. Thus, the application of refactoring methods is driven by detected design anomalies, whereas KA scripts try to provide methods for preserving a usable state of the knowledge after a manual modification of the knowledge base.

The refinement of the knowledge base performed by restructuring methods differs from *refinement techniques*, e.g., described by Boswell and Craw [3] or Knauf et al. [12]. Thus, refactoring is usually not applied for improving the accuracy of the system, but for improving the design of the knowledge base. Especially for this reason a refactoring method is currently applied manually, while supported by automated adaptations of attached knowledge. In ontological engineering [8] the problem of restructuring ontologies has been also recognized. For example, Góméz-Pérez and Rojas-Amaya [9] describe an approach for reengineering ontologies for reuse that consists of the three activities *reverse engineering*, *restructuring*, and *forward engineering*. Although no procedural methods were given for implementing the restructuring activity some typical concepts for refactoring the design of an ontology with a subsequent validation were discussed.

1.2 Basic Definitions

We consider the refactoring of diagnostic knowledge bases using instances of the domain ontology as given in the following.

Let Ω_Q be the universe set of all *questions* available in the application domain. The type of a question $q \in \Omega_Q$ depends on the domain $dom(q)$, i.e. the range of values for the answers to the question q. The value range can define

- numerical values for reals or integers,
- symbolic values with either one-choice or multiple-choice answers, and
- arbitrary content for text answers.

The assignment of a value $v \in dom(q)$ to a question $q \in \Omega_Q$ is called a *finding*, and we define $\Omega_\mathcal{F}$ to be the set of all possible findings in the given problem domain; a finding $f \in \Omega_\mathcal{F}$ is denoted by $q{:}v$. Each finding f is defined as a possible input of a diagnostic knowledge system. Questions concerning specific areas of the application domain are grouped into question sets. A *question set* Q contains a list of questions that are semantically related, i.e. $Q \subseteq \Omega_Q$.

A *diagnosis* is representing a possible output of a diagnostic knowledge system. We define $\Omega_\mathcal{D}$ to be the universe of all possible diagnoses for a given problem domain. With respect to a given problem a diagnosis $d \in \Omega_\mathcal{D}$ is assigned to a symbolic state $dom(d) = \{not\ probable, unclear, suggested, probable\}$.

A *case* is defined as a tuple $c = (\mathcal{F}_c, \mathcal{D}_c)$, where $\mathcal{F}_c \subset \Omega_\mathcal{F}$ is a set of findings given as input to the case. Often $\mathcal{F}_c$ is also called the set of *observed findings* for the given case. The set $\mathcal{D}_c \subseteq \Omega_\mathcal{D}$ contains the diagnoses describing the solution of the case c. The set of all possible cases for a given problem domain is denoted by Ω_C. We call a collection of cases $CB \subseteq \Omega_C$ a *case base*.

A *rule* $r = cond(r) \rightarrow action(r)$ consists of a rule condition $cond(r)$ containing disjunctions, conjunctions, and/or negations of arbitrary findings $F \in \Omega_\mathcal{F}$ or assigned diagnosis states, and a rule action $action(r)$, that is executed if the rule condition evaluates to true in a given case. The rule is called an *abstraction rule*, if the action assigns a value $v \in dom(q)$ to a question $q \in \Omega_Q$. Alternatively, an action $d{:}s$ of a *scoring rule* can assign a certainty score s to a given diagnosis $d \in \Omega_\mathcal{D}$. The dialog of the system, i.e., presenting questions and question sets to the user, is controlled by *indication rules*; such rules indicate a list of questions or a list of question sets in their rule action. We define $\Omega_\mathcal{R}$ to be the universe of all possible rules for a given universe of diagnoses $\Omega_\mathcal{D}$ and a given universe of questions Ω_Q; we call $\mathcal{R} \subseteq \Omega_\mathcal{R}$ a *rule base*.

2 A Framework for Refactoring Methods

When a knowledge system is build in an evolutionary way the design of the knowledge base needs to be improved by refactorings from time to time. The refactoring is motivated by the fact that the knowledge base is extended incrementally and consequently the knowledge design becomes messy. Furthermore, the use of the knowledge system in a real life environment can reveal overdetailed or unused knowledge objects, such as unneeded diagnoses or questions that are never answered. However, since the modification of (large) knowledge bases can be a difficult and error-prone task it is suggestive to propose structured approaches for such modifications.

It is worth mentioning, that refactoring methods differ from ordinary knowledge modifications: Refactoring methods provide an exact procedure describing the particular modification of the knowledge base, and they include all possible consequences

for connected knowledge elements. For example, with the modification of a knowledge object the required adaptations and possible conflicts for the implemented knowledge are described. With the adaptation of such knowledge also the adjustment of the accompanying test knowledge is discussed.

Furthermore, refactoring methods differ from ordinary modifications with respect to their tight connection with automated test methods. For an automated application of tests the expected result of a test needs to be known beforehand. A commonly used automated test method is *empirical testing*, i.e., running a collection of previously solved cases and comparing the stored solution with the result derived by the knowledge system. There exist further tests that could be executed without any additional test knowledge like cases, e.g., anomaly testing [15]. The general procedure for the application of refactoring methods is as follows:

1. The actual refactoring is motivated by the detection of a design anomaly, or by a complex extension of the knowledge base.
2. An appropriate refactoring method is selected for the task.
3. An automated test suite is used for inspecting the valid behavior of the knowledge base. A refactoring cannot be applied, if the tests uncover an invalid behavior.
4. The refactoring method is performed with respect to the mechanics given in the methods description.
5. After the successful application of the method the test suite is again used for validating the behavior of the knowledge base. The method can only applied successfully if no errors have occurred.

In the context of this paper we consider rules and cases as possible representations for diagnostic knowledge. It is worth noticing, that case-based knowledge is not only suitable for diagnostic reasoning, but can be also used as test knowledge, e.g. by the application of empirical testing. In the next section we sketch some situations that typically point to design anomalies in the knowledge base. Based on such anomalies the knowledge base can be refactored in order to improve the design.

3 Design Anomalies in Knowledge Bases

With the manual construction of knowledge systems the design of the knowledge base often becomes complex over the time. For example, the level of detail for some findings turns out to be too specialized or findings/diagnoses are contained in the knowledge base, but are actually not used. Since refactorings are applied in order to simplify knowledge design it is important to discuss some typical examples of bad knowledge design. Such *design anomalies* are related to the classical definition of anomalies, e.g. described for rule bases by Preece and Shinghal [15]. Here, the knowledge base is checked for containing redundant, ambivalent or circular rules. According to Preece and Shinghal anomalies are not errors but are symptoms for probable errors in a knowledge base. Analogously, Iglezakis et al. [11,18] introduced quality measures for the detection of design anomalies for case-based knowledge. In the context of this paper we focus on areas in the knowledge base that are mainly responsible for worsening the knowledge design. The understandability and maintainability of the system can be improved by modifying these areas, which we call *design anomalies*. In software engi-

neering research design anomalies are often identified as *bad smells* in software code as described by Fowler [5]; we have adapted the concept of bad smells to knowledge design, and synonymously use the terms bad smell and design anomaly.

Although no exact metrics can be given for identifying such smells we want to introduce some situations in which a bad smell is probable. We do not give an exhaustive list of design anomalies but sketch some typical settings that are *smelly*.

Lazy Knowledge Object. The use of the implemented knowledge objects like findings and diagnoses is an important issue for the design of knowledge systems. If the finding or diagnosis is never used or used very infrequently in the real life environment, then the deletion of the object should be considered in order to simplify the knowledge design; fewer objects included in the knowledge base improve its simplicity. In such a case the methods REMOVEDIAGNOSIS and REMOVEQUESTION are appropriate refactorings. However, such an anomaly should be considered very carefully: of course, detected diagnoses or questions that are used very infrequently but can be used functionally in the problem domain, usually should not be removed from the knowledge base. Nevertheless, the detection of a lazy object can identify abandoned entities that have no more a functional meaning in the present version of the system, and can be therefore deleted without reducing the derivational power of the system.

Overdetailed Question. For a choice question the range of possible values may be too detailed in the used application. For example, the developer of the system has defined the five values in

$dom(temperature) = \{very\ low, low, normal, high, very\ high\}$

for a question *temperature*. During the following development of the knowledge base it turns out that a less detailed value range with only three possibilities is more suitable, e.g., *low*, *normal*, and *high*. In such cases, the method COARSENVALUERANGE can be used. Alternatively, the level of detail can be reduced by converting a question with a numerical domain into a choice question with a discrete and ordered value range; then, the refactoring method TRANSFORMNUMINTOOC is appropriate.

Lengthy Dialog. If the end user has to enter the findings manually, then the dialog efficiency of the knowledge system is an important issue to consider. Lengthy dialogs with many unnecessary questions are often perceived to be annoying and can critically affect the practical success of the implemented system. If the dialog turns out to be too long and too verbose for the end users, then the method EXTRACTABSTRACTION can be used to replace a list of original questions by a single, semantically equivalent and abstracted question. Alternatively, the method EXTRACTQUESTIONSET can be used to divide a large question set into a list of smaller questionnaires. Further modifications can improve the dialog capabilities by omitting extracted question sets that are perceived to be irrelevant.

Finding Clump. The rules defining the diagnostic knowledge for particular diagnoses frequently contain the same collection of findings. The design of the rule base may be simplified, if this *clump of findings* is substituted by a single finding representing an abstraction of the jointly occurring findings, i.e., using the EXTRACTABSTRACTION

method. Then, complex rule conditions can be simplified by replacing the finding clump with the abstraction finding. However, the finding clump is very difficult to eliminate if the combined occurrence of the findings mean a disproportionate confirmation or disconfirmation of a diagnosis. It is very difficult, to map such reenforcing observations to a single abstraction, automatically.

Besides the bad smells mentioned above there exist various others mainly depending on the actual knowledge representation. For detecting some of the sketched design anomalies we can provide simple tests, e.g., using a sufficiently large case base with real life cases lazy objects can be easily identified by counting their occurrences in the cases. A question can be assumed to be overdetailed, if there exist similar rules, i.e., rules with equivalent rule action that only differ in assigned question values contained in the rule condition. In the following section we introduce the refactoring methods COARSENVALUERANGE and EXTRACTQUESTIONSET in detail and briefly describe other methods related to the described design anomalies.

4 A Catalog of Refactoring Methods

Refactoring methods are defined in a template-like from in order to allow for a convenient and simple application. Each refactoring method is described by the following seven elements.

Name	A short and meaningful name is chosen in order to simplify the identification of the particular methods. The names are used to build a vocabulary of refactoring methods.
Summary	A description of the method summarizing the functionality of the refactoring method.
Motivation	A collection of situations in which this refactoring method should be applied, e.g. referring to design anomalies.
Consequences	A report of experienced conflicts and restrictions, when applying this refactoring method. Additionally, hints are given as workarounds in the case of a conflict.
Mechanics	A description of the actual refactoring method in an algorithmic and step-wise style.
Example	A simple example is given depicting the application of the refactoring method.
Related methods	An enumeration of related (e.g. inverse) refactoring methods.

Using this framework we can define a catalog of refactoring methods describing the particular modifications of a knowledge base in more detail. The developer can decide about an appropriate refactoring by simply browsing the catalog and retrieving the possible conflicts given in the consequences section.

The consequences of each refactoring strongly depends on the applied knowledge representation. In the context of this paper we only consider rule-based and case-based knowledge as possible representations; in [1] we also discussed the consequences for

further types of knowledge, e.g. causal set-covering models. In the following, we provide (shortened) catalog entries of the methods COARSENVALUERANGE and EXTRACTQUESTIONSET.

COARSENVALUERANGE

Reduce the size of the value range of a choice question in order to scale down the granularity – interactive tool support can be provided.

Motivation

Often domain experts start implementing the ontological knowledge with choice questions providing detailed value ranges. During ongoing development the value range of some questions turns out to be unnecessarily precise, e.g. an overdetailed question. Furthermore, a smaller value range may simplify the dialog for the end-users, e.g. a lengthy dialog.

Consequences

Let $q \in \Omega_Q$ be the selected choice question with value range $dom(q)$. For the execution of the method the developer has to specify a transformation function $t : dom(q) \rightarrow dom'(q)$, which maps the values of the original value range to the values of the reduced value range.
Redundancies and conflicts can be caused due to the mapping to a smaller value range. The applied knowledge, i.e. the rule base $\mathcal{R}$ and case base CB, is investigated in oder to detect conflicts. The following conflicts can arise:

Rule-Based Knowledge

Creation of identical sub-conditions:
Due to the refactoring two rules with identical sub-conditions can be created.

Choice question, *or* condition
: For choice questions the rule condition can contain an *or* condition of two equivalent sub-conditions that were originally referring to different choice values and have been mapped to the same value. One of the two equivalent sub-conditions can be deleted automatically in order to remove this redundancy.

Multiple-choice question, *and* condition
: If the refactored question is a multiple-choice question, then the rule condition can contain an *and* condition of two equivalent sub-conditions targeting the same transformed choice value. This redundancy can be automatically removed by deleting one of the equivalent sub-conditions.

Creation of identical conditions:
The refactoring method modified two rules so that their conditions are equal. This can cause ambivalent and redundant rules.

Redundant rules

If all rules with identical rule condition contain an equal rule action, then all except one rule can be deleted automatically.

Ambivalent rules

The refactoring method can cause a conflict by creating ambivalent rules, i.e., rules with equal rule condition but different rule action.

Abstraction rules: A different value of the same question is derived by two rules with the same condition. The rules are presented to the user in order to resolve this conflict by manual adaptation.

Scoring rules: There exist two rules with identical rule condition, that derive a (different) state for a single diagnosis. By default, the rules are replaced with a newly created scoring rule with the same rule condition and a score aggregated from the original scores. However, the aggregated rule is presented to the user for a subsequent adaptation.

Case-Based Knowledge

Similarity knowledge needs to be adapted according to the transformation function. Refactored cases can cause redundant and ambivalent cases.

Identical cases

If two cases $c, c' \in CB$ have an identical set of findings and an equal set of diagnoses, i.e., $\mathcal{F}_c = \mathcal{F}_{c'}$ and $\mathcal{D}_c = \mathcal{D}_{c'}$, then by default the cases remain in the case base, and the developer has to decide manually about a possible deletion.

Ambivalent cases

If two cases $c, c' \in CB$ have an identical set of findings but a different set of diagnoses, i.e., $\mathcal{F}_c = \mathcal{F}_{c'}$ and $\mathcal{D}_c \neq \mathcal{D}_{c'}$, then by default the cases remain in the case base. However, for specialized case bases, e.g., defining test cases, ambivalence denotes a semantic contradiction, and therefore this conflict can require the method to be canceled.

Mechanics

The refactoring method is performed by the following procedure:

1. Apply the test suite to the knowledge system and abort, if errors are reported.
2. Select the choice question $q \in \Omega_Q$ for which the value range $dom(q)$ should be reduced; define a new value range $dom'(q)$ for q; $|dom'(q)| < |dom(q)|$.
3. Define a transformation function $t : dom(q) \rightarrow dom'(q)$, which maps the original values $v \in dom(q)$ to the new values $v' \in dom'(q)$. Usually, $dom'(q) \subset dom(q)$ and $t(v) = v$ for all $v \in dom'(q)$.
4. Adapt the available test knowledge with respect to the new value range $dom'(q)$, e.g., modify test cases containing findings $q{:}v$.
5. Modify the knowledge attached to question q according to the transformation function t. During the mapping of the values of q check for conflicts as described in the *Consequences* section.
6. Apply the test suite to the refactored knowledge system and cancel the refactoring method, if errors are reported; alternatively start a debug session.

Example

The one-choice question "*temperature*" (*temp*) with the value range

$$dom(temp) = \{very\ low, low, normal, high, very\ high\}$$

is too detailed and should be simplified by the value range

$$dom'(temp) = \{low, normal, high\}\,.$$

The developer defines a transformation function t given by the following table:

$dom(temp)$	$dom'(temp)$
$very\ low$	low
low	low
$normal$	$normal$
$high$	$high$
$very\ high$	$high$

Originally, the following rules connected with the question $temp$ are contained in the knowledge base (with diagnosis "$infection$" and question set QS):

$$r_1 : temp{:}high \vee temp{:}very\ high \rightarrow infection{:}s_{suggested}$$
$$r_2 : temp{:}high \rightarrow indicate(QS)$$
$$r_3 : temp{:}very\ high \rightarrow indicate(QS)$$

After the application of the refactoring method we obtain the following rules:

$$r'_1 : temp{:}high \vee temp{:}high \rightarrow infection{:}s_{suggested}$$
$$r'_2 : temp{:}high \rightarrow indicate(QS)$$
$$r'_3 : temp{:}high \rightarrow indicate(QS)$$

The rule r'_1 contains a redundant sub-condition and is further reduced. Since the rule r'_2 and r'_3 are equal we also remove rule r'_3. We obtain the following final rules:

$$r''_1 : temp{:}high \rightarrow infection{:}s_{suggested}$$
$$r''_2 : temp{:}high \rightarrow indicate(QS)$$

Related Methods

n/a

EXTRACTQUESTIONSET

An existing question set is divided into two question sets by extracting a collection of questions from the original question set into a newly created question set – automated tool support can be provided.

Motivation

The number of questions contained in one question set may accumulate during the continuous development of the ontological knowledge. In order to facilitate a more compact and meaningful representation of the available questions large question sets can be partitioned into smaller chunks containing semantically related questions. If the *lengthy dialog* smell is detected, then this method is often applied as a first step. In subsequent steps the rules indicating the extracted question sets are modified in order to provide an optimized and reduced dialog.

Consequences

The behavior of the implemented dialog is affected, since extracted questions are not indicated any more. Let $Q = \{q_1, \ldots, q_n\}$ be the original question set with $q_i \in \Omega_Q$, and $Q' = \{q_k, \ldots, q_m\}$ be the questions extracted from Q, i.e., $Q' \subseteq Q$.

Indication Rules
During the execution of the refactoring method we need to consider all indication rules targeting the question set Q. All rules indicating Q are modified so that they are also indicating the extracted question set Q'. Here, the order of indication is an important aspect: If the first question of the extraction set is the first question of the original question set Q, i.e., $q_k = q_1$, then we indicate Q' before Q; otherwise, Q is indicated before Q'. With this procedure the original indication sequence often can be preserved.
If follow-up questions $q \in Q$ are extracted without their parent question, then indication rules targeting q are modified so that they are indicating the extracted question set Q'.

Mechanics

The refactoring method is performed by the following procedure:

1. Apply the test suite to the knowledge system and abort, if errors are reported. Especially, consider the test knowledge for the dialog behavior.
2. Select the question set Q and define the question set $Q' = \{q_k, \ldots, q_m\}$ with $q_i \in Q$ to be extracted.
3. Create a new question set Q' at the position after the question set Q.

4. Move questions $\{q_k, \ldots, q_m\}$ to the question set Q'.
5. Modify indication rules that target the original question set Q and follow-up questions contained in Q (see *Consequences* section).
6. Apply the test suite to the refactored knowledge system and cancel the restructuring method, if errors are reported; alternatively start a debug session. Especially, consider the test knowledge for the dialog behavior.

Example

The question set "*lab tests*" (*labtest*) with questions "*red blood cell*" (*rbc*), "*white blood cell*" (*wbc*), "*lt1*", and "*lt2*" should be simplified;

$$labtest = \{\, rbc, wbc, lt1, lt2 \,\}.$$

The questions "*rbc*" and "*wbc*" should be extracted to a new question set considering the blood parameters (*bp*). The following rule is contained in the knowledge base

$$r_1 = cond(r_1) \rightarrow indicate(labtest).$$

With the transformation we obtain the question sets $labtest = \{\, lt1, lt2 \,\}$ and $bp = \{\, rbc, wbc \,\}$, and the rule is modified as follows

$$r_1' = cond(r_1) \rightarrow indicate(bp, labtest).$$

The generated question set is indicated before the original question set because the first question of "*labtest*", i.e. the question "*rbc*", is extracted to "*bp*".

Related Methods

COMPOSEQUESTIONSETS (inverse).

Further Refactoring Methods

We briefly sketch the following refactoring methods REMOVEDIAGNOSIS, REMOVEQUESTION, TRANSFORMNUMINTOOC, and EXTRACTABSTRACTION.

REMOVEDIAGNOSIS *and* REMOVEQUESTION. These methods are executed if the corresponding diagnosis or question was detected as a lazy object. However, simply deleting the object can cause conflicts within the related knowledge. For example, if the object is contained in a rule condition, then the developer has to decide (e.g., by a default value) whether to delete the entire rule or only the affected sub-conditions. Furthermore, the deletion of the objects can cause deficiencies, such as ambivalent or redundant cases. Default settings can support the developer by removing deficient cases automatically.

TRANSFORMNUMINTOOC. This refactoring method converts a numerical question into an one-choice question, and is only applicable if the numerical value range can be disjointly partitioned given the available knowledge for the numerical question. Otherwise, the method is aborted. The generated partitions are mapped to choice values. Consequently, the generated value range is an ordered sequence. Notice, that often a post-processing of the generated partitions is reasonable, e.g., by defining a more coarse value range. Rule conditions containing the numerical questions are mapped to a choice value according to the disjunctive partition. Analogously, findings contained in cases are mapped to a choice finding. With a given definition of the partition this method can be fully automated.

EXTRACTABSTRACTION. A conjunction F of findings can be aggregated into an abstract finding f using the EXTRACTABSTRACTION refactoring method:

1. A new rule is created: the rule action is f, and the rule condition is given by F.
2. All rules using F in their rule condition are modified such that the modified rule condition contains f instead.

Case-based knowledge is not changed by the EXTRACTABSTRACTION method by default, i.e., the aggregated findings are not replaced with the abstract finding f. More generally, we can introduce rules of the form

$$r \;=\; q_1{:}X_1 \wedge \ldots \wedge q_n{:}X_n \;\rightarrow\; q{:}(X_1, \ldots, X_n),$$

where the X_i are variable symbols which can be instantiated by values $v_i \in dom(q_i)$, for refactoring several rules at the same time. If a rule condition contains a conjunction $F = q_1{:}v_1 \wedge \ldots \wedge q_n{:}v_n$ which is an instance of the condition of r, then we can replace it by the corresponding instance $f = q{:}(v_1, \ldots, v_n)$ of the action of r. Observe that the domain of the new abstract question q is the cross product of the domains of the original questions: $dom(q) = dom(q_1) \times \ldots \times dom(q_n)$. In many cases a subsequent editing of the modified rules r and the abstract question q by the user is necessary. E.g., using COARSENVALUERANGE we can condense the cross product to a simpler domain.

5 Automated Refactorings with d3web.KnowME

The system d3web.KnowME is a highly integrated workbench for the development of diagnostic knowledge systems. d3web.KnowME is the successor of the knowledge acquisition tool of D3 [16], which has been successfully applied in many medical, technical, and other domains. The workbench offers visual editors for implementing various types of knowledge, e.g., heuristic rules, case-based knowledge, and model-based approaches. Furthermore, the development process is supported by an automated test tool and a (preliminary) refactoring browser. The significance of combining refactoring methods with automated tests during the development of knowledge systems was discussed in [2]. The implementation of the test tool provides advanced methods for the identification of bad smells, e.g. lazy knowledge objects or integrity tests for detecting anomalies in knowledge. Furthermore, the correctness of the knowledge can be

validated by, e.g., empirical testing and unit cases. The refactoring browser is under development, and currently only offers simple methods for the extraction/composition of question sets and for the modification of the type of questions, e.g. transforming a multiple-choice question into a set of semantically equivalent yes/no questions. A further extension of the browser by more sophisticated methods, that consider an interactive resolution of generated conflicts, is planned and will be available in the near future. A recent version of the d3web system can be downloaded at `http://www.d3web.de`.

6 Conclusion

We have introduced a structured approach for the automated modification of diagnostic knowledge systems. Refactoring methods are very useful for changing the design of (large) knowledge bases. When compared to refinement techniques the main goal of the presented methods is not the improvement of the system's accuracy, but the design of the implemented knowledge. Such refactorings of the knowledge are commonly initiated by the developer and are motivated by design anomalies detected in the design of the knowledge base. Furthermore, the knowledge is often restructured in order to facilitate a simplified extension of the knowledge base in a subsequent step.

In large knowledge systems design changes are complex and error–prone. For this reason, developers commonly avoid even important changes. We propose to use automated tools for these tasks but claim that structural approaches for the modification of knowledge are a necessary requirement for the application of automated tools. Thus, a method can be supported by interactive wizards that perform the implied adaptations.

The *automatization* of the refactoring and consequent adaptations allows for the accomplishment of even complex changes of a knowledge base, that were very difficult to perform manually in the past. For example, the reduction of a symbolic value range (performed by the method CoarsenValueRange) can imply the change of hundreds of rules and thousands of test cases if applied to a real world knowledge base. Hence, such modifications were not performed in the past, even if indicated by experience gained by a real world application of the system.

Due to the limited space a comprehensive catalog of typical refactorings could not be given in this paper; only two refactorings methods were described and some others were sketched. In [1] more methods are introduced in detail and in conjunction with the implications for rule-based, case-based, and set-covering knowledge. Furthermore, appropriate test methods for the refactoring methods are discussed.

We have described refactoring methods that consider the modification of small spots of the knowledge base. In the future we are planning to work on larger methods, i.e., *big refactorings*, that are applied in order to modify the entire design of the knowledge base. A promising motivation for big refactorings is the integration of *knowledge formalization patterns*. The idea of knowledge formalization patterns [17] is comparable to design patterns [6] known from software engineering, by providing a guideline for the developers on how to formalize and structure their knowledge. Appropriate refactoring methods can support the developer in modifying an existing knowledge base so that it complies with the specification of a pattern.

Since the definition and implementation of new refactoring methods is a complex task, we are developing methods for specifying refactorings in a declarative way. For this purpose we use a semi-structured knowledge representation based on XML, and we apply logic-based methods for managing and updating. The proposed approach was already applied for querying and visualizing knowledge bases, and presented in [19]. Such visualization techniques are especially useful for the inspection of (parts of) the knowledge base before starting a refactoring.

References

1. Joachim Baumeister. *Agile Development of Diagnostic Knowledge Systems*. PhD thesis, University Würzburg, Germany, 2004.
2. Joachim Baumeister, Dietmar Seipel, and Frank Puppe. Using Automated Tests and Restructuring Methods for an Agile Development of Diagnostic Knowledge Systems. In *Proceedings of the 17th International Florida Artificial Intelligence Research Society Conference (FLAIRS-2004)*. AAAI, 2004.
3. Robin Boswell and Susan Craw. *Organizing Knowledge Refinement Operators, In: Validation and Verification of Knowledge Based Systems*, pages 149–161. Kluwer, Oslo, Norway, 1999.
4. Hans-Peter Buscher, Ch. Engler, A. Führer, S. Kirschke, and Frank Puppe. HepatoConsult: A Knowledge-Based Second Opinion and Documentation System. *Artificial Intelligence in Medicine*, 24(3):205–216, 2002.
5. Martin Fowler. *Refactoring. Improving the Design of Existing Code*. Addison-Wesley, 1999.
6. Erich Gamma, Richard Helm, Raplh Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
7. Yolanda Gil and Marcelo Tallis. A Script-Based Approach to Modifying Knowledge Bases. In *Proceedings of the 14th National Conference on Artificial Intelligence and 9th Innovative Applications of Artificial Intelligence Conference (AAAI/IAAI)*, pages 377–383. AAAI, 1997.
8. Asunción Gómez-Pérez, Mariano Fernándes-López, and Oscar Corcho. *Ontological Engineering*. Springer Verlag, 2004.
9. Asunción Gómez-Pérez and Dolores Rojas-Amaya. Ontological Reengineering for Reuse. In *Proceedings of the 11th European Workshop on Knowledge Acquisition, Modeling and Management (EKAW 1999)*, pages 139–156, 1999.
10. Matthias Hüttig, Georg Buscher, Thomas Menzel, Wolfgang Scheppach, Frank Puppe, and Hans-Peter Buscher. A Diagnostic Expert System for Structured Reports, Quality Assessment, and Training of Residents in Sonography. *Medizinische Klinik*, 3:117–22, 2004.
11. Ioannis Iglezakis and Thomas Reinartz. Relations between Customer Requirements, Performance Measures, and General Case Properties for Case Base Maintenance. In *Proceedings of the 6th European Conference on Case-Based Reasoning (ECCBR 2002)*, LNAI 2416, pages 159–173, Aberdeen, Scotland, 2002. Springer Verlag.
12. Rainer Knauf, Ilka Philippow, Avelino J. Gonzalez, Klaus P. Jantke, and Dirk Salecker. System Refinement in Practice – Using a Formal Method to Modify Real-Life Knowledge. In *Proceedings of 15th International Florida Artificial Intelligence Research Society Conference 2002 Society (FLAIRS-2002)*, pages 216–220, Pensacola, FL, USA, 2002.
13. Karl-Werner Lorenz, Joachim Baumeister, Christian Greim, Norbert Roewer, and Frank Puppe. QualiTEE - An Intelligent Guidance and Diagnosis System for the Documentation of Transesophageal Echocardiography Examinations. In *Proceedings of the 14th Annual Meeting of the European Society for Computing and Technology in Anaesthesia and Intensive Care (ESCTAIC)*, Berlin, Germany, 2003.

14. William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, Urbana-Champaign, IL, USA, 1992.
15. Alun Preece and Rajjan Shinghal. Foundation and Application of Knowledge Base Verification. *International Journal of Intelligent Systems*, 9:683–702, 1994.
16. Frank Puppe. Knowledge Reuse among Diagnostic Problem-Solving Methods in the Shell-Kit D3. *International Journal of Human-Computer Studies*, 49:627–649, 1998.
17. Frank Puppe. Knowledge Formalization Patterns. In *Proceedings of PKAW 2000*, Sydney, Australia, 2000.
18. Thomas Reinartz, Ioannis Iglezakis, and Thomas Roth-Berghofer. On Quality Measures for Case Base Maintenance. In *Proceedings of the 5th European Workshop on Case-Based Reasoning (EWCBR 2000)*, pages 247–259, 2000.
19. Dietmar Seipel, Joachim Baumeister, and Marbod Hopfner. Declaratively Querying and Visualizing Knowledge Bases in XML. In *Proceedings of the 15th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2004)*, pages 140–151, 2004.