Agile Development of Diagnostic Knowledge Systems

Dissertation zur Erlangung des

naturwissenschaftlichen Doktorgrades

der Bayerischen Julius–Maximilians–Universität Würzburg

vorgelegt von

Joachim Baumeister

aus
Würzburg

Würzburg, 2004

Eingereicht am: 18.03.2004

bei der Fakultät für Mathematik und Informatik

1. Gutachter: Prof. Dr. Frank Puppe
2. Gutachter: Prof. Dr. Dietmar Seipel

Tag der mündlichen Prüfung: 02.07.2004

To my family

Foreword

Today, the development of knowledge-based diagnostic systems, e.g. in medicine or service support, is not a principal problem, but success depends on the costs/benefits relation. The benefits of interactive diagnostic systems depend on a trade-off between the accuracy of the inferred diagnoses and therapies and the duration of the dialog for data gathering. It is extremely difficult to assess the details of the best trade-off in advance. This lack of specification poses serious problems for systematic, document-centered development processes. Therefore, early feedback is necessary for tuning knowledge engineering by avoiding both too detailed and too superficial knowledge modeling. However, rapid prototyping approaches are not satisfactory neither, because they result in unstructured knowledge bases being difficult to maintain.

Joachim Baumeister addresses this key problem by transferring and adapting so called agile process models from software to knowledge engineering. His model emphasizes early knowledge formalization and thus early feedback, combined with continuous monitoring of a good structure of the knowledge base. For detected design anomalies he offers a catalog of restructuring methods to improve the structure in small steps inspired by refactoring methods in software engineering. To avoid the introduction of new errors during restructuring, the quality of the knowledge bases is tightly monitored with constraints and partial test cases. These steps are integrated with approved more global process models for developing diagnostic systems: the introduction of knowledge containers for ontological strategic, structural and support knowledge, and various patterns (templates) for formalizing heuristic, case-based and set-covering knowledge sometimes supported by inductive machine learning methods.

The process model is implemented in the Java-based diagnostic shell d3web and successfully applied in an environmental and a medical project. It promises in particular for projects with a small or medium budget a significantly increased control for building cost-effective and maintainable diagnostic systems.

[illegible]

Prof. Dr. Frank Puppe

Abstract

The success of diagnostic knowledge systems has been proved over the last decades. Nowadays, intelligent systems are embedded in machines within various domains or are used in interaction with a user for solving problems. However, although such systems have been applied very successfully the development of a knowledge system is still a critical issue. Similarly to projects dealing with customized software at a highly innovative level a precise specification often cannot be given in advance. Moreover, necessary requirements of the knowledge system can be defined not until the project has been started or are changing during the development phase. Many success factors depend on the feedback given by users, which can be provided if preliminary demonstrations of the system can be delivered as soon as possible, e.g., for interactive systems validation the duration of the system dialog.

This thesis motivates that classical, document-centered approaches cannot be applied in such a setting. We cope with this problem by introducing an agile process model for developing diagnostic knowledge systems, mainly inspired by the ideas of the eXtreme Programming methodology known in software engineering. The main aim of the presented work is to simplify the engineering process for domain specialists formalizing the knowledge themselves. The engineering process is supported at a primary level by the introduction of knowledge containers, that define an organized view of knowledge contained in the system. Consequently, we provide structured procedures as a recommendation for filling these containers. The actual knowledge is acquired and formalized right from start, and the integration to runnable knowledge systems is done continuously in order to allow for an early and concrete feedback. In contrast to related prototyping approaches the validity and maintainability of the collected knowledge is ensured by appropriate test methods and restructuring techniques, respectively. Additionally, we propose learning methods to support the knowledge acquisition process sufficiently.

The practical significance of the process model strongly depends on the available tools supporting the application of the process model. We present the system family d3web and especially the system d3web.KnowME as a highly integrated development environment for diagnostic knowledge systems. The process model and its activities, respectively, are evaluated in two real life applications: in a medical and in an environmental project the benefits of the agile development are clearly demonstrated.

Acknowledgments

The topic of this thesis is the agile development of diagnostic knowledge systems. The writing of the thesis was a kind of agile development, too: Starting with a more or less precise idea of the main issues of the work, the particular topics were modified and replaced according to the results of own and other research results. Although done manually agile techniques as restructuring (e.g., extract section), testing (cross-proof reading with other sections), and learning (reading papers about new topics) proved to be suitable methods for finishing with a comprehensive thesis about agile knowledge engineering.

However, to be serious: My work actually was primarily supported by many people giving me advices, inspiring me in (sometimes contrary) discussions, or by simple encouragement, rather than by sophisticated techniques. First of all, I want to thank my supervisors Frank Puppe and Dietmar Seipel for providing me a perfect environment of research with open doors and open minds; with both I spent a lot of time in fruitful discussions about the practical and theoretical dimensions of knowledge systems. In the past years I learned so much about the practical and academic factors of successful research. Furthermore, I want to thank my colleagues of the AI working group at the University of Würzburg, especially Martin Atzmüller for being my partner in the *learning methods* dream-team. Norman Brümmer, Rainer Herrler, Alexander Hörnlein, Franziska Klügl-Frohnmeyer, Christoph Oechslein, and my other colleagues also supported me during the years. Last but not least, Petra Braun – the good heart of our group – helped me coping with various problems of bureaucracy and other ordinary things of life, or simply by extending my musical horizon. The practical parts of my work would have not been possible without Dr. Karl-Werner Lorenz and Dr. Michael Neumann, who were my partners in the medical project ECHODOC and the biological projects LIMPACT and ILMAX, respectively. I am grateful for their valuable and constructive input to my work. Besides my colleagues I would like to thank Michael Wolber, Ioannis Iglezakis, Sven Wahler, and Rüdiger Hain for reading and polishing parts of my thesis: I happily look back to fruitful and contrary discussions with Michael, Janni, and Rüdiger; Sven was and is a loyal support for such a long time. Thanks guys.

I thank my family for their encouragement and for providing me an anchor of life, especially my father and mother, my brother with his family, and my three grandparents. Finally, I dedicate this work to my two girls who are my own small family: you are the reason for the longest nights and also for my greatest pleasure. Love.

[illegible]

Joachim Baumeister

Contents

Part I.

The Knowledge Engineering Bottleneck

1. Introduction

Expert systems are among the most exiting new developments in computer science and technology. (...) Prototype systems are now in use around the world in areas such as medical diagnosis, mineral prospecting, chemical structure elucidation, and computer-system configuration.

Blurb of *Hayes-Roth et al., Building Expert Systems, 1983*

1.1. Research Goal and Context

Now, 20 years later, research has progressed and knowledge systems (expert systems) are all around us. Starting with academic systems, e.g. DENDRAL [25], in the 60's, intelligent programs are helping us to diagnose the medical condition of patients, or to detect faults of cars, copy machines, and large manufacturing plants. In the medical domain, knowledge systems are routinely used for supporting the physician during specialized examinations, e.g. the SONOCONSULT system [58], or are embedded in devices for monitoring and controlling the condition of critical patients, e.g., the recent weaning systems of Dräger. Also in the technical domain, many knowledge systems have been established and are in daily use, e.g. for diagnosing gas turbines [77].

However, although a lot of research has been done in the last decades, the development and maintenance of such systems is still a complex, difficult, and error-prone task. Often promising projects were withdrawn or not continued because of problems and errors during the development and maintenance phases. For the development of knowledge systems we distinguish prototypical approaches, e.g., by Budde et al. [26, 27], and document-centered/structured approaches, e.g., the CommonKADS methodology [119]. Experiences in several knowledge systems projects in the medical and technical domain have shown that both extremes are not sufficient for small-sized innovative projects. In such projects the estimation of the development costs is of major concern and projects with a vague scope are often canceled due to their unpredictability. Document-centered approaches are typically heavy-weight and the cost-effectiveness, i.e., the trade-off between the utility of the deployed system and the development costs, is difficult to judge upon the beginning. Thus, many documentary and design phases are passed before the implementation of the system is started. In contrast, prototypical approaches support the estimation process by initial pilot systems. Fast results and preliminary experiences can be drawn from such pilots, and a flexible adaptation and extension of these small systems is also simple. In the past, pilots were rapidly build and extended in an unstructured way. Consequently, prototyping approaches haven been criticized because of their lack of design and hard maintainability.

The focus of this work is the definition of a process model for the development of diagnostic knowledge systems. In general, the process model should achieve the following design goals:

- domain specialists should be able to self-acquire formalized knowledge, i.e., the domain specialist is the developer of the system
- the process should allow for an early and continuous feedback with respect to both the utility, usually requiring feedback of the end-user, and the costs for developing the knowledge base, which strongly depends on its granularity (level of detail).
- the process should emphasize the development of trustable systems, i.e., provide measures for determining the quality of the implemented knowledge
- the maintenance of developed systems should be significantly simplified by providing appropriate methods

In this thesis we present a process model which follows the just mentioned design goals. In addition, the presented process model was defined according to the following observations based on experiences gained in various knowledge system projects:

- the systems are build by small teams consisting of domain specialists, mostly 1-3 people
- the team is motivated, if early and successful results can be drawn; in fact, the projects are mostly continued because of the (successful) experience made with rapid pilots
- the scope and requirements of the planned system are often changing during the development
- the accuracy and also the efficiency of the system is a critical success factor of the deployed system

In the next section, we briefly introduce the process model, and we motivate how the design goals and observations from above are fulfilled.

1.2. Approach

We sketch the proposed process model which tries to balance the advantages and drawbacks of prototyping and document-centered approaches. Such a process model should be able to produce quick results, but they also should yield a system that is reliable and simple to maintain. The presented process model was inspired by the agile process model *eXtreme programming* (XP). In software engineering research and practice XP [19] has attracted great attention, and showed its significance in numerous projects. An agile process model has the following properties:

- an early, concrete and continuing feedback
- an incremental planning approach
- a flexible schedule of the development process
- the design process lasts as long as the system lasts

The presented agile process model and XP work on different domains: Whereas XP organizes the process of coding a software in a general purpose *programming language* (e.g., Java or C#), the presented agile process model defines the development of a knowledge system by insertion, change, and review of knowledge in a *declarative language*. The process model and the included practices, respectively, are adapted and refined in order to meet the requirements of knowledge system development. Whereas the general development phases *system metaphor*, *planning game*, *implementation*, and *integration* are simply adapted with respect to the characteristics of knowledge systems, the included practices are thoroughly customized for knowledge engineering. For example, appropriate *test methods* are defined that are suitable for analyzing the design and for validating the expected behavior of the implemented knowledge. Furthermore, the refactoring methods known from software engineering are refined with respect to the different types of knowledge; we call these methods *restructuring methods* since not all methods preserve the semantics of the knowledge system as it is claimed for refactoring techniques. Moreover, we extend the process model by the additional practice *learning*; by the knowledge of the author the automated generation of general purpose software was not considered so far, but such techniques can be very reasonable for the development of knowledge systems. Other agile practices like *pair programming*, *on-site customer*, and *collective ownership* are also discussed in the context of the development of knowledge systems, but they are not considered to be as relevant as the previously mentioned.

The actual implementation of the knowledge system is simplified by the introduction of *knowledge containers*: The used knowledge is organized into four different containers classified according to their use. For each container appropriate methods for testing, restructuring, and learning are given. The concept of knowledge containers is a specialized design abstraction appropriate for the development of knowledge systems and no counterpart in software engineering can be found so far. In the following, we describe the particular elements of the approach in more detail.

1.2.1. The Agile Process Model

The agile process model is a light-weight process model and consists of the following cycle: Analysis of the system metaphor, design of the planning game, implementation (plan execution: including tests, restructuring and maintenance), and integration. A new knowledge system project starts with the analysis of the system metaphor, which should include an overall plan of the knowledge system. Then, the development steps into a cyclic development phase, which consists of the planning game, the implementation of plans and the integration of the new implementation. This cycle is executed during the development phase and lasts as long as the system lasts. Since the process model is thoroughly explained in Chapter 2, we only briefly discuss the steps in the following:

The System Metaphor The system metaphor describes the basic idea and the designated goals of the knowledge system to be implemented, and it is used to facilitate a better communication between the developers (i.e., the domain specialists) and the users of the system. Thus, the metaphor stands for a *common system of names* and a common system description. Using a common system metaphor can greatly simplify the development and the communication between users and developers.

In general, we distinguish between a *local system metaphor* and a *global system metaphor*.

The local system metaphor defines the names and semantics of the basic entities used during the development of the system, e.g., diagnoses (solutions), questions (input), and cases (solved problems). The global system metaphor is used to describe the overall idea of the system to be implemented. Typical pre-defined global system metaphors are the *consultation system* (focussing on interactive problem solving), the *documentation system* (focussing on the standardized and correct acquisition of input data, and the *embedded system* (focussing on the inference and the technical integration of the system into an existing machine).

The Planning Game The planning game is the starting point for the development phases: During the planning game the developer and the user decide about the scope and the priority of future development, i.e., extensions or modifications of the current system. For each extension/modification a plan is defined, which is documented by *story cards*. Besides the desired functionality the costs and priority of each plan are estimated. Based on the estimated values of these factors the developer and the user define the next release by selecting story cards. They define the scope of the release by ordering the collected cards and defining a release deadline according to the risk estimations made before. It is worth noticing, that these factors provide a benchmark for feedback in order to enable adaptation of plan estimation in the future. The planning game provides a flexible method for guiding the development process of knowledge systems. On the one hand, plans are documented in story cards and deliver a structured sequence of the development process, in which the user as well as the developer are integrated. On the other hand, during the definition of stories the user and the developer specify the expected behavior of the planned knowledge extension, and thus prepare useful validation knowledge for the subsequent implementation phase. Furthermore, by providing methods for estimating and documenting implementation costs (derived from the implementation velocity), an accurate feedback can be given to assess the whole development process.

The Implementation The implementation phase considers the realization of the story cards specified in the planning game phase. In the context of the agile process model, the implementation phase follows a test-first approach: Any implementation of the functionality of a story is preceded by the implementation of appropriate tests. Therefore, we distinguish between a *test-implementation phase* and a *code-implementation phase*.

In the test-implementation phase the developer defines test knowledge describing the expected behavior of the new story to be implemented. The kind of test knowledge depends on the representational language of the code-implementation. It is easy to see that, e.g., the test knowledge for a rule-based knowledge representation can differ from the test knowledge of a case-based representation. Test knowledge needs to be suitable for automated tests, i.e., the results of the test can be evaluated automatically by the system. As a main idea of the process model, the continuous application of the cyclic process yields a suite of tests, which can be executed as a whole. The code-implementation phase considers the actual realization of the story, e.g., by acquiring and formulating new knowledge. Alternatively, the code-implementation consists of a restructuring task in order to simplify or adapt the design of the already implemented knowledge.

The Integration If the newly implemented functionality passes the corresponding tests and the test suite, respectively, then this knowledge is committed to the knowledge base. Since the integration is done continuously, we always can access a running system enclosing the currently implemented knowledge. For a reasonable integration additional tests need to be available, which are too time consuming to be included into the working test suite, but which are applied during integration to check more aspects of the functional behavior of the knowledge system. We call these tests *integration tests*. Integration tests often contain a larger number of previously solved cases, which can be run against the knowledge system. These previously solved cases typically contain a set of question-answer pairs and a set of expected diagnoses for these pairs, but sometimes also knowledge about dialog behavior is available. Running thousands of cases can take several minutes or hours. Therefore it is not practical to include them into the working test suite, since the suite usually is applied many times during the implementation of a story. Nevertheless, before the integration of a new version of the knowledge system the integration tests are an essential indicator for the correct behavior of the knowledge system.

1.2.2. Knowledge Containers as a Design Abstraction

During the application of the agile process model the knowledge system is modified by the extension and redesign of existing knowledge. The introduction of knowledge containers simplifies this modifications. Thus, we classify the applied knowledge according to its use into ontological, structural, strategic, and support knowledge. For all containers and used knowledge representations, respectively, we present suitable test methods, restructurings, and learning methods (if possible).

The *ontological knowledge container* (Chapter 4) collects all basic entities of the knowledge system, i.e., diagnoses and questions. Diagnoses as well as questions are typically grouped by problem areas and question sets, respectively; the ontological knowledge container also contains hierarchical information between these basic entities. The *structural knowledge container* (Chapter 5) embodies the inferential knowledge used by the diagnostic system. In literature, numerous approaches have been proposed to represent structural knowledge, e.g., production rules, case-based reasoning, Bayesian networks or several model-based approaches like set-covering models. The *strategic knowledge container* (Chapter 6) provides knowledge applied to guide the user dialog of the knowledge system. To avoid needless data acquisition costs, the user typically is guided through an appropriate dialog path. Often, strategic knowledge is implemented by indication rules, which activate the questionary of specified questions or question sets according to the observation of findings or inferred diagnosis states. Informal *support knowledge* (Chapter 7) is used for enriching the ontological knowledge (e.g., diagnoses, questions) with additional information. For example, informal support knowledge can consist of text book entries or multimedia content (pictures, movies) used for explanation.

Knowledge containers provide an organized view of the knowledge base, and they can simplify the maintenance and validation of the knowledge system. The classification of knowledge with respect to its usage goes back to Clancey [31] who introduced the terms structural, strategic, and support knowledge.

In the context of the agile process model we need to consider appropriate validation techniques and restructuring methods for each particular knowledge container. However, especially the restructuring of ontological objects (e.g., modify the type of a question) can propagate changes to the remaining knowledge containers (e.g., rules concerning the modified question).

1.2.3. Explicit versus Unified Languages

When defining knowledge, and especially structural knowledge, we have to decide about the actual representational language. In the knowledge engineering community two different research directions have emerged, which we call *unified* and *explicit* languages in this thesis.

Firstly, the use of a unified language is proposed to be applied for all kinds of knowledge systems projects. From a historical viewpoint, the application of logical languages (e.g., Horn logic) to be used as a unified language has a long tradition in knowledge system research, and it is still considered to be very successful for building knowledge systems; e.g., Lucas et al. [69] use Horn formulae representing knowledge for pacemaker reprogramming. For the representation of uncertain knowledge Bayesian networks have been established as state of the art; e.g., the PATHFINDER system [56] for diagnosing lymph-node diseases, and the HEPAR-II system [86] for diagnosing liver disorders. The advantage of using unified languages is their well-understood semantics and broad scientific acceptance. Thus, for logical languages and Bayesian networks the complexity of knowledge acquisition and knowledge inference is thoroughly investigated. The main drawback of unified languages is their level of abstraction: Usually, domain specialists are not familiar with logic or probability theory. For this reason, often a knowledge engineer becomes necessary for translating the collected expert knowledge into logical terms or Bayesian relations. The disadvantages of using a knowledge engineer have been repeatedly discussed in the past, e.g., the possibility of errors during knowledge translation, the difficult maintenance of knowledge, the possibility of misunderstandings, and the costs/availability of knowledge engineers.

Secondly, the use of explicit languages appropriate for the actual task are propagated. Explicit languages are trying to map the *mental models* used by the specialists applied during problem-solving; for each sub-task contained in the knowledge system the appropriate language is selected. Examples for explicit languages are scoring rules, case-based diagnosis, and set-covering models. The use of knowledge representations similar to the mental models of domain specialists comes with several advantages: Thus, there is no need for a knowledge engineer in most cases, because the specialists are able to formalize the knowledge themselves. Furthermore, it is easier to provide specialized process models and suitable tools for each explicit representation. E.g., the process of building a knowledge system using rules differs from a knowledge system based on cases, but the acquisition of the different types of knowledge can be simplified by adequate methods and tools.

In this work, we focus on the use of explicit languages motivated by mental models. Experience has shown that domain specialists are able to formalize and maintain the required knowledge, if the applied representation is similar to a familiar mental model. Therefore, we discuss the integration of the explicit languages like abstraction rules, categorical rules, score-based rules, case-based reasoning, and set-covering models into the proposed agile process model. All presented languages have been applied in real world knowledge systems.

1.2.4. Highly Integrated Environments

The use of explicit representational languages allow for specialized tools supporting the application of the agile process model. Visual programming environments can support even unexperienced users in building a knowledge system, i.e., in formalizing and maintaining the required knowledge.

The application of the agile process model is supported by the highly integrated knowledge modeling environment d3web.KnowME, which provides appropriate editors for the definition of the presented knowledge containers. Furthermore, editors for the acquisition and the use of test knowledge, together with the application of restructuring methods is provided by the tool.

1.3. Results

The presented work describes the agile development of diagnostic knowledge systems. It comprises both the initial phases of a project with the definition of the system metaphor as well as the following development process by the planning game, the implementation, and the integration; thus, early and continuous feedback concerning the cost-effectiveness are combined with a structured methodology. The implementation phase is identified as the most important part of this process, and it is simplified by the introduction of different knowledge representations similar to mental models applied by the domain experts. According to the project requirements the developer can choose a suitable representation. Furthermore, knowledge is classified into different knowledge containers providing an abstract overview of the different types of knowledge. The agile process model and the use of knowledge containers were presented in Baumeister et al. [17]. For the used knowledge representations we provided appropriate knowledge acquisition procedures. For example, an incremental development process for set-covering models was presented and analyzed in Baumeister et al. [16, 15]. The developer is supported during the implementation phase by the agile activities *testing*, *restructuring*, and *learning*. All three activities form a significant contribution to the knowledge engineering task, and we discuss their benefits in the following.

For different knowledge representations we present appropriate automated *test methods*, that are able to evaluate their test results by themselves. Thus, testing can be applied continuously during the development process without increasing the workload for the developer. We distinguish two types of test methods: The first type of methods is used to monitor the correct behavior of the implemented knowledge, and errors or warnings are reported in the case of a discovered defect. The second type tries to find deficiencies of the knowledge design using specialized metrics. An advice is reported in the case of ambiguous or unstructured knowledge design, which we call *bad smells* according to analogous metrics known from software engineering. Often knowledge formalization patterns can give clues for repairing deficient knowledge design.

The second key activity of the agile process model are *restructuring methods*. They provide algorithmic procedures for typical modifications of the implemented knowledge in order to increase the simplicity of the knowledge or in general for improving the design. Furthermore, restructurings are a suitable technique for the stepwise implementation of knowledge systems; pilot systems are modified by such methods according to changing project requirements, and larger systems can be refactored if the knowledge design becomes messy. For this reason, restructuring methods are a significant contribution for facilitating the (evolutionary) development of high-quality knowledge systems. The significance of test methods and restructurings was explained in Baumeister et al. [18].

Learning methods are the third key activity of the agile process model. If sample cases of the domain are available, then they can provide a suitable technique for the automatic generation of prototypes or they are used for extending an existing knowledge base. In the agile process model the quality of a learning method is determined not only by the accuracy but also by the simplicity of the learned patterns. The simplicity is an important aspect since learned knowledge should be integrated into the manual knowledge development process. For several patterns of knowledge we present learning methods focussing on the accuracy and the simplicity of the learning result. We have presented suitable learning methods for different types of knowledge in [14, 8, 9, 10].

Our *implementation* demonstrates the practical applicability of the presented process model and their activities, respectively, and we present the implementation of the visual knowledge modeling environment d3web.KnowME. The tool supports the developer during all phases of the project, for example by providing a planning editor, specialized editors for different types of knowledge, an automated test tool, a restructuring interface, and a visual debugger.

The process model and their activities, respectively, were (partially) evaluated in two *real life projects* and the benefits of the presented research is clearly demonstrated. The project ECHODOC (formerly QUALITEE) was described in [68], the evaluation of LIMPACT project can be found in [79, 83]. A position paper of the ILMAX system, an information center, was presented in [81].

1.4. Structure of this Work

This work is structured into four parts and two appendices, that we summarize in the following:

The *first part* gives an introduction into the basic concepts of the agile development of diagnostic knowledge systems. It focusses on the social aspects of the implementation of the process model (e.g., defining a suitable metaphor for the planned system, planning the most current development steps, monitoring and steering the development process), and the basic ideas of the phases knowledge implementation and continuous integration. As the key practices of the agile process model we propose automated testing of knowledge, the application of organized restructurings on knowledge, and the use of learning methods.

The *second part* of this work considers the details of knowledge system development. We introduce the concept of knowledge containers and extensively discuss the particular containers including the application of the agile practices *testing*, *restructuring*, and *learning* for each container.

The *third part* describes the practical aspects of this work: The system family d3web is introduced, which contains the highly integrated knowledge modeling environment d3web.KnowME. We motivate, that a tight integration of specialized editors for the acquisition of knowledge and corresponding test knowledge in connection with integrated restructuring methods can greatly simplify the development of diagnostic knowledge systems. Furthermore, experiences made with the process model are reported. The process model was partly applied in a project for building a biological knowledge system, and fully adopted in another project developing a medical knowledge system.

The *forth part* concludes this thesis with a summary of the presented work and an outlook for promising research directions in the future.

Two appendices to this work summarize two important aspects of agile knowledge engineering: restructuring and testing. *Appendix A* presents a compact catalog of selected restructuring methods. Each restructuring method is coherently explained with respect to its motivation, the consequences of its application, a detailed description of its mechanics, an example, and related methods. *Appendix B* summarizes the presented testing methods in a tabular manner with respect to its corresponding knowledge container, its required test knowledge, and further characteristics.

2. An Agile Process Model for Evolutionary Development of Knowledge Systems

In this chapter, we will introduce an agile process model for developing diagnostic knowledge systems. This process model enables developers to flexibly build knowledge systems, since changes during the development phase can be implemented easily. We first motivate the usage of agile methods. After that, we introduce the agile process model and conclude the chapter with a discussion and a comparison with related knowledge engineering approaches.

2.1. Motivation for an Agile Process Model

The development of knowledge systems is still a complex and costly task. Experiences with real world applications show that systems...

- are costly to develop, i.e., they need more time and resources as expected and planned
- show unexpected and/or faulty behavior
- do not fulfil the customer's requirements
- are hardly maintainable, when delivered into productional use

A lot of research has been done to cope with the problems stated above.

Starting with rapid prototyping approaches (e.g., Budde et al. [26]) some efforts have been undertaken to define a more structured methodology for the development of knowledge systems. Furthermore, from the repeated programming of custom-tailored systems for each project, the knowledge engineers switched to reusable shells or components. Thus, inference engines and knowledge acquisition tools are reused and the domain expert/knowledge engineer can focus on the problem of knowledge modeling. The most prominent example for a structured and reusable knowledge modeling approach is KADS [118], which was followed by CommonKADS [119]. These approaches focussed on a predetermined method for creating models describing the intended knowledge system. In this way, these methods are document-centered and design-oriented. However, we have experienced, that knowledge base development often faces problems, when using such approaches:

- For a lot of projects the *technical feasibility* is not known in advance. Then, a quick prototype needs to be developed in short time, to convince the customer of the feasibility of the focused project. Design-oriented processes do not (sufficiently) support the quick creation of a prototype, which then can be integrated into a real, subsequent project.

- In many cases a *full specification* of the desired system properties cannot be given in advance. Moreover, the requirements of the system often change during the development phase. Requirement changes arise with the growing confidence of the developers in the applied representation or often come up because of changing demands of the customer. Requirement changes in a design-oriented process model involve costly changes of documents, design and planning efforts.

These typical problems are not unique for knowledge base development; they also have received great attention for general software engineering research.

Recently, *agile methods* have been presented that allow for an evolutionary design and development of software and that claim for coping with the problems stated above, e.g., Cockburn [32].

The most prominent agile representative is called *eXtreme Programming* [19], which has attracted huge attention in the last years because of several successful projects, e.g., the DaimlerChrysler *purchase request tracking system* (C4) described in [19], and the *chrysler comprehensive compensation* (C3) project described in [5].

Extreme Programming (XP) shows interesting characteristics, which can be summarized by a quote from its founder Kent Beck [19]:

> XP is a light-weight methodology for small-to-medium-sized project teams developing software in the face of vague or rapidly changing requirements.

In this way, it distinguishes from other methodologies by

- early, concrete and continuing feedback. This is realized by short implementation cycles. The result of each cycle is a deliverable system.
- incremental planning approach: The project starts with a coarse overall plan, which evolves during the life-time of the project.
- allowing for the flexible schedule of implementation of functionality which responds to the customer needs.
- producing reliable software. One of the central ideas of XP is the creation of automated tests written by customers and developers to control the development of the system and its evolution.
- a fundamentally evolutionary design and development process, which lasts as long as the system lasts.

In the following sections we introduce a novel agile process model for developing diagnostic knowledge systems. We transfer most of the principles and techniques applied for XP to an approach appropriate for the knowledge engineering task.

Of course, an evolutionary model can not help with ”chaotic” projects for which the focussed production system changes from time to time. In fact, a proper pre-design of the system is still one major key to the successful conclusion of a project. However, the presented process will provide restructuring methods and automated tests for a safe redesign of existing knowledge systems. Furthermore, we will limit our approach to configurable role-limiting methods [91] (CRLM). For short, CRLM are based on role-limiting methods [71] (RLM), which are implementations of strong problem-solving methods. Since RLM focus on a selection of problem-solving methods, user-friendly shells with convenient knowledge acquisition tools can be provided, which efficiently support the developer during implementation. Their main disadvantage is their inflexibility according to the variety of possible requirement within a knowledge system project. CRLM try to reduce this, by offering a set of problem-solving components, that can be (mostly) freely configured with respect to the project requirements.

2.2. Steps of an Agile Process Model

The agile process consists of a cycle of steps, which are described in detail. The agile process model is a light-weight process model and consists of the following steps:

1 Analysis of the system metaphor
2 Design of the planning game
3 Implementation (plan execution)
Including tests, restructuring and maintenance
4 Integration
5 Start a new planning game

A new knowledge system project starts with the analysis of the system metaphor, which should include an overall plan of the designated knowledge system. Then, the development steps into a cyclic development phase, which consists of a planning game, the implementation of plans, and the integration of the new implementation. This process model is executed during development phase and lasts as long as the system lasts.

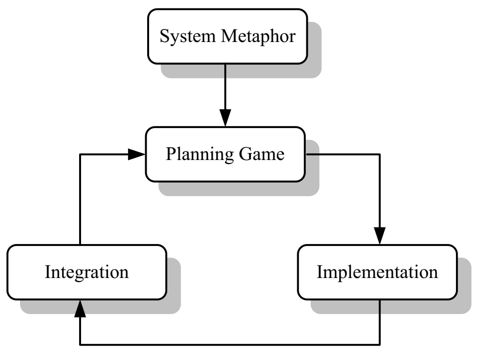


Figure 2.1: The agile process model for developing knowledge systems.

The agile process model is depicted in Figure 2.1 and we will describe the steps of the model in more detail in the following.

2.2.1. System Metaphor

The system metaphor describes the basic idea and designated aims of the knowledge system to be implemented.

It is used as a media of communication between the developers and users of the system. Thus, the system metaphor describes a common *system of names* and a common system description. Using a common system metaphor can greatly simplify the development and the communication between users and developers.

We can distinguish between a global and a local system metaphor. The *global system metaphor* describes the overall idea of the system, and the *local system metaphor* defines the set of basic names, which are used to implement the global metaphor.

Local System Metaphor For diagnostic knowledge systems the main part of the local system metaphor is pre-defined, because a diagnostic knowledge system always performs a fixed task, i.e., obtaining input data and deriving solutions, which explain the given input. The local metaphor consists of a set of classes for the basic entities, and we see that there exist alternative names for them in the literature:

- *Diagnosis / solution:* Instances of the diagnosis class are inferred as the result of a diagnostic system run. Often, the alternative name *solution* is used for describing a composite set of diagnoses.
- *Finding (symptom, parameter, observable, observation, data, input, fact):* The class of entities, that are mostly given as input to the diagnostic system and that are used to infer diagnoses.
- *Question set (question container, test):* A composite entity class, containing a group of findings belonging together in a sense. It is often used to structure the set of available findings into meaningful partitions.
- *Problem (problem description)*: A problem is described by a set of findings, that appear together in a given situation. This problem is given to a diagnostic knowledge system in order to infer one or more diagnoses.
- *Case*: The instance of a case consists of a problem, i.e., set of findings, and a set of solutions, that explain the given observation. Sometimes cases additionally contain a set of information (mostly unstructured and informal) describing the solved case in more detail.

To avoid communication problems it is advisable to commit to an unique naming convention. Therefore, if not necessary, no alternative names should be used to describe the same entity class.

Global System Metaphor Beyond the basic entities of a diagnostic knowledge system there exists a set of typical application classes that are used to define the global system metaphor. Each class describes the focus of the designated application, which is planned for development. In the following, we describe the four most typical application classes:

Documentation System A documentation system is focussed on a high quality data acquisition and usually implements a detailed dialog control. This kind of class is often implemented, when high quality data entry is the most important feature of the knowledge system. In this context the term *quality* is defined by completeness of the input, the structure of the data set, and the correctness of the input. For documentation systems a guided dialog control and consistency tests need to be implemented.
Example: In the medical domain, the standardized documentation of examinations often is associated with legal requirements. Thus, for any medical attendance a consistent documentation is obligatory. Moreover, the application of documentation systems can be a starting point for mass-acquisition of sample data, reused for machine learning or statistical analysis. Example systems that also implement the documentation system metaphor are SONOCONSULT [142] and ECHODOC [140].

Embedded System Sometimes the diagnostic knowledge system is embedded in another system as a component. Then, the system receives its findings from the overlying application and returns its diagnoses back to the application. Because usually data is not manually entered, often no dialog control is needed. Embedded systems are also called *closed-loop systems*.
Example: Diagnostic knowledge systems are often instances of embedded systems. For example, in technical domains diagnostic knowledge system can be embedded in printing machines. In this example, the system receives the error codes directly from the printing machine interface and diagnoses the current status of the machine. TIGER is another example taken from a technical domain, which diagnoses gas turbines used at oilrigs [77, 78]. In the medical domain there exist also examples of embedded systems, e.g., in [69] a system is presented as a successful implementation for reprogramming cardiac pacemakers. Furthermore, there has been a lot of research on on-board diagnosis of automobile engines [129].

Consultation System Consultation systems are the typical kind of diagnostic knowledge system: The user is guided through a structured dialog in order to enter findings for the given problem. Then, the system uses the findings to infer diagnoses, that are presented for the user. We can see, that a consultation system is a kind of documentation system, because it often contains a guided data entry. Furthermore, it also includes a diagnostic component to infer diagnoses. *Critiquing systems* can be seen as an extension of consultation systems, since they also infer suitable solutions. However, critiquing systems differ from ordinary consultation systems by not presenting the derived solution in any case but only on demand. Additionally, they are able to criticize the solution manually entered by the user, e.g., if the user's solution could not be derived by the system or important system solutions were ignored by the user. For a detailed discussion of critiquing systems with respect to consultation systems we refer to Puppe [101].
Example: There have been a lot of examples of diagnostic consultation systems in the medical domain. Classical systems are MYCIN [24] or PATHFINDER [56]. A recent and successful example is HEPATOCONSULT [28], which was developed with the shell-kit D3 [104]. In the geo-ecological domain, we have made own experi-

ences with consultation system LIMPACT [83, 80].

Information center The information center does not focus on well elaborated diagnostic inference knowledge (e.g., as needed for a consultation system), but mostly consists of informal knowledge like documents, multimedia content, and structured cases. Usually, this content is hierarchically structured and methods for intelligent retrieval and navigation are available. Obviously, this kind of system can be also used for solving a given problem, i.e., by browsing the content in order to find helpful contents for a given problem. On the one hand, the user is more flexible and independent when compared to a consultation system, since he has not to traverse a predefined questionary. On the other hand, the user is more responsible for obtaining a suitable solution for his problem, since he has to browse through informal content which needs to be interpreted by himself and adapted to the given problem situation.
We differ information centers from usual information systems by their capabilities of intelligent navigation: Whereas pure information systems only provide a simple navigation and search facility to obtain the desired content, an information center at least should offer intelligent dialogs and questionnaires in order to narrow the current problem and offer appropriate content for the problem.
Example: ILMAX [81] is an information center containing various kinds of content about the regeneration of the river Ilm located in Germany, e.g., PDF documents, Excel sheets, sample case bases, and formalized knowledge. ILMAX is currently under development.

The global metaphors described above are not an exclusive list of all possible application classes. Furthermore, classes may be combined or extended to fulfill the project requirements.

An example is the mixture of an embedded system with a consultation system: For instance in technical domains it is very common to link the diagnostic system with the system to be diagnosed (e.g., printing machine). Then, the diagnostic system reads all available error codes and findings from the machine in an embedded way. If necessary, the system pops-up with a dialog to obtain further, mostly refining questions that can establish or exclude a suggested diagnosis inferred by the embedded dialog.

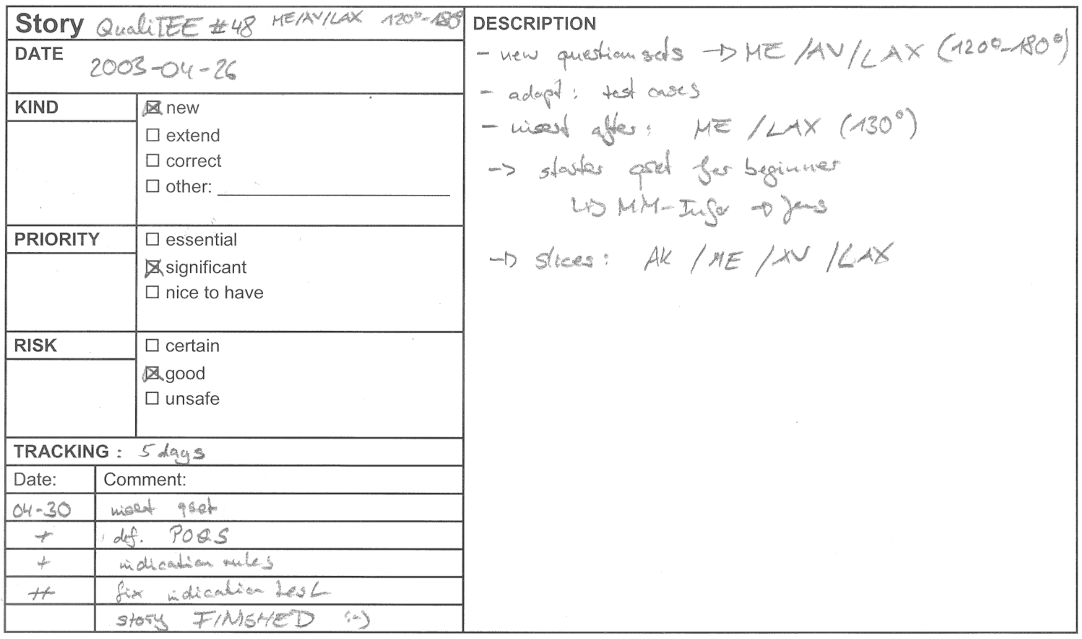
As explained above the global system metaphor describes the basic aim of the designated diagnostic knowledge system. Of course, the chosen metaphor can change during the development phase. For example, starting with the idea of a consultation system it comes up, that formalizing explicit knowledge for inferring final diagnoses is too time-consuming and costly. To reduce knowledge acquisition costs the team decides to formalize only coarse diagnoses and attach an information center to the already implemented dialog knowledge.

2.2.2. Planning Game

In summary the overall aim of the planning game is to maximize the value of the built system and to minimize development costs.

The value of the system is mainly defined by the consumer satisfaction derived from the usability, the functionality and the correctness of the system.

Purpose of the Planning Game The main purpose of the planning game is to decide about the scope and priority of future development. Furthermore, the costs of the planned implementation are estimated and the selected plans are scheduled. It is also useful to provide a benchmark for feedback in order to enable adaptation of plan estimation in the future.



Story QualiTEE #48 ME/AV/LAX 120°-180°

DATE 2003-04-26

KIND [✓ new] ☐ extend ☐ correct ☐ other: ______

PRIORITY ☐ essential [✓ significant] ☐ nice to have

RISK ☐ certain [✓ good] ☐ unsafe

TRACKING : 5 days

Date:	Comment:
04-30	insert qset
+	def. POQS
+	indication rules
++	fix indication test
	story FINISHED :-)

DESCRIPTION

- new question sets → ME /AV/LAX (120°-180°)
- adapt: test cases
- insert after: ME /LAX (130°)
→ starter qset for beginner
↳ MM-Info → Jens
→ slices: AK /ME /AV /LAX

Figure 2.2: An exemplary story card.

In principle it is not advisable to make long-range plans, because the priorities of designated functionality can change or customer requirements can (and will) evolve during the project. Thus, it is reasonable to discuss long-range plans only as coarse artifacts and concentrate on detailed plans to be realized in the near future.

Plans are documented by *story cards*. Story cards are recorded by the developer and the costumer. They contain information about, e.g., the recording date, the estimated implementation time, a task description, and additional notes. Story cards are useful to document the overall development process of the knowledge system project. In Figure 2.2 an exemplary story card is shown. It is worth noticing, that plans do not only carry out new functionality of the knowledge system, but also may consist of extending an already implemented functionality or correcting a broken part of the knowledge system. The planning game is divided into three phases, called *planning moves*: Exploration, commitment and steering.

Exploration In the *exploration phase* the costumer and the developer consider the system functionality that should be changed or added. For each functionality a story is written down to a story card describing the task in more detail. Then, the development costs of each story are estimated and stories are partitioned into smaller grained stories, if necessary.

Commitment In the *commitment phase* the costumer and the developer decide about the realization of the stories recorded in the exploration phase.

The customer can assign a value to each story (i.e., the priority) and the developer sorts the stories independent of this by their risks. The possible values for the priority and the risk of a story are given in the following table:

Priority		**Risk**	
essential	The system will not work without implementing this story.	*certain*	The implementation costs of the story can be estimated precisely.
significant	The story will add a significant value (i.e., functionality) to the system.	*good*	The implementation costs of the story can be estimated reasonably well.
nice to have	The story will add not a necessary but a useful feature to the system.	*unsafe*	There is no estimation possible about the implementation costs of the story.

Based on the estimations made above the developers and the costumer define the next release by picking story cards. They define the scope of the release by ordering the collected cards and defining a release deadline according to the risk estimations made before.

Steering Of course, the implementation of the plans develops not always as expected. Therefore, the *steering phase* updates and refines the plan currently under implementation. The steering phase offers three moves:

Steering moves	
Iteration	During the implementation of the plan stories are picked iteratively, i.e., after successful implementation of a story the most valuable remaining story is chosen to be implemented next.
Recovery	It can happen that the developers overestimate the development speed. This occurs sometimes at the beginning of a project, when the development process has not been well-established. Then, after adaptation of the plan estimations, the costumer and the developer need to decide about the most valuable set of stories, that should remain in the current plan and that can be moved to a later planning game.
Reestimate	In general, during the implementation of the plan the developers can reestimate the remaining stories, if the plan no longer provides a sufficiently precise table of the development.

We have seen that the planning game provides a flexible method for guiding the development process of knowledge systems. Plans are documented in story cards and deliver a structured sequence of the development process, in which the costumer as well as the developer is integrated. The steering phase enables the process to flexibly adapt the currently implemented plan by inventing new stories, by reestimating the predicted plan costs or by reordering the priority of the remaining unimplemented stories.

Furthermore, by providing methods for estimating and documenting implementation costs (derived from the implementation velocity), an accurate feedback can be given to assess the whole development process.

Example With the following example we want to illustrate how the planning game can be applied when developing diagnostic knowledge systems. For a running system the costumer comes up with the requirement to include a problem area (represented by a set of diagnoses), which should be detected by the knowledge system. Furthermore, the costumer wants to have multimedia information (e.g., hyper-linked texts with figures) to be included for several diagnoses.

The planning game starts with the exploration phase, i.e., the requirements stated above are collected and written on story cards. It is obvious, that the costumer has to clarify his requirements in this phase, when writing a story card for the new diagnoses and for the distinct multimedia extensions.

Having finished the exploration phase, the developer and the costumer estimate the risk and the value for each story to bring them into a useful order. In this example, the value of the multimedia extensions of several diagnoses is assumed to be less valuable than the inferential knowledge of the most new diagnoses.

After the first iteration (e.g., implementation of the most valuable story) the costumer requests a new, very important diagnosis to be inserted into the system. A new story is defined for this diagnosis and risk and value are estimated. Because of the specified time-table, the costumer and the developer decide about removing a low-priority story from the plan. This procedure ensures that the project keeps in the intended time. With the completion of the plan the estimations concerning costs can be assessed by the experiences made during the implementation. The feedback gained from this assessment can help the developer and the costumer to improve their planning and estimation capabilities.

2.2.3. Implementation

Once the planning game has arrived at a sequence of stories, the implementation phase is initiated. Then, each story is implemented in the determined order given by the elaborated plan. Actually, implementation is done during the steering phase, so that new stories can be invented because of new requirements of the costumer.

Splitting Stories into Tasks For the implementation of a story the developer split the story into distinct tasks, which are sequentially handled. A *task* is defined as separable part of a story, which can be easily implemented and tested. If the story has a large number of tasks, then a planning game concerning the tasks may be reasonable. However, a large list of tasks can also indicate the necessity of partitioning the story into several (sub-)stories.

Implementation Phases When we talk about implementation of stories we follow a strict *test-first* approach: Each implementation of a task passes a test-implementation phase and a code-implementation phase.

In the *test-implementation phase* the developer assigns test knowledge to the knowledge base, which describes the expected behavior of the new task. Test knowledge needs to be automatically executable in order to instantly validate the knowledge system. In the *code-implementation phase* the developer actually acquires and implements the new functionality described in the task.

Following this test-first approach the implemented tests will accumulate to a suite of tests, which can be executed as a whole. This suite can be executed anytime to verify the already implemented functionality of the knowledge system.

After the code-implementation phase has been finished, the knowledge system is validated using the test suite. If the tests report no errors, then the implementation of the task is complete and the next task is considered for implementation. If some of the tests fail, then we have to enter a debugging task. There are many reasons that can cause a test to fail. The most typical ones are implementation errors in the new implementation and the missing adaptation of old tests. In the later case, old tests fail because the new implementation has (possibly accidently) changed an existing functionality. Then, either the new functionality needs to be adapted or the old tests need to be adapted according to the new functionality.

Significance of Tests At first sight, the construction of tests for each task is an additional and huge effort during the implementation phase. Nevertheless, implementing tests besides the actual functionality is good for the following reasons:

- *Validation of the code implementation*
  Tests are primarily defined to validate the subsequent code-implementation. If the tests pass, then the developer and the customer feel confident, that the newly implemented functionality shows the expected behavior.

- *Removing communication errors*
  Tests are often implemented as examples of typical system runs. Defining such examples together with the costumer (which has to know the typical system behavior) will clarify story or task definitions. It is worth noticing, that often ambiguous requirements are timely exposed due to the test-implementation phase.

- *Detecting side effects*
  Since all tests are collected in a common test suite, all available tests will be executed at least before completing the implementation of a story. In this way, side effects can easily be discovered, i.e., if a new functionality has accidentally changed the behavior of a previously implemented and still remaining functionality.

Example We illustrate the considerations made above by a simple example: During the implementation of a documentation system a new story comes up for introducing a new question set containing additional questions. The story tells us, that the question set should be indicated, i.e., asked to the user, after an already implemented question set. According to the process model, we start with the test implementation and we define some typical system dialogs containing the new question set. Then, we start with the code implementation and insert the question set and the contained questions accordingly. Before completing the implementation of the story the test suite is executed now additionally containing our new test. Several old tests fail, since they expect a different sequence of question sets to be indicated not containing the new question set. In this case, the old tests need to be adapted according to the new question set, i.e., we modify the old tests so that they also consider the new question set. After the adaptation the test suite passes and the implementation of the story can be seen as completed. Another example is the definition of tests for inferential knowledge. Then, we can build tests that postulate the derivation (or exclusion) of a given set of diagnoses for a given set of findings. It is obvious, that these tests can fail, if new inferential knowledge is inserted.

2.2.4. Integration

In the *integration* step the newly implemented functionality is embedded into the production version of the knowledge system. Integration is done by putting the current knowledge system on an integration unit (e.g., a distinct computer or storage device), and by running a suite of integration tests. The integration is finished when the integration test suite passes.

Continuous Integration Integration is done continuously, which means that every successfully implemented story is immediately integrated into the production system. Continuous integration guarantees an always up to date knowledge system containing all the functionality, which has been implemented to this date. Thus, we receive a fully running system after each integration at the integration unit.

Integration Testing To guarantee the practicability of the integration builds the test suit ensures that the system is not broken because of the new functionality. For a reasonable integration additional tests need to be available, which are applied during integration to check more aspects of the functional behavior of the knowledge system. We call these tests *integration tests*. Integration tests often contain a larger number of previously solved cases, that can be run against the knowledge system. These previously solved cases typically contain a set of findings and a set of expected diagnoses for these findings, but sometimes also strategic knowledge is available. Running a number of thousands of cases can take several minutes or even hours. Therefore, it is not practical to include them into the working test suite, since the suite is applied many times during the implementation of a story. Nevertheless, before the integration of a new version these integration tests are an essential indicator for the correct behavior of the knowledge system.

Errors during Integration Of course, it can happen that the integration candidate has passed the working test suite, but the integration tests fail. In this case, we stop integration immediately. We invent a new story containing a debugging task in order to find the reason for the incorrect performance of the system. Once the error has been found, it is advisable to create tests covering the defective parts and include them in the working test suite.

2.3. Story Idioms

We have seen, that the definition of stories is essential for the implementation of the agile process model. For the agile development of diagnostic knowledge systems we can refer to a set of typical stories (i.e., story idioms), which are briefly described in an abstract manner in the following.

New Problem Area The *New Problem Area* idiom considers the introduction of a new set of diagnoses, which together can be grouped to a problem area. Then, usually the following tasks have be accomplished: 1) Define appropriate test knowledge. 2) Insert new diagnoses of the problem area. 3) Extend the finding hierarchy by the required new findings for the problem area. 4) Insert associated inferential and strategic knowledge. 5) Update existing tests with respect to the new problem area.

New Question Set The insertion of a new group of questions into the knowledge base can be described by the *New Question Set* idiom. Then, we need to consider the following tasks: 1) Select the position of the new question set in the question hierarchy. 2) Define the type and the level of the questions contained in the new question set. 3) Insert the question set with the associated questions. 4) Possibly adapt inferential and strategic knowledge with respect to the new question set. 5) Update existing tests.

Increase Clarity of Question The *Increase Clarity of Question* idiom should be applied, if, e.g., a question often is falsely answered or users have problems answering the question. The clarity of a question can be increased by one of the following tasks: 1) Reduce or increase answer range of the question (for choice questions). 2) Attach informal support knowledge to the question explaining its meaning. 3) Divide the question into a set of smaller and simpler questions. 4) Change the type of question, e.g., translate a numerical question to a qualitative one-choice question.

Increase Dialog Velocity If the questionary contains too much questions or the dialog is too time-consuming for the user, then the *Increase Dialog Velocity* idiom should be considered. The application of at least one of the following tasks can increase the dialog velocity: 1) Add control question, in order to avoid useless questions (only meaningful, if control question can reduce useless question significantly). 2) Simplify questions, e.g., combine associated yes/no questions to multiple-choice questions or shrink the value range of questions. 3) Link the dialog with a data base system, which can auto-answer a set of questions. 4) Coarsen diagnosis detail with corresponding reduction of a set of required questions. 5) Add free text questions. 6) Use more technical questions in dialog (that were possibly used as abstracted questions before). 7) Omit questions or answer alternatives with little diagnostic importance (similar to 5). 8) Consider the complete redesign of the implemented strategic knowledge.

Increase Diagnostic Accuracy If the system does not supply a correct diagnosis for all cases, then the *Increase Diagnostic Accuracy* idiom can improve the diagnostic behavior of the system by at least one of the following tasks: 1) Refine and extend the diagnostic knowledge. 2) Refine the diagnostic hierarchy by, e.g., adding more detailed diagnoses with associated knowledge. 3) Refine the question structure to enable the system for a more detailed acquisition of the problem description and adapt inferential knowledge with respect to the refinement. 4) Coarsen the diagnostic hierarchy, if the currently desired accuracy cannot be reached.

Introduction of Plausibility Checks The correctness of the acquired data is necessary for a correct diagnostic behavior and can be improved by plausibility checks. These checks can be included into an existing system by the following tasks: 1) Introduction of additional, redundant questions with associated plausibility rules comparing the corresponding answers. 2) Define plausibility constraints for corresponding findings (not necessarily redundant).

2.4. Practices of the Agile Process Model

The basic practices of the agile process model are depicted in Figure 2.3. They summarize the presented methodology with its main aspects.

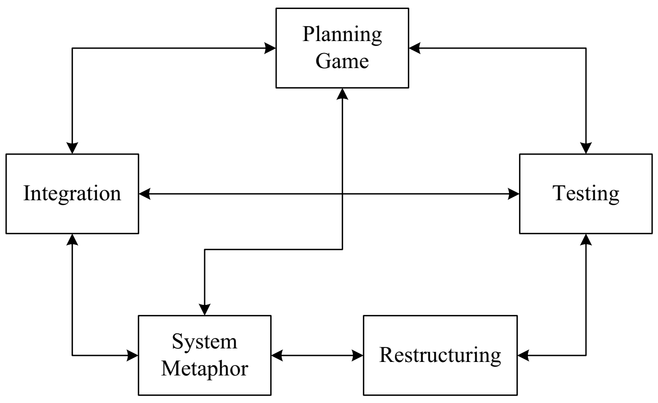


Figure 2.3: The practices of the agile process model.

It is worth noticing, that each practice is necessary for a successful implementation of the agile process model, since there exist strong dependencies (indicated by links in the figure) between the several practices.

2.4.1. Summary of the Practices

The center of the process model is the *system metaphor*, which contains the basic idea of the designated system. A system metaphor has to be chosen at the beginning of the project and determines the system's focus and denotation. For developing a diagnostic knowledge system there exists a predefined set of global and local system metaphors, which should be extended or changed, if required. During the development *planning games* are executed iteratively by writing story cards and determining a schedule for the plan. A planning game narrows the project course and also contains estimations about the project velocity and an assessment of previously estimated plans. *Testing* is a central practice of the agile process model, since they enable an evolutionary design of the intended knowledge system. Tests have to be written in advance to refine the implementation specification of a plan. During the development tests accumulate to a test suite, which covers the intended functionality of the system and which enable for refactoring and integration. Since the design of the modeled knowledge evolves during the project, *restructurings* have to be implemented from time to time in order to improve the knowledge design and clarity. Restructurings are feasible due to the existence of a test suite. A specific characteristic of the agile process model is the presence of an always running system. This can be guaranteed by a continuous *integration* after the successful implementation of any planning game. Any integration build corresponds to a part of the system metaphor and is done by running (possibly additional) test suites.

2.4.2. Comparison with XP

As mentioned in the introduction of this chapter the presented process model is an adaption of XP, a successful software engineering methodology. In contrast of coding general software by an all-purpose programming language the agile process model considers role-limiting knowledge as the language to be used for developing the system. Usually, a highly integrated knowledge system shell is used to acquire, model and maintain the knowledge. Such a shell is comparable to integrated development environments (IDE) for generic programming languages. However, there are further practices known from XP which are not explicitly discussed by the presented agile process model.

On-Site Customer In an XP project a real customer must accompany the development team in order to answer upcoming questions or to decide about small-scaled priorities. For the development of a knowledge system the customer is represented by the domain expert, which decides about the focus and functionality of the system. Since we propose the domain expert to model the knowledge by himself – supported by appropriate visual development tools – we already have the customer always on-site.

Pair Programming One of the most famous practices of XP is pair programming, which means that all production code is written by two people sitting on the same computer. Experience has shown that pair programming yields better code written in shorter time. While one pair partner is coding on the keyboard, the other parter can prepare tests, correct the partner's oversights, and think strategically about the next iterations. From our experience knowledge systems are usually not build by pair programming. Often only one expert is available to provide the knowledge. For projects with more than one experts, it is often usual that the experts coordinate basic principles and milestones of the knowledge to be implemented. Then, the knowledge is implemented by oneself and after that the experts meet again for a review of the already modeled knowledge. However, a kind of pair programming is usually performed at the beginning of a knowledge system project, when the expert together with a knowledge engineer decide about the basic functionality and design of the system. The knowledge engineer gives advices on formalizing and structuring the knowledge to be implemented.

Collective Ownership In XP every programmer takes the responsibility for the whole code. If somebody changes functionality, adds new one or tries to improve existing code, then he has to assure, that the complete code has not been broken due to his change (which is supported by test suites). In knowledge system projects the collective ownership is a problematic issue. For example, the reputation of medical knowledge systems mainly corresponds with the reputation of the implementing expert. Thus, collective ownership is often undesired and therefore mostly not performed.

2.5. Discussion and Comparison

The systematic development of knowledge systems has been investigated for over 20 years. In the following, we give a short historical overview of knowledge engineering research.

In the late 70s and early 80s the task of building knowledge systems has been understood as a *knowledge transfer* task. This task was interpreted as just collecting and implementing knowledge, which was thought to be already available ”in the heads” of the domain experts:

> *This transfer and transformation of problem-solving expertise from a knowledge source to a program is the heart of the expert system development process.* ([54], p. 23)

Typically, these systems were implemented using simple production rule formalisms, which were easy to understand and supported the acquisition process during expert interviews.

Experiences with numerous systems – prototypical but also productional ones – have shown, that several problems arise when following the knowledge transfer approach: Due to the uniform use of production rules as knowledge representation structural knowledge was mixed up with other kinds of knowledge, e.g., strategic knowledge [31]. Besides the fact, that this representation is not suitable for all kinds of tasks, the maintenance and acquisition of such systems became very complex and the complexity growed drastically with the size of the represented knowledge. Because of the experiences and problems with the knowledge transfer approach, research shifted to a *knowledge modeling* approach. Now developing a knowledge system was seen as the task of building a computer model, which shows problem-solving capabilities comparable to a human domain expert.

Though it was not intended to develop an exact copy of the expert’s cognitive abilities, the artifact, i.e., the system, should be able to infer comparable results for a given problem within a focussed domain. Thus, the knowledge appeared to be not always accessible and was acquired, structured and build up within a continuous process. This *model construction process* can be seen a significant progression of the former transfer process.

In the following, we describe a selection of process models for developing knowledge systems. All these approaches have one significant difference compared to the presented agile process model: Whereas the agile process model is designed for the usage with configurable role-limiting methods the most of the following process models are generic models supporting the implementation of the problem-solving capabilities as well. For the discussion we will only consider the development of the knowledge base and not the development of the problem-solving method.

For a detailed survey of knowledge engineering research and knowledge engineering methodologies, respectively, we refer to [90, 130, 131].

2.5.1. Classical Knowledge Engineering Approaches

Since the beginning of knowledge engineering research the work has been oriented on the concepts developed by software engineering.

In this way, it is not surprising that early knowledge engineering process models adopted process models known for developing generic software systems.

Stage-Based Process Model

The *stage-based process model* [33] defines a life-cycle for developing a knowledge system, which is very similar to the well known waterfall model [124].

The stage-based model is depicted in Figure 2.4.

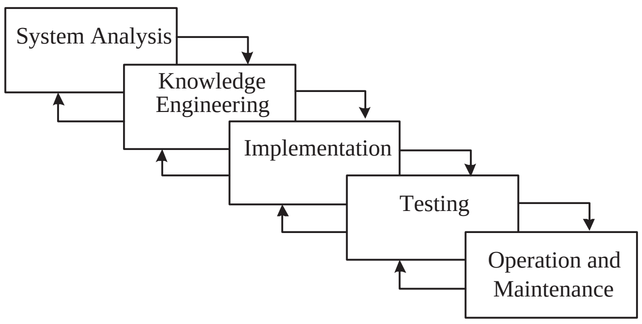


Figure 2.4: The phases of the stage-based model.

We briefly discuss the phases of the process model in the following:

System Analysis The organizational environment of the intended system is analyzed and described. Thus, the involved hardware, software, people and additional systems are determined. Furthermore, the analysis defines the tasks that are affected and substituted by the system, and which tasks are not influenced by the system.

Knowledge Engineering In this phase the necessary knowledge is acquired, structured, represented, and specified.

Implementation According to the specification in the previous phase the knowledge system is implemented. The implementation strongly depends on the underlying tools used for the implementation. In earlier days, often AI languages like LISP [41] or Prolog [73] were applied. Nowadays, for the implementation often visual development environments are used, e.g., HUGIN [4], PROTÉGÉ [84, 50] or D3 [104].

Testing In this phase the implemented system is tested for the presence of errors. It is well agreed, that the absence of errors cannot be shown. Therefore, only a sufficient large number of test cases covering the whole functionality of the system can give a strong confidence into the validity of the system.

Operation and Maintenance After the system has been delivered it goes into operation. In order to keep the system running, maintenance may be necessary. One can distinguish between corrective, adaptive and perfective maintenance. Maintenance will be discussed in more detail in Section 3.2.

The presented stage-based model has the following disadvantages:

- The process model relies on the assumption that sufficient detailed requirement specification can be prepared in advance. This is often not possible due to the uncertainty of the costumer regarding the actual requirements of the project, or due to the uncertainty of the available knowledge.
- The customer is not involved during a long period of the project development. Since he is excluded from the implementation phase no feedback can be given, e.g., corrections of misunderstandings or specification adaptations. However, if specification errors are recognized late, then correcting the consequences is often very costly.

This criticism has led to a more interactive and evolutionary process model, i.e., the incremental process model.

Incremental Process Model

The incremental process model [33] adopts the characteristics of the rapid prototyping approach [26], which has been accepted to be a suitable model for software projects with uncertain requirements specifications. The incremental process model is depicted in Figure 2.5 The main idea of the process model is rapidly building a small but working system, which covers a limited part of the intended problem domain. This initial system is often called *pilot* or *prototype*. In the following steps this pilot is incrementally extended by further parts of the problem domain, which have not been considered so far. During these iterations a strong interaction between the user, the developer, and the domain expert is postulated (that can be partly identical).

A prominent system, which has been implemented according to the incremental process model is MYCIN [24]. In the following we discuss problems arising with the incremental process model:

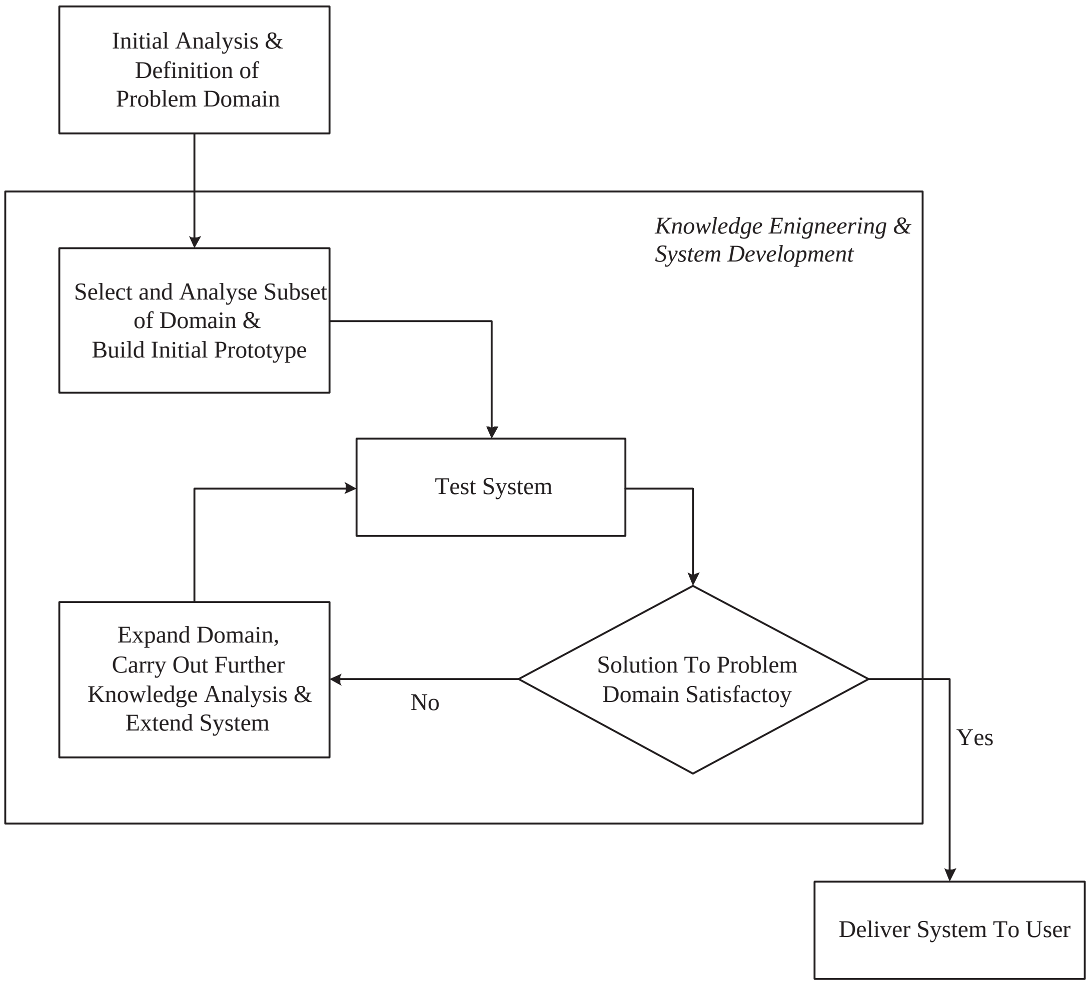

Initial Analysis & Definition of Problem Domain
Knowledge Enigneering & System Development
Select and Analyse Subset of Domain & Build Initial Prototype
Test System
Expand Domain, Carry Out Further Knowledge Analysis & Extend System
Solution To Problem Domain Satisfactoy
No
Yes
Deliver System To User

Figure 2.5: The phases of the incremental process model.

- *Strong interaction*
  The involvement of the user and the domain expert is necessary for the whole project. This may not be possible or feasible for all projects.
- *Unstructured procedure*
  The evolutionary increments will evolve to an unstructured and chaotic system and knowledge design.
- *Hard to estimate*
  Due to the uncontrolled project flow it becomes difficult for the management level to prepare cost estimations or to define project milestones. However, for larger projects the possibility of creating milestones and cost estimations is a necessary project requirement.

We can see that the incremental process model is similar to the presented agile process model. Therefore, we now will discuss the problems of the incremental process model compared to the agile process model:

The *strong interaction* requirement also holds for the agile process model, i.e., the costumer and the domain expert need to interact during the hole project. However, this requirement may not be a disadvantage but a desirable characteristic, if analysis and specification of the intended system is not possible in advance. Experience has shown that we often face this situation during knowledge system projects.

The *unstructured procedure* problem describes a serious issue often recognized for prototypical process models. In fact, it has been the reason for the bad reputation of prototypical approaches. Nevertheless, this situation has been also observed for software engineering approaches. To cope with this problem *refactoring* methods [87, 42] have been developed for object-oriented approaches. Refactoring methods describe structured procedures to change the design of existing software without changing the underlying semantics. In [74] it was shown that these ideas can easily be adopted for refactoring diagnostic knowledge and we presented a selection of typical methods. The agile process model defines refactoring as a special kind of the implementation step (i.e., the implementation of a re-design in contrast to the implementation of new functionality). In this way, the agile process model tackles this problem by offering structured methods. This issue is described in more detail in Section 3.2.2 (p. 40) and Appendix A.

The remaining problem *hard to estimate* is also considered by the agile process model: During the planning game the costs of each plan are estimated by priority and risk. Since feedback about the estimation accuracy after each implementation of a plan is demanded by the process model, the estimation skills are continuously improved during the project. This will result in more precise cost estimations and a pleased management level.

2.5.2. CommonKADS

The most prominent process model for developing knowledge systems is KADS [118] and its successor CommonKADS [119].

In Schreiber et al. [117] CommonKADS is defined as a project management approach, which is more flexible than a stage-based approach and more structural than rapid prototyping.

For this, CommonKADS delivers a *model suite*, which consists of distinct models each covering a specific facet of the project. Whereas each model considers a limited aspect of the project, all models together provide a comprehensive view of the complete project.

The CommonKADS model suite knows the following models:

Organizational model The organizational model deals with the analysis of the participating organization in order to discuss the applicability of a knowledge system, i.e., its kind of application, its feasibility, and its impacts on the organization.

Task model The task model provides a more detailed analysis of the organizational unit, in which the knowledge system is intended to be applied. The analysis covers input and output of the focussed knowledge system, possible preconditions and performance assumptions. Furthermore, the analysis also covers required resources and competencies for the development and use of the knowledge system.

Agent model Agents are an abstract definition for something that performs a task. Thus, agents can be humans or artifacts like databases or information systems. The agent model provides an analysis of the agents involved in the project, lists their competencies, constraints, and possible communications interactions among each other.

Knowledge model The knowledge model contains an implementation-independent description of the types and structures of the required knowledge. Due to its conceptual description the knowledge model is understandable by humans and therefore is an appropriate communication media between the user, the domain expert, and the developer.

Communication model The communication model specifies the communicative transactions between the agents in a conceptual and implementation-independent way. Then, the type of information which is exchanged between the agents is defined.

Design model The design model specifies a detailed design of the intended application, i.e., the knowledge system. For example, it contains the architecture, platform specification, and applied software.

We can summarize the purpose of each model with the following: The design model is defined according to the requirements specification resulting from the previous models. The organizational model, task model, and agent model analyze and specify the organizational structure in which the knowledge system should be applied. Furthermore, this analysis is important to determine the feasibility of the system. Based on these models the communication model and knowledge model deliver the problem-solving capabilities with the corresponding input and output behavior.
In Figure 2.6 the models are depicted with the interactions among each other. It is

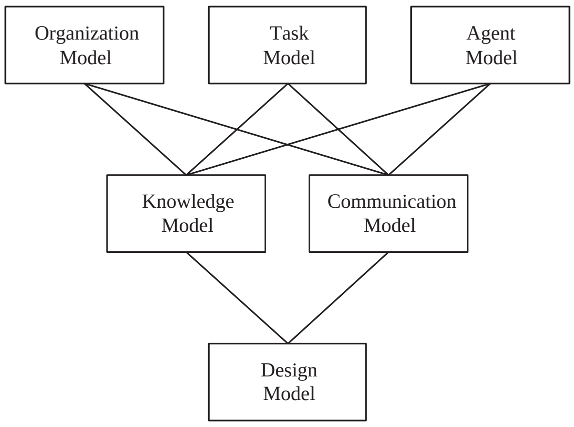


Figure 2.6: The model suite of CommonKADS.

worth noticing, that not all models need to be constructed during a knowledge system project. The detail of implementation of each model strongly depends on the goals of the project and the degree of experience with earlier projects and the current project. Then, the construction of the knowledge system is carried out by a stepwise and cyclic construction of the models. In summary, in addition to the implemented knowledge system the

CommonKADS process model also delivers documents prepared according to the several models.

Although CommonKADS has been accepted as the de-facto standard it is not appropriate for all kinds of knowledge system projects: CommonKADS can deliver a wide range of documents and often a lot of analysis and specification effort need to be done before the first implementation starts. This situation often has been experienced as frustrating for the experts. Especially, when domain experts are not fully working on the project and/or the project has only a small size (e.g., 2-3 people working on it), then this organizational overhead is often considered as needless. Consequently, dissatisfied expert can threaten the success of the whole project. Contrary to this, a rapid implementation of a pilot, facilitated by the agile process model, will motivate experts to proceed with their work and extend the knowledge system by further knowledge or functionality. Nevertheless, for large and costly projects, demanding a highly structured procedure, CommonKADS is the state-of-art process model for developing knowledge systems.

2.5.3. MIKE

MIKE [6] (Model-based and Incremental Knowledge Engineering) tries to combine prototyping with (semi-)formal specification techniques. It differs from CommonKADS by providing a framework for incrementally developing knowledge systems in a reversible process. The MIKE process model is a spiral model consisting of the following steps:

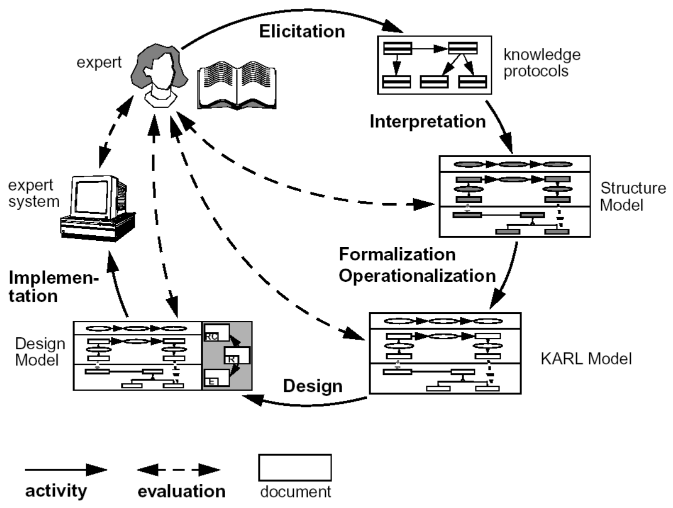


Figure 2.7: The MIKE process model with its phases and corresponding documents.

Elicitation In this phase informal knowledge about the domain is being gathered and the corresponding problem-solving task is elicited by, e.g., structured interviews. The elicitation step produces so-called *knowledge protocols*.

Interpretation In the interpretation phase the knowledge protocols are structured: Informal content is transferred into a semi-formal *structure model*, which is expressed in a restricted language. However, though the coarse structural elements are fixed, the basic blocks of the language (e.g., describing inference mechanisms) usually contain free text. Thus, the structure model is an appropriate communication media between the user, the developer, and the domain expert.

Formalization/Operationalization Based on the interpretation phase a formal model is generated containing the concepts stated in the structure model. Here the executable language KARL [40] is applied. The resulting KARL model is comparable to the knowledge model known from CommonKADS but is directly executable.

Design In addition to the KARL model defined in the Formalization/Operationalization phase further requirements are acquired in the design phase. These functional requirements can consist of efficiency considerations, remarks on the maintainability or non-functional software and hardware specifications. The result of this phase is the *design model* which is formulated in DesignKARL [65], an extension of KARL. In DesignKARL it is possible to additionally structure the available KARL model and to specify the used data types and algorithms.

Implementation In the last phase of the cyclic process the design model is implemented in the postulated hardware and software configuration. The implementation produces a deliverable knowledge system which needs to be evaluated in the target environment. In a next pass of the cyclic process the knowledge system may be corrected, modified or extended according to the results of the preceding evaluation.

In Figure 2.7 the MIKE process model is depicted with its phases and corresponding documents. Compared with CommonKADS and the presented agile process model the MIKE process model provides a balance between document-centered and rapid development approach. Similar to the agile process model a deliverable system is always produced after each iteration of a spiral development cycle. On the other hand, MIKE emphasizes a more structured design process than it is required by the agile process model.

2.6. Conclusion

In this chapter, we have introduced an agile process model for developing diagnostic knowledge systems. We have seen that the presented process model adapts the general ideas of the eXtreme programming methodology, which has been proven to be a successful method for small and mid-size projects in a rapidly changing environment. We have motivated why the agile process model fits into the problem of developing diagnostic knowledge systems, and we introduced the methods and tasks for applying the adapted process model. We concluded the chapter with a comparison of the agile progress model with related approaches, e.g., prototyping, stage-based or incremental process models.

Part II.

Containers of a Diagnostic Knowledge System

3. Agile Development using Knowledge Containers

3.1. The Architecture of Diagnostic Knowledge Systems

In general, the architecture of a diagnostic knowledge system is described by the components *knowledge, inference,* and *user interface*. For the development and maintenance of such a system, i.e., editing the knowledge, the additional component *knowledge modeling environment* is required. In the literature, similar architectures are described, e.g., Puppe [99, p. 16], and Stefik [128, p. 296].

Figure 3.1 depicts the proposed architecture. We briefly discuss each component in the following.

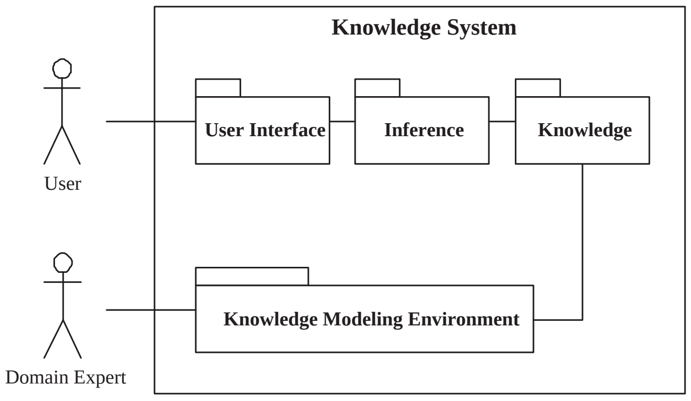


Figure 3.1: The architecture of a diagnostic knowledge system.

The *user interface* provides a uniform access to the diagnostic knowledge system: The user can start a new consultation, enter findings, and obtain diagnoses, i.e., a solution for the stated problem. Furthermore, the user should be able to ask for a justification explaining the retrieved diagnoses. The user interface delegates newly entered findings and user requests to the kernel of the knowledge system, i.e., the *inference* component. The inference component processes the findings and requests by using particular domain knowledge contained in the connected knowledge component. The inference component should not be envisioned as a monolithic problem-solver (e.g., a single rule interpreter), but is designed as a mediator of problem-solvers and controllers, which allows for the processing of various kinds of knowledge and information.

The *knowledge* component embodies the knowledge that is used by the knowledge system, i.e., the knowledge base. There are miscellaneous facets of knowledge contained in the component: The knowledge base can define the ontology of the underlying problem domain (ontological knowledge); it can be used to infer diagnoses for a given set of findings (structural inference knowledge); it can be responsible for controlling a guided questionary presented to the user (strategic knowledge); or it might consist of additional informal content describing findings or diagnoses in more detail.

According to Clancey [31] it is reasonable to classify these different kinds of knowledge into distinct parts. In our work we call these parts *knowledge containers*. Clancey introduced the terms structural knowledge, strategy knowledge and support knowledge, which we analogously apply. In the proposed framework we further separate ontological knowledge from structural knowledge, since ontological knowledge is also applied in the remaining knowledge containers. Furthermore, we do not restrict our framework to a rule-based representation of knowledge, but we also discuss the usage of case-based, model-based, and other facets of knowledge.

Besides the general architectural issues of defining a knowledge system the construction and maintenance of such systems is an important aspect to consider. Therefore, we first describe maintenance of knowledge systems from a traditional viewpoint and then present knowledge maintenance as an activity of the agile process model. The Chapters 4-7 describe the knowledge containers in more detail and introduce possible representations that are applicable to the particular knowledge containers.

3.2. Maintenance of Diagnostic Knowledge Systems

Maintenance of diagnostic knowledge systems is the process of modifying existing knowledge. After giving an overview of the traditional categorization of maintenance we define maintenance in the context of the agile process model.

3.2.1. Traditional Definition of Maintenance

Traditional process models for software engineering and knowledge engineering define maintenance as a task, which is executed *after* the project has been finished and the application has been already deployed into routine use. The IEEE glossary [59] defines:

> **Maintenance:** The process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment.

Then, the running system is modified in order to correct observed malfunction, to implement additional functionality resulting from changed user requirements, or to adapt the software to changed environments. Swanson [132] classified the term maintenance of general software according to the desired aims into *corrective maintenance*, *perfective maintenance*, and *adaptive maintenance*.

Coenen and Bench-Capon [33] adopted this classification for the knowledge engineering perspective. In the following, we briefly describe the three terms with respect to the knowledge engineering task according to Coenen and Bench-Capon.

Corrective Maintenance Complex knowledge systems often show faulty behavior in routine use. The process of eliminating this malfunction is called corrective maintenance. According to Sommerville [124, p. 660] errors can occur as coding errors, design errors or requirement errors. For example, for a correct input, i.e., set of observations, a false output, i.e., set of diagnoses, is retrieved. In many cases, structural knowledge needs to be adjusted or additional knowledge needs to be inserted. However, corrective maintenance can be also considered for other knowledge containers, e.g., for ontological knowledge (incompletely defined objects) or support knowledge (objects with false linkage).

Perfective Maintenance It is very common that additional user requirements appear after the knowledge system has been put into routine use. The experiences made with the system cause previously recorded requirements to become less important, and new requirements are demanded, e.g., to improve the usability of the system. Then, perfective maintenance methods are applied to integrate new user requirements into the already running system.
Typically, perfective maintenance becomes necessary when a new diagnosis should be included into the system, e.g., if a knowledge base needs to be refined by subdiagnoses. Another example for perfective maintenance is the extension of the system by multimedia support knowledge, since experience has shown that additional help would be useful for the user of the system. Perfective maintenance on strategic knowledge can be accomplished by reducing the duration of a typical dialog.

Adaptive Maintenance Knowledge about the domain can change during the development of the system. If revised knowledge becomes available during the routine use of the system, then adaptive maintenance should be applied. For example, in a medical application new guidelines for performing investigations or therapy actions may be established. Then, the knowledge system need to by adjusted with respect to this change.

For traditional software engineering projects, there exist studies about the frequency of the applied maintenance activities. For example, Sommerville [124, p. 661] reports the proportions given in Figure 3.2.

Task	Frequency
corrective maintenance	17%
perfective maintenance	65%
adaptive maintenance	18%

Figure 3.2: Proportions of the implementation of maintenance tasks in traditional software engineering projects according to Sommerville [124].

Unfortunately, no studies for knowledge system projects can be found in the literature. For traditional, heavy-weight process models comparable numbers may be obtained. Since the agile process model introduced in Chapter 2 relies on continuous maintenance during the development process the numbers will differ significantly. For example, in Figure 3.2 we can see that corrective maintenance has only a small fraction of the overall maintenance costs. This results from the fact that only corrections are counted that are implemented after the system has been delivered. However, for the agile process model corrective maintenance is done continuously during the project life-cycle, while corrections during development are also counted. Therefore, it is reasonable that corrective maintenance will play a more important role.

However, we will see that this classical three-fold view of knowledge maintenance is not appropriate for the presented agile process model. Since maintenance is done continuously during the knowledge base development, there are further issues to consider. In the next section, we therefore introduce maintenance as an agile implementation activity.

3.2.2. Maintenance as an Agile Implementation Activity

In the agile process model the term maintenance is subsumed by the implementation step, when new knowledge is integrated into the knowledge base or when changing existing knowledge. In the agile implementation step we consider the following maintenance activities besides the manual insertion of new knowledge:

- testing knowledge
- restructuring knowledge
- learning knowledge

At first sight, the classical definition of maintenance can be subsumed by the term *restructuring knowledge*, but also testing and learning knowledge should be seen as a maintenance activity. We now discuss the activities in more detail.

Testing Knowledge

In the agile process model, testing plays a central role during knowledge base development. With testing the knowledge developer can review, if the implemented knowledge yields the expected behavior. Test knowledge is used as meta-knowledge for testing the knowledge system. Tests and test knowledge, respectively, can be characterized by the following items:

- **Self-executable**: Tests need to be self-executable since they are applied during the development process very frequently. A test is self-executable, if it contains the expected result beforehand and an automated interpretation of the generated output is possible. In general, a self-executable test should report its result either as success, warning or error. The success message simply reports the proper execution of the tests without any errors. The warning and error messages are a graded answers of an error found in the investigated knowledge. We call the process of performing self-executable tests *automated testing*.
- **Understandable**: Often test knowledge can be used as documentation of the in-

tended system behavior. Then, the understandability of the test knowledge is important. Furthermore, readable test knowledge greatly simplifies the debugging of the knowledge base, if tests fail during the progressing development.

- **Compact**: For the efficient acquisition of test knowledge its compactness is an important issue to consider. Therefore, test knowledge should be represented as compact as possible in order to reduce its acquisition costs. However, the compactness of test knowledge may be opposing to its understandability.

It is easy to see, that test knowledge heavily depends on the representation of the underlying knowledge.

However, the quality of the knowledge included in one of the knowledge containers can be classified according to the following four criteria:

- **Correctness**: A system is called correct, if it shows the expected behavior. Mostly, correctness is tested for structural knowledge (i.e., correct inference of solutions), but it can also be considered for strategic knowledge (e.g., correct dialog behavior) or ontological knowledge (e.g., correct hierarchical relationships).
- **Anomalies**: An anomaly is a certain part of the knowledge base, which can cause the system to behave irregularly. Examples for anomalies are redundant, cyclic, or ambivalent knowledge. Anomalies can be found in all knowledge containers.
- **Robustness**: The degree of robustness of a knowledge system is defined by its correct behavior with respect to difficult or noisy environments. Robustness can be tested by applying noise to the input or the used knowledge. Robustness mainly affects the structural knowledge container, but also can be considered for strategic knowledge. Robustness of a knowledge system can be increased by a well-elaborated ontology design, by meaningful support knowledge, and by redundancy and plausibility checks.
- **Understandability**: The understandability of the modeled knowledge was only studied a little in the past. However, for the agile development of knowledge systems, the understandability of the working knowledge base is very important. Understandability can be defined for all kinds of knowledge containers.

With the introduction of the particular knowledge containers we also introduce various approaches formalizing test knowledge according to the given criteria.

In knowledge engineering research, validation techniques have been undergoing fruitful research for the last decades. Classical work by Coenen and Bench-Capon [33] was continued, e.g., by Preece [93]. Especially, for the rule-based implementation of knowledge systems an extensive framework was defined by Knauf [61]. Interestingly, this framework does not only describe the evaluation of test knowledge (represented as test cases), but especially its suitable generation.

An interesting issue remaining only partially solved is the maintenance of test knowledge. This problem was formulated by Menzies as the *recursive maintenance problem* [72]. In his article, he argued that test knowledge was introduced to simplify the maintenance of knowledge, but the maintenance of test knowledge also needs to be considered sufficiently, possibly by the use of meta-test knowledge. A partial solution for this problem will be presented by the automated propagation of the restructuring methods also adapting test cases according to the performed changes. Modifications of the knowledge base and the test knowledge with respect to a changing world, i.e., domain expertise, can be resolved by refinement techniques mentioned in the following section.

Restructuring Knowledge

During knowledge base development existing knowledge needs to be restructured from time to time, e.g., for facilitating the insertion of new knowledge or for rearranging the design of the knowledge base. Restructuring methods can be characterized according to the following issues:

- **Targeted knowledge container**: Restructuring methods are executed on knowledge that is part of a specific container. We can distinguish between restructuring methods for ontological, structural, strategic, and support knowledge. However, due to dependencies the restructuring of ontological knowledge mostly implies restructurings of the remaining knowledge containers (e.g., if the value range of a question is modified, then the attached structural knowledge needs to be adapted as well).
- **Preserving knowledge semantics**: Restructuring methods can be classified according to their performed changes on the knowledge base. If the method only changes the design of the modeled knowledge without affecting the semantics, i.e., the reasoning behavior, then we call it a *refactoring* method (analogously to refactoring methods known from software engineering). Refactoring methods are a specialization of restructuring methods, which in general can affect the semantics of the knowledge base.
- **Restructuring complexity**: When executing a restructuring method we can distinguish between different complexities. We call methods *atomic restructurings*, if only one or few knowledge base entities are affected. *Composite restructurings* are assembled from atomic methods, in order to accomplish more complex restructuring tasks. For example, an atomic restructuring method is the displacement of a question from one to another question set in the hierarchy. This atomic method is iteratively applied in a composite method that moves a complete question set to another question set.

In general, the need for restructuring arises by the following fives situations:

- **World changes**: Insert new knowledge into the knowledge base or remove/adjust existing knowledge in order to cope with the changed environment or domain expertise.
- **Increase accuracy**: If the diagnostic accuracy is not satisfactory, then the knowledge base needs to be restructured by adjusting falsely modeled knowledge.
- **Improve understandability**: During the development the size of the knowledge base can grow significantly and the design becomes complex. Then, the understandability of the knowledge base can be improved by restructuring methods that adjust knowledge (e.g., shrinking values ranges) or by removing useless knowledge (e.g., unused rules or diagnoses).
- **Enhance usability**: First versions of the knowledge systems often lack in the usability for the end-user. This may be caused by insufficient support knowledge (e.g., missing explanation texts for questions) or a not well-elaborated strategic container

(i.e., the dialog control). In such cases, restructuring methods can be applied that, e.g., improve the strategic knowledge.

- **Achieve reusability**: In large knowledge systems the reusability of the contained knowledge is an important issue. For example, reusability of modeled knowledge can be obtained by creating finding abstractions, which are used by more than one diagnosis or by several structural knowledge elements.

The application of restructuring methods often implies the change of the available test knowledge, since e.g., the tests strongly depend on the structure of ontological or inferential knowledge. However, the adaptation of test knowledge can be carried out automatically in most cases.

Basically, the execution of a restructuring method passes the following four tasks:

1 *Testing the actual state:* Before a restructuring method is executed the working test suite is applied the existing knowledge base. A restructuring should only be considered, if the knowledge base is in a valid state.

2 *Feasibility test:* Checks, whether the restructuring causes unresolvable conflicts, if executed on the existing knowledge base. The feasibility test of the method is executed for all knowledge containers.

3 *Method implementation:* For all applied knowledge containers the following subtasks are performed:

  3.1 *Conflict resolution:* If the execution of the method causes conflicts, then it default values for the conflict resolution are used. For conflicts not resolvable by defaults the method is executed in interaction with the user. Alternatively, the method needs to be aborted.

  3.2 *Method execution:* If no conflicts are reported, then the method is executed according to the restructuring mechanics. See Appendix A for examples.

4 *Testing restructuring result:* After the restructuring method has been performed, the knowledge base is again tested using the working test suite. A restructuring method is only be considered to be successful, if the restructured knowledge base is again in a valid state.

The introduction of restructuring methods for a step-wise and algorithmic modification of the knowledge base was inspired by refactoring methods introduced for software engineering [87, 42]. The adaptation of existing test knowledge (e.g., unit tests) is a critical issue for the application of refactoring methods, and is mostly done manually. In contrast to software engineering the implementation of restructuring methods for knowledge systems often can propagate their changes to the attached test knowledge, e.g., by modifying the corresponding objects in test cases. However, the refinement of the knowledge base performed by restructuring methods differs from refinement techniques, e.g., described by Boswell and Craw [22] or Knauf et al. [62]. Thus, restructuring 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 restructuring method is currently applied manually, though supported by automated adaptations of attached knowledge. In Appendix A a catalog of selected restructuring methods is presented.

Learning Knowledge

(Semi-)Automatic learning of knowledge can significantly reduce maintenance costs in different ways: When applied at the beginning of a knowledge system project it can be used to produce a rapid prototype, which can be refined in further increments (*prototype learning*). During a project, learning methods can be used to extend the knowledge model by further structural knowledge (*knowledge extension*) or to generate an alternative knowledge model for comparison with the existing manually build model (*knowledge comparison*).

For all three applications of learning knowledge a convenient integration into the knowledge modeling environment needs to be available. Then, we have to consider the following three topics for learning knowledge:

- **Conformance**: The learned knowledge should be consistent with the desired knowledge representation in order to be directly used within the knowledge implementation step.
- **Understandability**: Machine learning methods often focus on the achieved accuracy of the learned knowledge. However, when learning methods are used in addition to manual knowledge acquisition methods, besides the accuracy of the learned knowledge its understandability is of major importance. Therefore, learned knowledge can be classified according to its compactness (e.g., number of learned rules) and simplicity (e.g., complexity of learned rules).
- **Semi-automatic**: In many cases, plain machine learning methods yield moderate results with respect to the understandability and the accuracy. Semi-automatic or knowledge intensive learning methods can improve the learned results by incorporating additional domain knowledge. Thus, semi-automatic methods can improve the results of learning methods, by applying more background knowledge to the learning task.

A semi-automatic method for learning understandable scoring rules is proposed in Atzmueller et al. [10]. Furthermore, a framework for evaluating the understandability of learned rules is proposed. In the next chapters we describe learning methods for the particular knowledge containers in more detail.

3.3. Summary

In this chapter, we presented the key methods for developing diagnostic knowledge systems using the agile process model: Testing knowledge, restructuring knowledge, and learning knowledge. All these activities can be subsumed by the term *agile maintenance*. We introduced the traditional definition of maintenance for software engineering and knowledge engineering, and then discussed them in the context of the agile process model. The key concepts for testing, restructuring, and learning knowledge were summarized as well.

4. The Ontological Knowledge Container

In this chapter, we describe the ontological knowledge container, which is the key knowledge container for all remaining containers. The ontological container needs to be considered for any kind of intended knowledge system, whereas all other containers are optional.

4.1. Classification of Ontologies

The usage of ontologies for the development of knowledge systems has been an issue of fruitful research for many years. However, the term ontology is used in many different meanings. In the following, we try to give a brief overview of the various aspects of ontologies. The term ontology is commonly explained according to Gruber's [51] well-known definition:

> An ontology is an explicit specification of a conceptualization. . . . In such an ontology, definitions associate the names of entities in the universe of discourse (e.g., classes, relations, functions, or other objects) with human-readable text describing what the names are meant to denote, and formal axioms that constrain the interpretation and well-formed use of these terms.

Heijst et al. [133, 134] define ontology more explicitly as a definition of the "vocabulary of the domain and constraints on the use of the terms in the vocabulary". Further, they classify the various kinds of ontologies with respect to the amount and type of structure of the conceptualization, and the subject of conceptualization.

For classifying ontologies based on the amount and type of structure of the conceptualization they identify three categories:

Terminological ontologies consider the definition of technical terms in order to represent knowledge in the domain of discourse, e.g., a lexicon.

Information ontologies are comparable to conceptual schemata of databases, i.e., they specify the record structure of databases (mostly as flat information).

Knowledge modeling ontologies describe the conceptualizations of the structure of the knowledge. They usually embody a deeper and richer structure than information ontologies.

If we consider the subject of conceptualization (i.e., the second kind of characterization), then the following types of ontologies can be identified:

Branch ontologies are similar to terminological ontologies and define standard lexical terms for the domain of discourse, e.g., in medicine the ICD-10 standard. They are designed in order to be reused for neighboring or semantically overlapping applications.

Domain ontologies define specific conceptualizations for a special application area. In contrast to domain knowledge, which contains knowledge about particular states in relation to other states (e.g., increased temperature is a manifestation of inflammation), the domain ontology defines constraints on the structure of domain knowledge expressions (e.g., diseases have findings as manifestations). [1]

Generic ontologies (also core ontologies, meta-ontologies [97]) are comparable to domain ontologies but define more universal concepts (to be reused) and are usually intended to be domain independent. However, sometimes a clear distinction between domain ontology and generic ontology is not possible.

Representation ontologies specify the conceptualization of knowledge representation formalisms. A representation ontology usually provides a framework of primitives, which can be used by domain ontologies or generic ontologies.

Application ontologies are focussed on a specific application and contain all definitions that are necessary to model knowledge for the designated application. Although more focussed on a specific application, application ontologies are often very similar to domain ontologies and no clear distinction can be made.

Within the ONIONS methodology [45] additional categorizations of ontologies can be found:

Intermediate ontologies define the general concepts and relations with respect to a specified domain. They can be used as an interface between domain ontologies and generic ontologies.

Top-level ontologies provide general notions of generic and intermediate ontological concepts. They are often used on top of a domain ontology. For example, the UMLS semantic network can be seen as a top-level ontology.

Task ontologies are using the vocabulary of generic, intermediate or domain ontologies in order to define tasks/activities on the basis of these ontologies, e.g., guidelines for the treatment of diseases. They are comparable to representation ontologies. A specialization are *domain task ontologies* [97], which are defined to be used within a specified domain.

In the context of organizational memories and information modeling [2] a categorization into information ontology, enterprise ontology, and domain ontology can be made:

[1] Sometimes this term is not accurately interpreted: Then, domain ontology is also seen as the instantiated knowledge base building on the defined domain ontology.

Information ontologies contain generic concepts and attributes, that apply to all kinds of information, e.g., the author or reliability of an information. They also define concepts and attributes for certain kinds of information sources.

Domain ontologies describe the content within the intended information system.

Enterprise ontologies provide the creation context and intended utilizations context of the knowledge.

In the previous paragraphs we have presented alternative classifications for ontologies. Any categorization of ontologies may be reasonable from a given viewpoint, and certainly there exist more and less known classifications of ontologies. However, in the context of the development of a diagnostic knowledge system the distinction between an ontology and a knowledge base is an interesting issue to consider.

Ontologies vs. Knowledge Bases

It is important to notice the difference between an ontology and a knowledge base. For example, Sowa [125] distinguishes the terms knowledge base and ontology as follows:

> **Knowledge Base:** An informal term for a collection of information that includes an ontology as one component. Besides an ontology, a knowledge base may contain information specified in a declarative language such as logic or expert-system rules, but it may also include unstructured or unformalized information expressed in natural language or procedural code.
>
> (also available online at `http://users.bestweb.net/~sowa/ontology/gloss.htm`)

Goméz-Pérez and Benjamins [97] give a similar definition:

> An ontology is a hierarchically structured set of terms for describing a domain that can be used as a skeletal foundation for a knowledge base.

We can summarize the definitions with the conclusion, that a (domain) ontology describes the conceptualization, which underlies a knowledge base. The concept consists of hierarchical relationships between the basic entities of the knowledge base and the semantics of their usage.
For this reason, a knowledge base can be seen as an instantiation of the declarations defined by a domain ontology. In the next section, we introduce the domain ontology applied in the context of this work. Furthermore, if we talk about the knowledge contained in the ontological container, then we consider instances of the presented domain ontology.

4.2. The Domain Ontology of a Diagnostic Knowledge System

As described in Section 1.2.3, our process model is limited on explicit representational languages. Thus, all remaining knowledge containers will build upon a fixed ontology and need not to be defined for each knowledge system project. In the following, we present a fixed ontology for building diagnostic knowledge systems.

The basic ontological classes of a diagnostic knowledge system are *diagnoses*, *questions*[2], *question sets* and *cases*. These entities directly correspond to the local system metaphor introduced in Section 2.2.1. Whereas the system metaphor emphasizes the knowledge engineering aspect, the ontology defined here considers more technical aspects of the entities.

In the past, ontologies has been described using KIF [141] or KL-ONE style languages. However, due to the popularity of UML [115] researchers started to formalize explicit knowledge and ontologies in (augmented) UML. For example, Cranefield and Purvis [35] propose UML in combination with OCL (Object Constraint Language, [113]) as a modeling language for ontologies. The use of UML for the development of ontologies is motivated in Kogut et al. [63]; successful applications of ontologies in UML are also reported. In their recent textbook Schreiber et al. [117] also use UML as their baseline modeling notion. Analogously, we describe the applied domain ontology in UML notion. In Figure 4.1 the domain ontology used for the agile process model is given. Abstract classes are depicted in italic font and are not used in an instantiation of this ontology.

A central part of the ontology is a *case*, which contains a (possibly empty) set of diagnoses and at least one question set. *Diagnoses* are classified into *problem areas*, which describe coarse solutions, and *final diagnoses* used to represent the exact solution for a case. A symbolic state is commonly assigned to a diagnosis depending on the current case presented to the system. Possible states are *unclear*, *suggested*, *probable* or *excluded* (not probable). A problem area itself can contain further problem areas or final diagnoses. In this way, diagnoses can be grouped hierarchically in order to define a taxonomy. In most cases this taxonomy describes *is-a* or *is-part-of* relations. During instantiation the exact meaning needs to be attached for each relation. Therapies, i.e., objects ”correcting” a detected diagnosis, are also represented as diagnoses, and are not specially treated by a separate entity class. Each *question set* in turn can contain other question sets or questions, which finally provide the input of the knowledge system. *Questions* are categorized according their expected value types into text, numerical, and choice questions. For choice questions we can distinguish between multiple-choice, one-choice and yes/no questions. For multiple-choice questions more than one answer can be given in a case; one-choice questions only can be assigned to at most one value in a case, and yes/no questions are a special kind of one-choice questions with only two assignable default values (i.e., yes/no). Additionally, questions can be classified according to their usage. Questions with usage *questioned* are directly presented in the dialog in order to be answered by the user (or by a connected machine for embedded systems). *Abstractions* syntactically are questions, which are not visible in the dialog and cannot be answered by the user. Values for abstractions are derived according to the values of other questions.

The basic functionality of a diagnostic system is depicted in Figure 4.2 as an augmented use-case diagram. The user starts a new consultation of the diagnostic system by creating

[2]This object is often named according to its current *role*: For the knowledge acquisition task it is commonly named *question*, and during the problem-solving task the names *finding*, *symptom* or *observation* are typical.

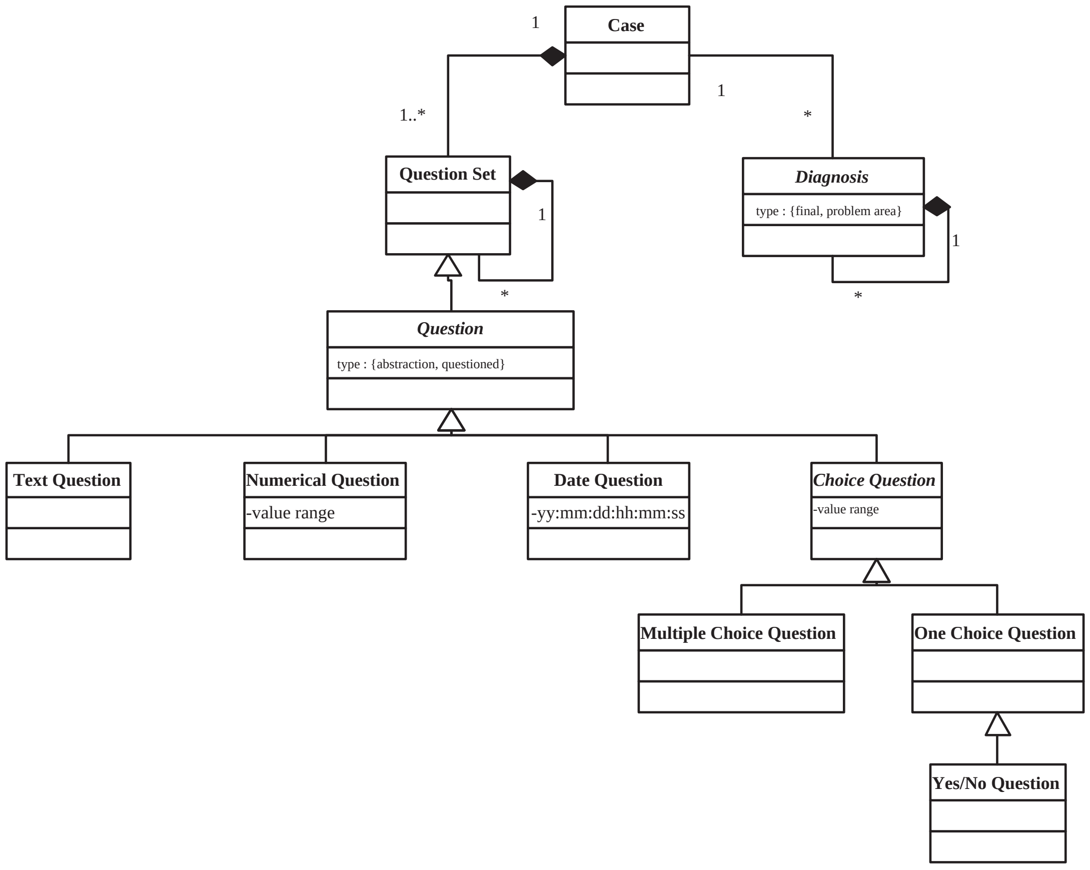


Figure 4.1: UML notion of the domain ontology (basic entities).

a new case. Within this case the diagnostic system is asking questions to be answered by the user or the user volunteers input. According to the user's response the system is able to provide diagnoses, that can explain the entered input. We emphasize that these diagrams only describe the basic object relations between the ontological entities. For a complete definition the notations of inferential relationships are still missing. However, since these inferential relations are part of the remaining knowledge containers we describe them later in combination with the presented containers.

4.3. Issues for the Instantiation of the Domain Ontology

When developing a diagnostic knowledge system the domain ontology needs to be instantiated by appropriate terms describing the focussed problem domain. Practical experience with defining instances of domain ontologies has shown that the following issues need to be considered: The level of granularity, the standard of knowledge of the designated users, the reusability, and the standardization. We discuss these issues in the following.

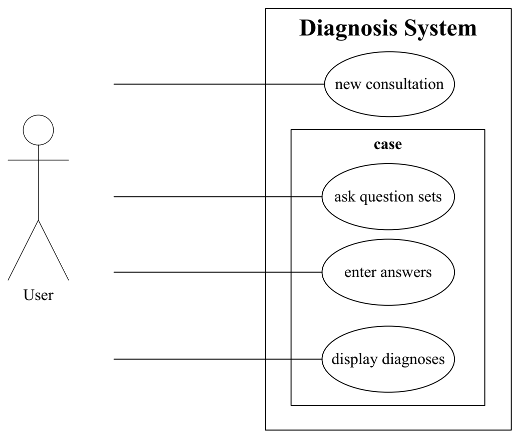


Figure 4.2: Augmented use-case diagram of a diagnostic system.

Level of granularity The level of granularity is one of the most difficult issues to consider for knowledge system design. It needs to be considered both for the granularity of diagnoses and for the granularity of questions. Some examples will clarify the problem in more detail.

For representing a solution its level of detail may be not clear from the beginning of the knowledge system project. In a medical system the developer has to decide about the level of detail for the included diseases, e.g., liver diseases. Modeling alternatives are the instantiation of a single diagnosis *liver disease* in contrast to specific diagnoses (e.g., hepatitis A-B-C, fat liver, cirrhosis of the liver, etc.). It is worth noticing, that the granularity of a solution often depends on the available therapy options.

For questions we face a similar problem: The developer has to decide how many questions should be asked to describe a focused finding. E.g., for a medical application a single question concerning the liver of a patient (with values normal/abnormal) may be appropriate. In the context of another medical application it would be necessary to provide a detailed set of questions asking for various aspects of the patient's liver. After the developer has decided about the number of questions the question type still remains open. E.g., for a medical application it is not obvious from the beginning how to represent a question like *temperature*. When represented as a numerical question the exact numerical value of the temperature has to be entered for any consultation. Often a more abstract value of the temperature is sufficient or suggestive. Then, the question may be represented as an one-choice question, e.g., with possible values {*normal, marginal, high, very high*}. We remark that also the value range of choice questions is often an issue of the level of detail. If the representation of the question is still too detailed, then the question may be instantiated as a text question. Then, the user can freely enter a text describing the answer of the question. However, the inferential power of text questions is limited when compared to numerical or choice questions. It is worth noticing, that the level of granularity corresponds to the diagnostic power of developed system (if sufficient knowledge is available).

User standard of knowledge When developing a knowledge system the standard knowledge of the designated users is also an important issue to consider. This topic is strongly related to the level of granularity, since on the one hand for expert users a highly detailed solution part is more valuable than coarse solutions. On the other hand, expert users are also able to answer abstract questions summarizing the interpretation of many concrete questions, that are typically presented to unexperience users. With the integration of additional explaining questions into the questionary as, e.g., implemented in the ECHODOC system [140], unexperienced users can be also enabled to enter high value input.

Reusability For larger knowledge systems, the reusability is an interesting issue to consider. Ontological objects like questions and diagnoses may be not only designed for the usage in the current knowledge base, but also for the (re)use in future parts of the knowledge base. However, the implementation of reusable objects can raise additional problems. For example, if we consider the design of a reusable choice question, then for one diagnosis a more detailed value range of the question will be necessary, which will be useless or bothering in context of another diagnosis. This problem can be addressed by the introduction of an abstraction that can infer an abstracted value, so that both the abstraction and the detailed value are available. Thus, an abstract and a detailed value for a finding are transparently represented.

Standardization An important issue for knowledge systems running in real world environments is the adaptation of standards. In many domains a terminological standard has been already established, and for many reasons it is advisable to conform the instantiation of the ontology to the existing standard. For example, in medical domains the ICD-10 standard is a well-known coding schema for diagnoses. The reuse of ICD-10 codes for the instantiation of diagnoses will provide a lot of possibilities like generation of standardized medical protocols or knowledge sharing with other (possibly connected) systems. However, in many cases the use of standards is opposing to other requirements of the knowledge system, e.g., the standard may be too specific or too general for the intended application.

We can summarize, that some of the presented issues consider conflicting goals (e.g., standardization vs. standard of knowledge) and therefore no uniform design guideline can be given in general. This problem is also represented by the *interaction problem* [29], which states that domain knowledge is always formulated in context to its usage during reasoning.
Although the definition of an ontology instance should be done thoroughly, requirements can arise that call for a change of the implemented ontology. The agile process model provides restructuring methods for modifying aspects of the ontology during the development of the knowledge system. Typical restructuring methods are briefly presented in the following. Furthermore, an important issue of the agile process model is the testing of the ontological knowledge container, which we explain in the following section. Before these two sections we will give a brief approach for acquiring ontological knowledge.

4.4. Acquisition of Ontological Knowledge

The acquisition of ontological knowledge considers the construction of the hierarchies of questions and the diagnoses. We describe for both hierarchies the typical development process.

4.4.1. Building the Question Hierarchy

The coarse structure of the question hierarchy is typically defined by abstract question sets (top-level groups), that are distinguished by their kind of examination method: Usually the expert orders the required question by the top-level groups ”initial survey”, ”manual examination”, and ”specialized tests”. For a medical application, this classification intuitively maps to ”history”, ”manual examination”, and ”tests”. An application in a technical domain may name this top-level groups ”complaints”, ”inspection”, and ”technical tests”. Within each top-level question set it is reasonable to order questions or question sets, so that firstly main findings are questioned and refined by follow-up questions. As discussed in Section 4.3 the developer needs to consider the level of granularity for questions, the detail of value ranges, and the user standard of knowledge. For example, expert users may find it annoying to answer additional questions, that are possibly helpful for beginners.

Often a hierarchical structure of questions and question sets may be not sufficient, because questions or question sets are meaningful at more than one point of the hierarchy. Then, we allow for grouping the question sets and questions hetarachically. Usually, the structure of the question hierarchy is changing during knowledge base development. Then, restructuring methods can help to automatically update relations between questions and other knowledge types. Typical changes consider the change of the question kind, the adaptation of value ranges for choice questions, or deletions of questions. Of course, appropriate tests methods need to be used to accompany the restructurings.

4.4.2. Building the Diagnosis Hierarchy

The diagnosis hierarchy is constructed by inserting new diagnoses in a structured way. Then, final solutions are grouped and abstracted by coarse diagnoses, that can be interpreted as intermediate solutions, or problem areas giving an abstracted conceptual description of the underlying diagnoses. As mentioned for the question hierarchy the level of detail of the diagnosis hierarchy is an important issue to consider, which we also discussed in Section 4.3. For the definition of a standardized diagnosis hierarchy often terminological or branch ontologies are used. We provide appropriate test and restructuring methods, since during the knowledge system development the structure of the diagnosis hierarchy is also a subject of change.

4.5. Testing Ontological Knowledge

We present three methods for testing the ontological knowledge container. In contrast to other knowledge containers, the test methods for ontological knowledge cannot evaluate the behavior of the knowledge system, but can check the implemented ontology for anomalies, its understandability, and its correctness with respect to standard ontologies.

4.5.1. Static Ontology Testing

The *static ontology testing* method is used for obtaining an overview of the structure of the implemented knowledge.

Mechanics The method generates a statistics of the implemented ontological entities. The knowledge base is investigated and the following metrics are reported: For diagnoses we count the total number and percentage of problem areas and final diagnoses, respectively. For problem areas we additionally compute the minimum and maximum number of contained diagnoses (with mean values). For final diagnoses the minimum and maximum depth of the diagnoses with respect to the root diagnosis is computed (with mean values).

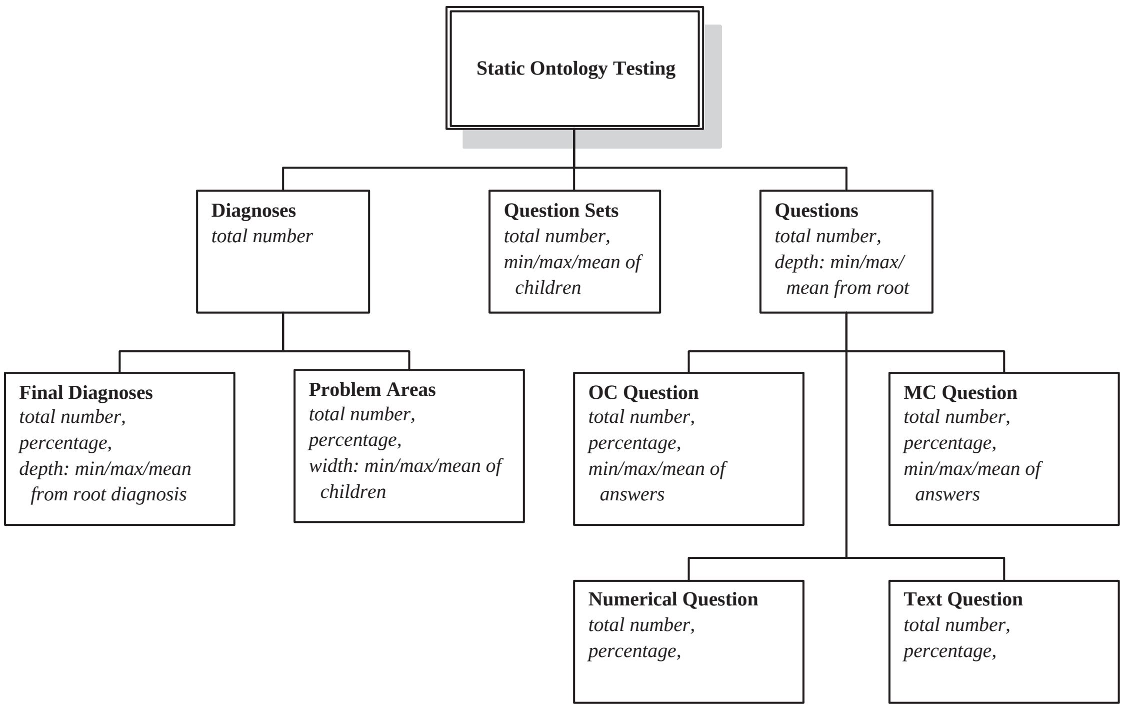

Static Ontology Testing
Diagnoses
total number
Question Sets
total number, min/max/mean of children
Questions
total number, depth: min/max/ mean from root
Final Diagnoses
total number, percentage, depth: min/max/mean from root diagnosis
Problem Areas
total number, percentage, width: min/max/mean of children
OC Question
total number, percentage, min/max/mean of answers
MC Question
total number, percentage, min/max/mean of answers
Numerical Question
total number, percentage,
Text Question
total number, percentage,

Figure 4.3: The considered metrics of the static ontology testing method.

For question sets we simply depict the total number contained in the knowledge base and the minimum/maximum values of contained questions and question sets, respectively (with mean values).

Besides the total number of questions we additionally report the minimum and maximum depths of questions with respect to the root question set (with mean values). Furthermore, we count for each question type (one-choice, multiple choice, numeric, text) the total number and the percentage. For one-choice and multiple-choice questions we additionally calculate the minimum and maximum number of represented answer alternatives (with mean values), and the total number of implemented choice values. In Figure 4.3 the measures calculated by the static ontology testing method are depicted in a diagram. In order to facilitate an automatic analysis of the generated statistics the method tries to detect significant peaks, that are displayed as warnings to the user. The following irregularities are detected:

- Question sets containing more questions than specified by a user defined threshold $\mathcal{T}_{maxQS}$
- Choice questions with...
  - exceptionally large or small value ranges relative to the mean size of implemented value ranges
  - a value range less than a user defined threshold $\mathcal{T}_{minVal}$
  - a value range greater than a user defined thresholds $\mathcal{T}_{maxVal}$
- Problem areas with...
  - exceptionally few diagnoses as sub-concepts relative to the mean value of implemented sub-concepts
  - only one diagnosis as sub-concept
  - more diagnoses as sub-concepts than defined by a threshold $\mathcal{T}_{maxDC}$
- Diagnoses and questions with an exceptional depth with respect to the root of the hierarchy

Statistical information about the remaining entities of the knowledge base is only displayed on demand. If no warnings are reproted, then the test method finishes with a success message.

Usage The method can be applied to investigate the *understandability* of the implemented knowledge. Then, detected irregularities can indicate problems regarding the understandability. According to the definitions known from *refactoring* in software engineering we call these irregularities *bad smells*. For a complete static ontology testing the following thresholds need to be defined: $\mathcal{T}_{maxQS}$ for the maximum size of a question set, $\mathcal{T}_{minVal}$ and $\mathcal{T}_{maxVal}$ for the minimum and maximum size of a choice value range, respectively, and $\mathcal{T}_{maxDC}$ for the maximum number of sub-concepts for a problem area.

4.5.2. Case-Based Ontology Testing

With *case-based ontology testing* one can determine the usage of the ontological objects under real world conditions. Thus, real cases are applied to find objects with very seldom and very frequent usage.

Mechanics A sufficient large number of real cases is required for this method. It is important that the applied case base represents a typical collection of cases gathered from real life usage. In the best case, all obtainable real cases are utilized.

Then, the real cases are given to the knowledge system and the usage of the ontological entities, i.e., diagnoses and findings, is counted. All entities that are never used in any case are indicated as an error. The low usage of entities is defined by a threshold, e.g., 1% of all cases, and will be reported as a warning. However, sometimes unfrequent diagnoses are still important in the context of the knowledge system. For a convenient use of the test method these objects should be marked as *important*, and thereafter the method does not consider them in the test results.

Usage Case-based ontology testing can detect diagnoses and findings, that were thought to be useful in the past, but appeared not to be necessary in real life cases. Such *lonely* objects in the ontology constitute redundancy in the knowledge base, which is a kind of *anomaly*. It is obvious, that a large number of (real) test cases is needed as test knowledge. With a given threshold for low usage of objects the test can be automatically executed.

4.5.3. Standardization Testing

With *standardization testing* an ontology can be checked against a given standard ontology. For example, the diagnosis hierarchy of a medical application can be checked against the ICD-10 standard .

Mechanics The method investigates the ontological hierarchy of the implemented ontological knowledge and checks the relations against a given ontology. A simple way for this check is the comparison of textual names specified for the implemented ontological entities and the standard ontology entities, respectively. Then, both ontologies are compared according to their linkage. An error is reported, if at least one link between implemented ontological entities is erroneous. Otherwise, the test finishes with a success message.

Usage With standardization testing the developer can detect incorrect ontological hierarchies. Therefore, this method can be applied for testing the *correctness* of the ontological knowledge container. However, the application of this method may be difficult in practice: For a proper comparison between the implemented ontology and the standard ontology, both must follow the same naming conventions and often the same granularity. This may produce conflicts, if the application, e.g., requires a more detailed or coarse definition of diagnoses.

4.6. Restructuring Ontological Knowledge

Restructuring of already implemented knowledge is one of the key practices of the agile process model. Since the ontological knowledge container provides the basis for all remaining knowledge containers, changes of ontological knowledge often imply changes of other knowledge containers. Thus, if an ontological restructuring method is executed, then the corresponding restructuring methods of the other knowledge containers are successively triggered in order to propagate the ontological change. In the following, we only will sketch some useful methods for ontological methods.

4.6.1. Restructurings on Objects

Restructuring methods on objects usually consider the transformation of an ontological object or the deletion of an existing one.

TRANSFORMMCINTOYN A multiple-choice question may contain answer alternatives with no semantical relation. Then, the TRANSFORMMCINTOYN method can convert the multiple-choice question, and its answer alternatives, respectively, into a set of yes/no-questions corresponding with the answer alternatives of the original multiple-choice question. Furthermore, this method facilitates the move of the converted yes/no questions into different question sets (see MOVEQUESTION method) after execution of the restructuring method. All remaining knowledge containers may be affected by this restructuring method, and therefore the corresponding methods for these methods are triggered subsequently.

TRANSFORMYNINTOMC The conversion of a collection of yes/no questions into a single multiple-choice question can be also reasonable. For example, a set of related yes/no questions are converted into one multiple-choice question in order to facilitate a more efficient user dialog. This is motivated by the fact that users are commonly answering one multiple-choice question faster than multiple yes/no questions. Analogously to the inverse method TRANSFORMMCINTOYN all remaining knowledge containers may be affected by the execution of this method, and hence are triggered consequently.

TRANSFORMNUMINTOOC The TRANSFORMNUMINTOOC method is often applied, if questions of the domain ontology were initially assumed to be acquired numerically, but then a qualitative gathering appeared to be more practical. For example, the entry of a qualitative answer choice is much simpler than providing an exact numerical value for a question. The remaining knowledge containers may be affected by this ontological change, and therefore the corresponding methods of the remaining containers are triggered consequently.

SHRINKVALUERANGE Often domain experts start implementing the ontological container with choice questions providing detailed value ranges. During ongoing development of, e.g., the structural knowledge container the value range of some questions exposes to be unnecessary precise. Furthermore, a decreased value range may simplify the dialog for the end-users. Then, the SHRINKVALUERANGE method can be applied for reducing the value range of choice questions. A transformation matrix has to be defined by the developer, which maps the values of the old value range to the values of the reduced values range. Since, all remaining knowledge containers may be affected by this restructuring, the corresponding methods of these methods are triggered subsequently.

MoveQuestionValue During the agile development of the knowledge system the intended meaning of a (choice-)question can evolve. It can happen that a value of the question is no longer semantically belonging to the question, but is better placed in the value range of another choice question. This situation is typical, if the gathering of originally one question is split up into two separate questions. The application of the MoveQuestionValue method moves the specified answer value of a choice question to the value range of another choice question, and then propagates this restructuring to the remaining knowledge containers. However, the method can cause multiple conflicts, e.g., if the original and the targeted one-choice questions are contained in a rule condition, both combined by an AND-connector.

IntroduceAbstraction The introduction of an abstraction (abstract question) may be meaningful for many reasons. For example, an abstraction often simplifies the complexity of structural and strategic knowledge (e.g., reducing the number of questions in a rule condition). Furthermore, an abstraction can increase the understandability for the end-user when used as an explanation instead of the number of original questions. The execution of the IntroduceAbstraction method firstly creates a new abstract question (if not already existent). In a second step, the developer specifies a condition of a set of findings (connected by *and* or *or*) that define a given value of the abstraction. This condition is triggered to the remaining knowledge containers.

RemoveDiagnosis The diagnosis hierarchy may evolve to be too structured and overdesigned during knowledge system development. Then, some diagnoses become redundant or useless. For increasing the understandability of the implemented knowledge, it is reasonable to delete redundant diagnoses. However, simply deleting a diagnosis can not only cause conflicts within the ontological knowledge container (e.g., when removing a problem area with several child diagnoses), but also can be difficult in context of the remaining knowledge containers (e.g., if the diagnosis is contained in structural knowledge). The RemoveDiagnosis method starts with analysing the implemented knowledge, and then interactively removes or re-links attached diagnoses or knowledge of the diagnosis to be removed.

RemoveQuestion Analogously to the RemoveDiagnosis method the RemoveQuestion restructuring removes not only a question from the ontological knowledge container, but also considers conflicts that may arise in the remaining knowledge containers with respect to the deletion.

4.6.2. Architectural Restructurings

Architectural restructuring methods focus on changing the design of the knowledge base. Often hierarchical relationships are modified or rearranged.

MoveQuestion Usually questions are grouped in question sets, and structured hierarchically. Due to extensions of the ontology, it may be reasonable to move an existing question to another (possibly new) question set. Then, the MoveQuestion restructuring can be applied, which provides a convenient method for this task. Since hierarchical changes in the ontology always can affect strategical knowledge the method additionally triggers the corresponding method of the strategic knowledge container.

MoveDiagnosis Analogously, there exists a method for moving diagnoses from one place in the diagnosis hierarchy to another place. The MoveDiagnosis restructuring, e.g., is useful, when a diagnosis was initially designed to represent a final solution, and becomes a problem area due to extensions and refinements of the knowledge base. Changes of the diagnosis hierarchy can affect the implemented strategic knowledge, if a diagnosis-centered indication strategy is used. Then, the available strategic knowledge belonging to the moved diagnosis needs to be considered manually by the user.

ExtractQuestionSet During the development of the knowledge system the size of a question set may has become too large, i.e., too many questions are contained in the question set. Then, the ExtractQuestionSet method can be applied, to simply move a number of questions into a newly created question set. For this method available strategic knowledge needs to be considered, and therefore the corresponding restructuring method for strategic knowledge needs to be triggered.

ComposeQuestionSets Inversely, two question sets can be combined to one question set by the ComposeQuestionSets method, if for example both question sets contain only a small number of questions, or many (semantic) relations can be identified between questions of the two question sets. Subsequently, the corresponding restructuring method for the strategic knowledge container is triggered.

4.7. Learning Ontological Knowledge

The automatic transfer of ontological knowledge is possible, if a standard ontology for the present application domain is available. However, often the standard ontology needs to be adjusted according to the project requirements. For example, the standard ontology may be too specific or too general for the intended application.

The (semi-)automatic construction of ontological knowledge from scratch is a current issue of research. We omit a detailed discussion in the context of this work, but refer to Maedche et al. [70] for learning taxonomic relations and to Staab and Studer [126, Ch.9] for a general introduction into semi-automatic learning algorithms considering the construction of ontologies.

4.8. Formal Definition of Ontological Objects

In the previous sections we introduced a domain ontology for building diagnostic knowledge systems. In the following chapters we will present approaches for implementing structural, strategic, and support knowledge building on this ontology.

However, these approaches do not require such a detailed distinction between the different kinds of an ontological object, e.g., the differentiation between a questioned and an abstracted question. In fact, the presentation of the methods is simplified by a common framework of general definitions, that consider a concrete instantiation of the presented domain ontology. This framework is provided in this section.

Firstly, we want to consider the objects, that are used as an input of a diagnostic knowledge system.

Definition 4.8.1 (Question and Finding) 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 value range $dom(Q)$. The value range can define

- numerical value ranges for real or integer values,
- symbolic value ranges containing choice answers, and
- arbitrary content for text answers.

A value $v \in dom(Q)$ assigned to a question $Q \in \Omega_Q$ is called a *finding* and we call $\Omega_{\mathcal{F}}$ the set of all possible findings in the given problem domain. A finding $F \in \Omega_{\mathcal{F}}$ is denoted by $Q{:}v$ for $Q \in \Omega_Q$ and $v \in dom(Q)$. Each finding F is defined as a possible input of a diagnostic knowledge system.

For findings the functions $a : \Omega_{\mathcal{F}} \rightarrow \Omega_Q$ and $val : \Omega_{\mathcal{F}} \rightarrow \Omega_{val}$ are defined to obtain the assigned question and the assigned value of a finding, respectively. Then, for a finding $Q{:}v$ we obtain $a(Q{:}v) = Q$, and $val(Q{:}v) = v$.

Questions can be structured by question sets into meaningful groups, e.g., tests concerning a specific area of the system are grouped into a common question set.

Definition 4.8.2 (Question Set) Let Ω_Q the universe of all questions defined for the knowledge system. Then, an ordered list of questions $Q_i \in \Omega_Q$

$$QS = (\, Q_1, Q_2, \dots, Q_n \,)$$

is called a *question set*. An *abstract question set* $QS^* = (\, Q_1, Q_2, \dots, Q_n, QS_1, \dots, QS_m \,)$ additionally can contain other question sets QS_i. We call Ω_{QS} the universe of all question sets (including abstract question sets) for a given universe of questions Ω_Q.

With abstract question sets it is possible to group questions and question sets hierarchically.

A diagnostic system usually comes up with a solution for a given problem. These solutions are defined as diagnoses.

Definition 4.8.3 (Diagnosis) Let D be a *diagnosis* representing a possible output, i.e., a solution, of the diagnostic knowledge system. We define $\Omega_{\mathcal{D}}$ to be the universe of all possible diagnoses for a given problem domain. For each diagnosis $D \in \Omega_{\mathcal{D}}$ a symbolic value $val(D)$ is assigned, which provides the diagnosis state with respect to a given problem. The value range of a diagnosis D is denoted by $dom(D)$.

For example, a reasonable range of symbolic diagnosis states is

$$dom(D) = \{not\ probable, unclear, suggested, probable\} .$$

A problem is presented to the knowledge system as a case, which we will define in the following.

Definition 4.8.4 (Case) A *case* c is defined as a tuple

$$c = (\,\mathcal{F}_c, \mathcal{D}_c, \mathcal{I}_c\,)\,,$$

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, and $\mathcal{I}_c$ contains additional meta-information describing the case c in more detail. The set of all possible cases for a given problem domain is denoted by Ω_C.

It is worth noticing, that the definition above represents an already solved problem. Of course, for a new problem the solution is not known in advance and often the set of observed findings is entered successively into the knowledge system. In the following chapters we will use these definitions for the description of the knowledge containers.

4.9. Summary

In this chapter, we have introduced the ontological knowledge container, which denotes the basis for building diagnostic knowledge systems. We firstly presented different categories of ontologies with respect to their subject of utilization. Then, we motivated the definition of a fixed domain ontology for diagnostic systems and introduced the applied domain ontology of the agile process model. We discussed problems that developers often face when building an application ontology for a knowledge system, e.g., level of detail or standardization. Furthermore, methods for automatically testing and learning the ontological knowledge container were presented. We concluded the chapter by providing a general framework of definitions representing an instantiation of the specified domain ontology. This framework will be utilized in the subsequent chapters for introducing the remaining knowledge containers.

5. The Structural Knowledge Container

In this chapter, we describe the structural knowledge container, which embodies the inferential knowledge used by a diagnostic system. As mentioned in Chapter 1 there exist various kinds of inferential knowledge representations that fit the mental models of domain experts and that are appropriate for different types of applications. In particular, we discuss the usage of *abstraction knowledge, case-based knowledge, categorical knowledge, score-based knowledge,* and *causal set-covering knowledge* in more detail.

5.1. Abstraction Knowledge

According to Giunchiglia and Walsh [46] the term *abstraction* can be defined as a *process of mapping a representation of a problem (ground representation) into a new representation (abstract representation)*. Furthermore, typical properties of the abstract representation are its simple handling compared to the ground representation and its preservation of certain desirable properties of the ground representation. In the context of the presented domain ontology we restrict the abstraction task to the mapping of findings to other findings, that are both part of the ontological knowledge container.

Thus, sometimes findings representing raw (input) data are mapped into findings describing a meaningful abstraction of the input. It is worth noticing, that we cannot clearly separate between highly abstracted findings and diagnoses, since abstractions can describe concepts similar to diagnoses. Abstractions can be interpreted as ontological knowledge as well as structural knowledge: From the viewpoint of stating an abstract concept for other ontological entities, it can been seen as ontological knowledge. Then, abstractions often state high-level technical terms of the domain. However, for inferring the actual value of an abstraction structural knowledge is required. In the context of this work we classify abstraction knowledge as structural knowledge, since defining the knowledge actually *deriving* an abstraction denotes an even more complex task than defining the ontological concepts of an abstraction.

5.1.1. Representation of Abstraction Knowledge

There exist alternative approaches for implementing abstraction knowledge: Starting with simple tabular mapping functions and abstraction formulas we can define more complex abstraction knowledge by plain abstraction rules or by score-based abstraction rules. We briefly discuss these approaches in the following four paragraphs.

Mapping Functions

Mapping functions can be applied, if the abstraction knowledge is very simple. A mapping function is defined for the values of a question, and directly transfers them to the values of an abstracted question. A suitable knowledge acquisition method for defining mapping functions are *tables*. The following table in Figure 5.1 shows an exemplary mapping function for the question ”temperature”.

	temperature			
fever	< 35	in [35, 37)	in [37, 38)	≥ 38
none	⊠	⊠		
increased			⊠	
high				⊠

Figure 5.1: Tabular representation for mapping the numerical values of question ”temperature” to a symbolic value of the question ”fever”.

Thus, for a numerical value of finding ”temperature” less than 35, and for a value between 35 and 37 the abstraction ”fever:none” is derived. For values of the question ”temperature” between 37 and 38 the abstraction ”fever:increased” is inferred, and values greater than 38 yield the abstraction ”fever:high”.

Formula-Based Abstraction

If one or more numerical questions are the starting point for the derivation of an abstraction, then the formula-based abstraction can be a useful method.

Then, the developer defines a formula that uses the raw values of the numerical questions as an input and computes the corresponding value of the abstraction.

In medical applications, a typical example for formula-based abstraction is the derivation of the ”body mass index” (BMI), which (in the simple case) uses the weight and the height of a human body as input:

$$bmi(\text{weight}, \text{height}) = \frac{\text{weight}}{\text{height}^2}$$

The computed value can be further abstracted according to the specifications of the WHO (using a mapping function):

	bmi(weight, height)				
weight class	< 18.5	$[18.5, 25)$	$[25, 30)$	$[30, 40)$	≥ 40
underweight	⊠				
normal		⊠			
overweight			⊠		
obesity				⊠	
extreme obesity					⊠

Figure 5.2: Mapping function for evaluating the body mass index.

Thus, for example the abstraction ”weight class:extreme obesity” is derived for patients with a body mass index greater than or equal to 40.

Abstraction Rules

Abstraction rules are a general representation of abstraction knowledge. Thus, a rule condition may contain arbitrary logical combinations of conditions as defined in Figure 5.1, and the rule action will set the value of an abstracted question.
In a first step we define a general rule. We will revive this definition in the further chapters, when introducing categorical and score-based knowledge.

Definition 5.1.1 (General Rule) A *general rule* r is defined as follows:

$$r = cond(r) \rightarrow action(r)\,[except(r), context(r)]\,,$$

where $cond(r)$ is a rule condition containing disjunctions, conjunctions, and/or negations of arbitrary findings $F \in \Omega_{\mathcal{F}}$; $action(r)$ is the rule action, that is executed if the rule condition evaluates true in a given case. In addition to the rule condition $cond(r)$ other conditions can be defined, i.e., a rule exception $except(r)$ and a rule context $context(r)$. Both conditions are optional and can contain disjunctions/conjunctions of findings and diagnoses states.

The presented definition of rules containing rule exceptions and rule contexts is motivated by long term experiences with medical knowledge system projects and goes back to Puppe [99]. The rule exception was introduced with respect to rarely observed findings that — if observed — detain a rule from firing. Then, even if the rule condition evaluates true, the rule is prevented from firing, if the rule exception evaluated true. If the rule exception evaluates false or cannot be evaluated at all (e.g., not all necessary findings are observed), then the rule is not detained from firing. The difference between the separate concept of a rule exception compared to an extended rule condition can be best explained by an example: We consider the following two rules r_1, r_2:

$$\begin{aligned} r_1 &= C_1 \rightarrow A \quad except\, C_2\,, \\ r_2 &= C_1 \wedge not(C_2) \rightarrow A\,, \end{aligned}$$

where C_1, C_2 are conditions and A is an arbitrary rule action.
On the one hand, with the concept of rule exceptions the rule r_1 can fire, even if the findings in C_2 are not observed and the condition cannot be evaluated at all. It is obvious that on the other hand the rule r_2 will not fire until all findings in C_1 and C_2 are observed and consequently can be evaluated. For a defined rule context $context(r)$ the rule only fires, if the rule condition $cond(r)$ evaluates true and the rule context $context(r)$ also evaluates true.
The concept of rule contexts is often applied if the rule base is structured into several modules. Then, the same rule context (often evaluating the established state of a special diagnosis) is attached to a number of rules, which are only allowed to fire if a specified diagnosis (e.g., a problem area) has been established.

A rule condition COND is described with respect to the presented domain ontology. In Table 5.1 the possible configurations of a rule condition are depicted in BNF (Backus Naur Form). Abstract concepts of conditions are written in capital letters, and concrete instances of conditions are written in small letters. In summary, conditions can be distinguished into terminal and non-terminal conditions. Terminal conditions directly apply constraints on single questions, whereas non-terminal conditions represent a composite grouping a set of arbitrary conditions. Terminal conditions are defined on all kinds of questions and diagnosis' states.

COND	=	TERMINAL
		|| NON_TERMIINAL
NON_TERMINAL	=	and(COND$^{1..n}$)
		|| or(COND$^{1..n}$)
		|| not(COND)
		|| minMax(COND$^{1..n}$, min, max)
TERMINAL	=	NUM_COND
		|| CHOICE_COND
		|| DATE_COND
		|| TEXT_COND
		|| KNOWN_COND
		|| DIAGNOSIS_COND
NUM_COND	=	numEqual(numQuestion, value)
		|| less(numQuestion, value)
		|| lessEqual(numQuestion, value)
		|| greater(numQuestion, value)
		|| greaterEqual(numQuestion, value)
		|| numIn(numQuestion, valueRange)
CHOICE_COND	=	choiceEqual(choiceQuestion, value)
		|| choiceIn(choiceQuestion, valueList)
DATE_COND	=	dateEqual(dateQuestion, dateValue)
		|| dateLess(dateQuestion, dateValue)
		|| dateLessEqual(dateQuestion, dateValue)
		|| dateGreater(dateQuestion, dateValue)
		|| dateGreaterEqual(dateQuestion, dateValue)
		|| dateIn(dateQuestion, dateValueRange)
TEXT_COND	=	textEqual(textQuestion, textValue)
		|| textContains(textQuestion, textValue)

KNOWN_COND	=	known(question)
		|| unknown(question)
DIAGNOSIS_COND	=	established(diagnosis)
		|| suggested(diagnosis)
		|| excluded(diagnosis)

Table 5.1: The condition framework of rules in BNF.

Besides the usual non-terminal conditions *and* and *or* the *minMax* condition represents a special non-terminal concept. For this condition a set of conditions is defined together with a minimum *min* and maximum *max* threshold (with *max* less that or equal to the number of defined conditions). Then, the *minMax* condition evaluates true if at least *min* conditions and at most *max* of the contained sub-conditions evaluate true. It is easy to see that a *minMax* condition is a compact representation of a semantically equivalent condition, containing a disjunction of conjunctions. For example, the condition $minMax(c_1, c_2, c_3, 2, 2)$ is semantically equivalent to the condition $or\big(and(c_1, c_2), and(c_1, c_3), and(c_2, c_3), not(and(c_1, c_2, c_3))\big)$ and will evaluate false if all three sub-conditions c_1, c_2, and c_3 evaluate true.

Definition 5.1.2 (Rule Base) 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*.

With the previous definitions we can represent arbitrary rules. However, the characteristics of a rule base can be best explained by the *complexities* of the included rules.

Definition 5.1.3 (Rule Complexity) Let $r \in \Omega_{\mathcal{R}}$ be a rule with

$$r = cond(r) \rightarrow action(r)\,.$$

Then, we say that r is a *simple* rule, if $cond(r)$ only consists of a terminal condition. Further, we say that r is a *one-level* rule, if $cond(r)$ contains only one non-terminal condition, which itself consists of terminal conditions. A rule r is called a *multiple-level* rule if the condition $cond(r)$ contains more than one non-terminal condition.

An example clarifies the given definition of rule complexities.

Simple rule
$r_1 = \quad less(Q_1, 10)$
One-level rule
$r_2 = \quad and(less(Q_1, 10), less(Q_2, 20))$
Multiple-level rule
$r_3 = \quad and(or(less(Q_1, 10), less(Q_2, 20)), or(less(Q_3, 15), less(Q_4, 25)))$

Specific kinds of rules can be classified according to their specialized rule action. Thus, a general rule is called an abstraction rule, if its rule action is executing an abstraction task.

Definition 5.1.4 (Abstraction Rule) A rule

$$r = cond(r) \rightarrow Q{:}v\,[except(r), context(r)]$$

is called an *abstraction rule*. If the abstraction rule fires, then the specified value $v \in dom(Q)$ is assigned to the specified question $Q \in \Omega_Q$.

For abstractions deriving a numerical value we additionally allow formulas contained in the rule action. Thus, the developer can determine the value of a numeric abstraction by performing (complex) computations. The value of the numeric abstraction may provide an input for further abstractions. It is worth noticing, that mapping functions can be interpreted as abstraction rules. Then, for the mapping function defined in Figure 5.1 we can define the four semantically equal abstraction rules:

$$\begin{aligned}
r_1 = {} & \text{less}(\text{temperature}, 35) \rightarrow \text{fever:none} \\
r_2 = {} & \text{numIn}(\text{temperature}, [35, 37)) \rightarrow \text{fever:none} \\
r_3 = {} & \text{numIn}(\text{temperature}, [37, 38)) \rightarrow \text{fever:increased} \\
r_4 = {} & \text{greaterEqual}(\text{temperature}, 38) \rightarrow \text{fever:high}
\end{aligned}$$

Formula-based abstraction rules can be generalized to abstraction rules as well (using formulas in the rule action). Then, for each question Q contained in the abstraction formula the rule condition of a generalized abstraction rule will be a conjunction of terminal conditions *known(Q)*, i.e., the abstraction rule will only evaluate the formula, if the participating questions are already known.

Score-Based Abstraction

Scores can be applied for defining a more complex kind of abstraction knowledge. Then, the value of an abstraction is not defined by the evaluation of a single condition, but can be specified with respect to the evaluation of multiple weighted conditions. In contrast to abstraction rules the score-based representation does not insist on the precise valuation of defined conditions, but allows for a more coarse definition of the actual abstraction condition.
Formally, a score-based abstraction is defined by a score-based abstraction rule:

Definition 5.1.5 (Score-Based Abstraction Rule) A *score-based abstraction rule* r is denoted as follows:

$$r = \big[(cond_1, p_1), \ldots, (cond_n, p_n)\big] \rightarrow \big[Q : [m_1, \ldots, m_k]\big]\,,$$

where $(cond_i, p_i)$ is an augmented rule condition that contains a rule condition $cond_i$ attached with points $p_i \in I\!N$, and $[m_1, \ldots, m_k]$ with $(m_i \in I\!N)$ is a valuation scheme for the abstracted question $Q \in \Omega_Q$. By definition the size k of the valuation scheme is fixed to $k = |dom(d)| - 1$.
The abstraction rules fires, i.e., assigns the value $v_i \in dom(Q)$ to the abstraction $Q \in \Omega_Q$ according to the sum of points of true conditions $s = \sum p_i$, and a mapping function generated by the valuation scheme $[m_1, \ldots, m_k]$:

Q	$s < m_1$	$s \in [m_1, m_2)$	$\ldots$	$s \geq m_k$
v_1	☒			
v_2		☒		
$\vdots$			$\ldots$	
v_{k-1}				☒

Score-based abstraction rules can be simplified to *threshold-based abstraction rules*: Then, a specified question is assigned to a pre-defined value, if the sum of points in a condition exceeds a given threshold.

Definition 5.1.6 (Threshold-Based Abstraction Rule) A *threshold-based abstraction rule* r is denoted as follows:

$$r = \big[(cond_1, p_1), \ldots, (cond_n, p_n)\big] \rightarrow \big[Q{:}v : m\big] \, ,$$

where $(cond_i, p_i)$ is an augmented rule condition containing a rule condition $cond_i$ attached with points $p_i \in I\!N$; $Q \in \Omega_Q$ is an abstracted question with specified value $v \in dom(Q)$, and $m \in I\!N$ is a threshold value. The abstraction rule fires, i.e., sets the specified value v to the targeted abstraction Q, if the sum of points p_i of true conditions $cond_i$ is greater than or equal to the defined threshold m.

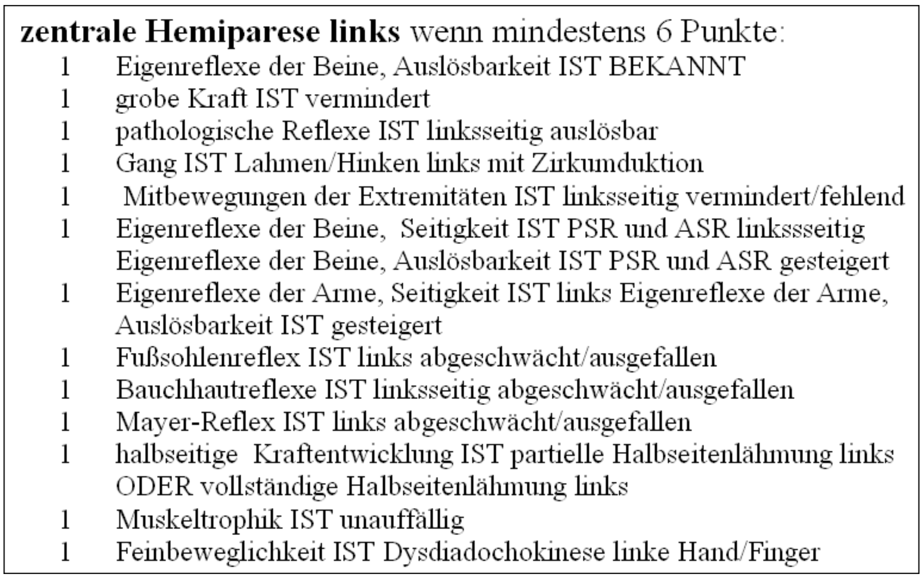
Score-based abstractions and threshold-based abstraction rules have been successfully applied in real world knowledge systems, e.g., for a medical knowledge system for neurology. For many years, the system was delivered together with a successful medical textbook [92]. In Figure 5.3 an example taken from this system is depicted for the abstraction ”central hemiplegia:left”. The threshold-based abstraction contains 13 rule conditions all attached with the default point 1. The minimum threshold is set to 6, which means that at least 6 sub-conditions need to evaluate true, in order to set the abstraction ”central hemiplegia” to the value ”left”.

Threshold-based abstraction rules are a generalization of abstraction rules, since on the one hand they offer a richer representation of rule conditions and on the other hand are able to attach points (i.e., weights) for each rule condition. The activation of a threshold-based abstraction rule is controlled by the minimum threshold.

Thus, an abstraction rule $r = cond(r) \rightarrow Q{:}v$ can be easily generalized to a threshold-based abstraction rule $r' = \big[(cond(r), 1)\big] \rightarrow \big[Q{:}v : 1\big]$.

5.1.2. Acquisition of Abstraction Knowledge

In general, for large knowledge systems an abstraction layer is inserted before applying knowledge for inferring diagnoses. As depicted in Figure 5.4 raw input findings are firstly used by abstraction knowledge to infer an abstract description of the case. Then, these abstractions are used to apply diagnostic knowledge for inferring diagnoses. This approach is described in more detail in Puppe et al. [104, p. 164f].



zentrale Hemiparese links wenn mindestens 6 Punkte:

1	Eigenreflexe der Beine, Auslösbarkeit IST BEKANNT
1	grobe Kraft IST vermindert
1	pathologische Reflexe IST linksseitig auslösbar
1	Gang IST Lahmen/Hinken links mit Zirkumduktion
1	Mitbewegungen der Extremitäten IST linksseitig vermindert/fehlend
1	Eigenreflexe der Beine, Seitigkeit IST PSR und ASR linksseitig Eigenreflexe der Beine, Auslösbarkeit IST PSR und ASR gesteigert
1	Eigenreflexe der Arme, Seitigkeit IST links Eigenreflexe der Arme, Auslösbarkeit IST gesteigert
1	Fußsohlenreflex IST links abgeschwächt/ausgefallen
1	Bauchhautreflexe IST linksseitig abgeschwächt/ausgefallen
1	Mayer-Reflex IST links abgeschwächt/ausgefallen
1	halbseitige Kraftentwicklung IST partielle Halbseitenlähmung links ODER vollständige Halbseitenlähmung links
1	Muskeltrophik IST unauffällig
1	Feinbeweglichkeit IST Dysdiadochokinese linke Hand/Finger

Figure 5.3: A score-based abstraction for defining the value "left-side" of abstraction "central hemiplegia" (in german).

As mentioned in the introduction of this section, abstraction knowledge can be interpreted partially as ontological and partially as structural knowledge. If we consider the ontological aspect of defining abstractions, then we can distinguish between a top-down and a bottom-up approach.

The *top-down approach* starts with investigation of the diagnostic profiles, i.e., the set of primary findings used for inferring each diagnosis. These findings are reviewed according to their understandability and expressiveness for the user and the dialog, respectively. Primary findings with high level of abstraction are defined as abstractions and additional questions are inserted describing this abstraction in more detail.

The *bottom-up approach* investigates the already defined questions. If a group of questions (often defined in the same question set) is frequently applied in the context of deriving a solution, then the introduction of an abstraction is considered in order to increase the reusability and expressiveness of this question group.

Having implemented the ontological aspects of the abstractions, we need to define structural knowledge deriving values of the abstractions. Then, we select from the previously introduced representations (e.g., mapping functions, abstraction rules) depending on the requirements of the abstraction task.

5.1.3. Testing Abstraction Knowledge

In the previous sections we motivated that all representational alternatives can be generalized to threshold-based abstraction rules. For abstraction rules we can define appropriate test methods. However, due to their similarities to categorical rules and score-based rules, we introduce test methods for rule bases in Section 5.4.4.

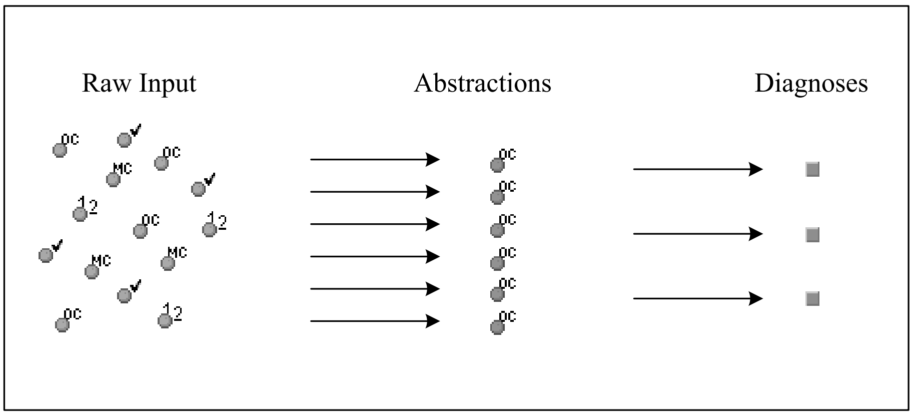


Figure 5.4: Three-layered architecture of a diagnostic knowledge system.

5.1.4. Restructuring Abstraction Knowledge

Abstraction knowledge mainly needs to consider the propagation of ontological restructurings. The ontological restructuring methods sketched in Section 4.6 are also propagated to the abstraction knowledge.

TRANSFORMMCINTOYN/ TRANSFORMYNINTOMC For the methods TRANSFORMMCINTOYN and TRANSFORMYNINTOMC the transformed objects are simply updated in the conditions and actions of the implemented abstraction rules or mapping functions. In general, no conflict will be produced for these two restructurings.

TRANSFORMNUMINTOOC Mapping functions can be simply updated according to the method TRANSFORMNUMINTOOC, since mapping functions commonly define a partition of the abstracted value range. However, if the restructured numerical question is contained in abstraction rules, then the method TRANSFORMNUMINTOOC is only executable, if the abstraction rules divide the numerical value range in distinct partitions. Otherwise, no direct transformation is possible, and the restructuring should be aborted.

TRANSFORMMCINTOOC The TRANSFORMMCINTOOC method similarly can cause conflicts, if the rule base contains abstraction rules with either multiple values of the specified multiple-choice question in their rule conditions, or multiple values of the specified multiple-choice question in their rule action. Then, the restructuring should be aborted and the conflict needs to be resolved manually.

SHRINKVALUERANGE The method SHRINKVALUERANGE simply exchanges the original values with the values of the reduced value range, if contained in a rule condition or rule action. The method causes a conflict, if syntactically ambivalent abstraction rules are generated due to the execution of the restructuring method, i.e., the method generates two rules with equal rule conditions, but different abstraction actions. For example, let us consider two rules

$$\begin{aligned} r_1 &= choiceEqual(Q, v_1) \rightarrow Q'{:}v' \,, \\ r_2 &= choiceEqual(Q, v_2) \rightarrow Q'{:}v'' \,. \end{aligned}$$

When shrinking the value range of question Q with a mapping $v_1 \rightarrow v_n$ and $v_2 \rightarrow v_n$, we obtain the following ambivalent rules

$$\begin{aligned} r_1 &= choiceEqual(Q, v_n) \rightarrow Q'{:}v' \,, \\ r_2 &= choiceEqual(Q, v_n) \rightarrow Q'{:}v'' \,. \end{aligned}$$

Then, the method is aborted and the rule base needs to be manually modified. Mapping functions are considered analogously.

MOVEQUESTIONVALUE For general rules we have to consider the following: If the moved value is contained in a rule condition, then the condition is modified, so that the new targeted question is contained in this condition. However, the restructuring can cause conflicts, if for example the new targeted question additionally is contained in the rule condition, and is assigned to a value different to the moved value. For one-choice questions this condition can never evaluate true, since an one-choice question is never assigned to two values at the same time. Then, the developer has to decide manually, which partial condition has to be removed from the condition. For abstraction rules we additionally have to consider the possibility that the moved value is contained in the abstraction action. Then, the action has to be modified so that the new targeted question is now assigned to the moved value.

INTRODUCEABSTRACTION For the method INTRODUCEABSTRACTION a corresponding abstraction rule needs to be inserted, which can be done automatically because of the specified abstraction condition. Then, all rules currently contained in the rule base are checked with respect to the new abstraction: If a rule condition contains the specified condition, then it is replaced by the newly created abstraction.

5.1.5. Summary

In this section, we introduced representational alternatives for abstraction knowledge and sketched acquisition approaches. Starting with mapping functions as a simple representation of abstraction knowledge, we increased the complexity in order to state even more complex abstraction knowledge, e.g., by score-based abstraction rules.

5.2. Case-Based Knowledge

The (re)use of previously experienced and solved cases is an appealing approach for developing diagnosis knowledge systems, since it often complies with the mental model of expert reasoning. The principle of case-based reasoning is that similar problems have similar solutions. Aamodt and Plaza [1] described CBR as follows:

> CBR is able to utilize the specific knowledge of previously experienced, concrete problem solutions (cases). A new problem is solved by finding a similar past case, and reusing it in the new problem situation.

For example, in the medical domain a physician commonly remembers old and similar cases, when he is examining a particular patient. If important findings are similar to findings of cases in the past, then often the diagnoses of these past cases are used to decide about the disease of the current patient.

5.2.1. Applications of Case-Based Knowledge

Cases can be applied for diagnostic reasoning quite simple. However, cases are relevant for many other aspects of diagnostic knowledge system development:

Use-case analysis: When starting knowledge system development, experts often tend to describe their problem domain by exemplary cases. These cases are experiences taken from their practice as well as imaginary cases illustrating particular circumstances of the application domain.

Case-based learning: Another important aspect of CBR (and distinction from other diagnostic methods) is the ability of continuous adjustment of existing structural knowledge and learning of new structural knowledge, since newly experienced cases are often integrated into the case base to be reused in future problem situations.

Semi-automatic learning If a sufficiently large number of cases is available, then these cases can be used to learn explicit structural knowledge. For example, in [14, 9] we presented semi-automatic methods for learning explicit structural knowledge from cases, e.g., set-covering models and scoring rules. We call these methods semi-automatic, since additional background knowledge can be applied to improve the learning results.

Test knowledge In the context of the agile process model, cases play a central role as test knowledge. Thus, solved cases can be used to validate the reasoning behavior of structural knowledge. We will discuss this in Section 5.6.1 in more detail.

However, in the following we will focus on the diagnostic reasoning using cases, i.e., case-based diagnosis.

5.2.2. The Case-Based Reasoning Framework

Aamodt and Plaza [1] have defined a common framework for CBR, which consists of a cyclic process model and a task-method decomposition model. The well-known cyclic process model is depicted in Figure 5.5, and describes the four basic steps of case-based reasoning: Retrieve, Reuse, Revise and Retain (also known as the ”4Re”s).

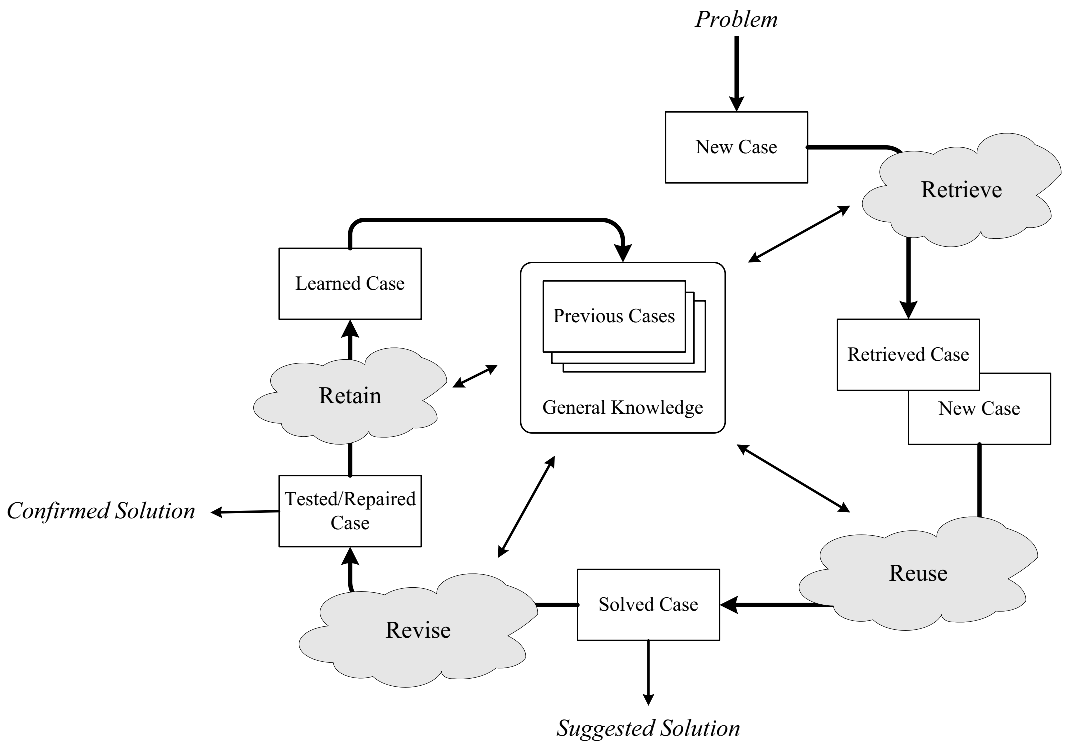


Figure 5.5: The cyclic CBR process model according to Aamodt & Plaza [1].

The four steps can be explained as follows: *Retrieve* is the process of finding the most similar cases for a new case (query) in a case base. *Reuse* is the process of applying the solutions of the found cases to the query. *Revise* is used to adjust/modify the applied solutions of the query, since the most similar cases may have led to an incorrect solution. *Retain* is the process of integrating the ”lessons learned” into the system. This is mostly done by simply adding the query and its correct solution into the existing case base. Sometimes this step is extended by maintenance operators, which additionally modify and improve the case base. It is worth noticing, that background knowledge can support the problem-solving process. Background knowledge depends on the specific application and may contain knowledge about similarities, weights or partitioning information.

In the following sections we will concentrate on a more formal definition of case retrieval. We refer to [1, 64] for a more detailed and technical description of the case-based reasoning steps.

5.2.3. Knowledge Representation

According to Richter [110] the case-based knowledge representation can be divided into four distinct *knowledge containers*:

- the vocabulary (terminological ontology), i.e, $\Omega_{\mathcal{D}}$ and $\Omega_{\mathcal{F}}$
- the similarity measures
- the case base, i.e., $CB \subseteq \Omega_C$
- the solution transformation

For diagnostic tasks commonly no knowledge about the solution transformation is necessary, since solutions from a retrieved case are often used with no adaptation. However, e.g., for design and planning tasks the solution transformation represents an important container. The similarity measure is mainly defined by a (local) similarity function comparing two findings, and a weight function stating the importance of single findings. Additionally, abstraction knowledge can be applied in order to facilitate an improved case comparison using high-valued findings.

Definition 5.2.1 (Local Similarity Function) A *local similarity function* is defined for a tuple of findings $F_1, F_2 \in \Omega_{\mathcal{F}}$ as follows:

$$sim : \Omega_{\mathcal{F}} \times \Omega_{\mathcal{F}} \to [0, 1] \,,$$

where the boundary value 0 means no similarity between the two findings and the value 1 indicates two equal findings.

If for two findings $F_1, F_2 \in \Omega_{\mathcal{F}}$ the similarity function is undefined, then the default similarity is applied: $sim_{def}(F_1, F_2) = 1$, if $F_1 = F_2$, and $sim_{def}(F_1, F_2) = 0$, otherwise.
Similarity functions have been thoroughly investigated in the past: For example, Goos [47] presents a well elaborated description of similarity functions according to the applied domain ontology (see Chapter 4). Richter [111] introduced a mathematical framework and basic concepts for classifying similarity measures. Bergmann and Stahl [21] investigated similarity measures from an object-oriented view.
In addition to the similarity function the expert can attach information about the importance of the implemented findings.

Definition 5.2.2 (Weight Functions) The *global weight function*

$$w_g : \Omega_Q \to I\!N_+$$

defines the absolute importance of a specified question $Q \in \Omega_Q$.
The *local weight function*

$$w_l : \Omega_{\mathcal{F}} \to I\!N_+$$

states the importance of a specified question $Q \in \Omega_Q$ assigned to a specified value $v \in dom(Q)$, i.e., the importance of a finding $Q{:}v \in \Omega_{\mathcal{F}}$. Then, higher values specify more important findings.

It is easy to see that the local weight function is a generalization of the global weight function, if for any $Q \in \Omega_Q$ it holds that $w_l(Q{:}v) = w_l(Q{:}v')$ for all $v, v' \in dom(Q)$.
In practical applications often a local weight function is defined using the product of an already existing global weight functions and an abnormality function. The abnormality function considers the possible values of each question. Then, the value range of a question Q is augmented with information about the abnormality state of the particular values $v \in dom(Q)$.

Definition 5.2.3 (Abnormality Function) The *abnormality function* is defined as follows

$$abn : \Omega_{val} \rightarrow [0, 1]\,,$$

and returns a real value between 0 and 1 denoting the abnormality state of the specified value. The boundary value 0 means a normal value of the question, and the boundary value 1 a totally abnormal state of the question.

For example, the question ”temperature” with value range $dom(temperature) = \{normal, increased, high\}$ defines the following abnormalities: $abn(normal) = 0$, $abn(increased) = 0.7$, and $abn(high) = 1$.
If no weight function is defined by the expert, then the *default weight function* is applied: $w_{def}(Q, D) = 1$ for all $Q \in \Omega_Q$ and $D \in \Omega_{\mathcal{D}}$.
In summary, we can see that case-based knowledge is represented by an appropriate case base with solved cases, and by a defined similarity measure, by a weight function, and by an optional abnormality function.

5.2.4. Knowledge Inference

For inferring a solution for a given problem, case-based diagnosis essentially follows the CBR-cycle defined in Section 5.2.2: After a set of sufficiently similar cases for the query has been retrieved, the cases are commonly reused by simply copying their solutions. Then, these solutions are presented as possible solutions for the query. The described inference structure is depicted in Figure 5.6 as a sequence diagram.
In this work, we will concentrate on the retrieve step, since it is the fundamental part of the case-base diagnosis task.
The retrieve step gathers cases from the case base, which are sufficiently similar to the query case. The similarity of a new case c with a retrieved case c' is defined by the (global) similarity function.

Definition 5.2.4 (Global Similarity Function) The *global similarity function* is defined for a tuple of cases

$$sim_g : \Omega_C \times \Omega_C \rightarrow [0, 1]\,.$$

For two totally similar cases $c, c' \in \Omega_C$ we obtain the similarity $sim_g(c, c') = 1$ and for two completely dissimilar cases we receive the similarity $sim_g(c, c') = 0$. Values between the boundary values 0 and 1 state the degree of similarity between the two cases.

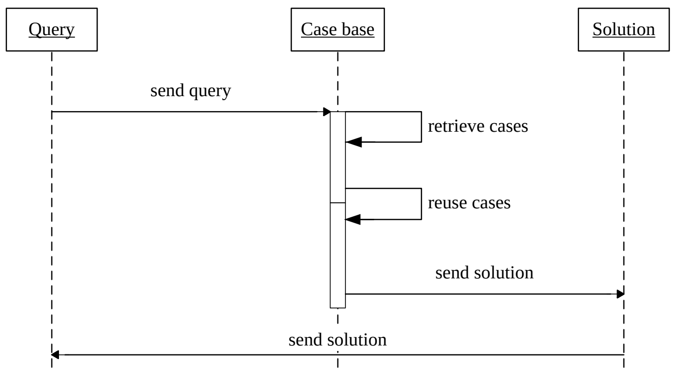


Figure 5.6: Sequence diagram of a case-based inference action.

A commonly used global similarity function is the *weighted Hamming distance*, which applies (global) weights and local similarities of findings. The weighted Hamming distance between a query cases $c \in \Omega_C$ and a retrieved case $c' \in CB$ is defined as follows:

$$sim_g^{Ham}(c, c') = \frac{\sum_{Q \in \Omega_Q} w_l\big(f(Q, \mathcal{F}_c)\big) \cdot sim\big(f(Q, \mathcal{F}_c), f(Q, \mathcal{F}_{c'})\big)}{w_l\big(f(Q, \mathcal{F}_c)\big)}, \qquad (5.1)$$

where $f(Q, \mathcal{F})$ is a function, which returns the corresponding finding of question Q in the specified set of findings $\mathcal{F}$.

For a more detailed discussion of similarity measures we refer to Richter [111]. In summary, the retrieve step tries to find a case $c' \in CB$ for a query case $c \in \Omega_C$, which maximizes the similarity $sim_g(c, c')$.

The efficiency of the linear retrieval of a maximum similar case evolves to become worse with an increasing size of the case base. Thus, we need to consider $\mathcal{O}(m \cdot n)$ comparisons for n cases, each case containing m findings. Therefore, several approaches have been proposed to cope with this problem: For example, in Goos [47] a *pre-selection strategy* for the cases contained in the case base is presented. Pre-selection can significantly reduce the costs of direct case comparison. Alternatively, an intuitive approach for reducing retrieval costs is the construction of *case clusters*, containing similar cases. For each cluster, a case representative is specified, which in a first retrieval is used for case comparison. In further steps, the most similar (and possibly hierarchically constructed) clusters are considered for an exhaustive case comparison. Furthermore, Lenz [66] introduced *case retrieval nets* as a specialized memory structure for efficient retrieval for suitable cases. Other approaches for case retrieval are, e.g., further indexing techniques [13] like k-d trees [135].

For a recent survey and assessment we refer to Richter's article [112] on case-based reasoning focussing the research on similarity measures. The work on similarity measures is described by introducing four periods: The naive period, sophistic period, systematic period, and the generalizing period. The resulting view on similarity of each period was motivated by additional requirements, which were not solved by the preceding period. Following Richter the similarity measure presented in the previous section can be classified to the sophistic period, since we do not only consider raw observations for the case comparison, but also high-level abstractions that allow for non-linear similarity computations.

5.2.5. Acquisition of Case-Based Knowledge

The use of case-based knowledge is appropriate if there are cases available for the targeted domain. However, if no or no usable cases are available, then the domain expert needs to manually generate a sufficient number of cases.

The minimal approach considers the construction of at least one *case representative* for each diagnosis. A case representative is a synthetical case that contains the typical findings for the described diagnosis. More than one representative for each diagnosis is required, if the diagnosis cannot be clearly described by one case, but is defined by a wider spectrum of symptoms.

Furthermore, a local similarity function (see Definition 5.2.1) needs to be defined since the default similarity sim_{def} is mostly not appropriate. Ideally, the local similarity function is manually defined by the domain expert. The function can be incrementally defined by first partitioning the value range of questions into normal and abnormal sections. The function is further refined by adding local weights for question values. Finally, the developer can explicitly define local similarity measures for question values.

Since an exhaustive definition of the similarity function denotes a complex and costly task, we alternatively provide methods for automatically learning the similarities and weights. For more details we refer to Section 5.2.8.

The case base may be enriched by additional cases gained by the usage of the system in a test or a productional environment: Any new case with a (possibly revised) solution is inserted into the case base. The acquisition of case-based knowledge is supported by restructuring methods, that for example propagate ontological restructurings or consider the structured insertion or deletion of cases. Furthermore, test methods for case-based knowledge check for anomalies (e.g., ambivalent cases, deficiency) and correctness.

5.2.6. Testing Case-Based Knowledge

Testing the quality of case-based reasoning systems has been undergoing fruitful research. The case-based approach states that the system continuously learns new knowledge by simply adding additional, previously solved cases to the case base.

Although these cases are assumed to be correct in general, there are two problems to consider: By inserting new cases also anomalies can be added to the case base. Furthermore, the cases may not cover the given domain ontology appropriately. In the following, we present methods for testing case-based knowledge for anomalies and ontology coverage.

It is worth noticing, that for the presented methods we have to distinguish between a case base of real cases and a case base constructed by exemplary cases. The second approach is applied, if the structural knowledge is defined from the scratch, using *case representatives* as described in Section 5.2.5.

Static Verification of the Case Base

The static verification method tries to detect anomalies in case based knowledge. Corresponding to the classification by Preece [93] we can distinguish between the following sub-categories of anomalies. Originally, Preece considered anomalies of rule-based systems but they can be easily transferred to the case-based representation.

Redundancy	Tests the case base for uniqueness and minimality of the contained cases.
Ambivalence	Determines the consistency of the cases contained in the case base.
Deficiency	Investigates, if the knowledge base contains diagnoses and findings that are not used by the cases of the case base.

As mentioned before, we have to consider the kind of included cases: Thus, only the detection of ambivalent and deficient case are important for all kinds of cases, whereas redundancy typically occurs in real life case bases. However, for a manually constructed case base containing case representatives the detection of redundant cases can be very interesting.

Iglezakis et al. [60, 108] introduced *quality measures* (case properties) for case-based maintenance, that can be directly applied to investigate a case base for the anomalies described above. Originally, the authors introduced these syntactical measures for the integration into an extended CBR cycle [109, 114].

Mechanics In the following, we show how these measures can be adapted for detecting redundancy and ambivalence in case-based knowledge. Additionally, we describe deficiencies in case bases.

► **Redundancy**	
Non-Uniqueness The case base violates the *uniqueness* criterion: There exist two cases $c, c' \in CB$ with equal problem descriptions and equal solutions, i.e. $\exists\, c, c' \in CB : \mathcal{F}_c = \mathcal{F}_{c'} \wedge \mathcal{D}_c = \mathcal{D}_{c'}$.	warning
Subsuming Cases The case base violates the *minimality* criterion: There exist two cases in the case base CB with equal solution, and the problem description of one case is subsumed by the problem description of the other case, i.e. $\exists\, c, c' \in CB : \mathcal{F}_c \subset \mathcal{F}_{c'} \wedge \mathcal{D}_c = \mathcal{D}_{c'}$.	warning
Coherent Cases The case base CB contains *coherent* cases, if for a given threshold $\mathcal{T}_{coh}$ $\exists\, c, c' \in CB : sim_g(c, c') > \mathcal{T}_{coh} \;\wedge\; \mathcal{D}_c = \mathcal{D}_{c'}$.	warning

► **Ambivalence**	
The case base violates the *consistency* criterion: There exist two cases in the case base CB with a subsuming problem description but different solutions, i.e., $\exists\, c, c' \in CB : \mathcal{F}_c \subseteq \mathcal{F}_{c'} \wedge \mathcal{D}_c \neq \mathcal{D}_{c'}$ or $\exists\, c, c' \in CB : sim_g(c, c') > \mathcal{T}_{amb} \wedge \mathcal{D}_c \neq \mathcal{D}_{c'}$	warning

► **Deficiency**	
There exists a deficiency, if the ontological knowledge contains diagnoses, that do not appear in any case of the case base.	warning

Usage With the presented static verification method *anomalies* can be detected. Then, a growing case base can be systematically investigated according to its redundancy, ambivalence or deficiency. If the case base consists of real cases, then we only have to consider the detection of ambivalent and deficient cases since redundant case are often appear in real-world environments.

Possible actions for redundant or ambivalent cases are the refinement or the deletion of a corresponding case. If deficiency is detected in the case base, then it is advisable to consider the insertion of a new and appropriate case.

For the evaluation of the method no additional test knowledge is required, with exception to the threshold $\mathcal{T}_{coh}$ for defining coherent cases.

Case Base Structure Testing

With the *case base structure testing* method an overview of the current case base is given according to the appearance of diagnoses and questions.

Mechanics The method generates a statistics of the diagnoses and questions contained in the case base. Thus, the total numbers and percentage with respect to the complete case base of contained diagnoses and questions is counted.

The statistics reports the following irregularities as a warning:

- Diagnoses with an exceptionally high or low usage (compared to the mean usage) in the case base.
- Questions with an exceptionally high or low usage (compared to the mean usage) in the case base.

Additionally, the statistics of the remaining diagnoses and questions is presented as information on user request.

Usage For the execution of the method no additional test knowledge is required. Significant high or low usage of objects can give hints for adding new cases to the case base or removing existing cases.

5.2.7. Restructuring Case-Based Knowledge

In this section, we briefly discuss restructuring methods for the case-based knowledge container. Most of the methods are triggered by corresponding methods originally executed by the ontological knowledge container.

Since cases additionally are often used as test knowledge the presented methods can be also used for updating test cases according to preceding restructuring methods.

Propagation of Ontological Restructuring Methods

As introduced in the previous chapter ontological restructuring methods propagate themselves to the remaining knowledge containers, since ontological changes often have an effect on other knowledge.

Thus, the following methods are simply propagated by exchanging the transformed object with the old one, no conflicts are expected to occur during operation: TRANSFORMMCINTOYN, TRANSFORMYNINTOMC, and TRANSFORMNUMINTOOC.

The method TRANSFORMMCINTOOC may cause conflicts, if in a case the transformed one-choice question is assigned to more than one value. Then, this method should be aborted and the case need to be manually edited. The method SHRINKVALUERANGE is executed by simply exchanging the choice answers of the specified question according to user-defined transformation matrix.

The MOVEQUESTIONVALUE method is applied by removing the old question, if assigned to the moved value, and by inserting the new targeted question assigned to the moved value into the case. However, this restructuring can cause conflicts, if the targeted question is already contained in the case, but assigned to a different value. Then, the developer has to decide, probably by defaults, which finding is removed from the case.

With the execution of the method REMOVEQUESTION the corresponding finding of the specified question is removed from all cases in the case base. The following conflicts may arise: The method may yield an empty observation set $\mathcal{F}_c$ for a case c, or the method may produce redundancy or ambivalence contained in the case base. Therefore, static verification for case-based knowledge (p. 77) needs to be applied after execution of the restructuring method.

The REMOVEDIAGNOSIS method is executed by removing the specified diagnosis from all cases and their solution parts, respectively. Analogously to the execution of the REMOVEQUESTION method this can cause either an empty case solution or an ambivalent or redundant case base. Hence, static verification for case-based knowledge also needs to be applied after method execution.

Restructuring of the Case Base: INSERTCASE and REMOVECASE

For restructuring the case base we can distinguish two basic methods: The INSERTCASE and the REMOVECASE method.

Both methods simply insert a new case into the case base or remove an existing case from the case base. Although cases have no dependencies between each other and can be considered as isolated knowledge structures, the insertion or deletion of a case can change the overall properties of a case base.

Especially, the deletion of cases have to be performed carefully, since, e.g., *pivotal cases* can reduce the competence of the case base significantly. According to Smyth and Keane [123] a pivotal case describes a problem area that is currently only solvable using this pivotal case. A deletion of the case therefore would yield a decreased competence of the case base. The absence of a pivotal case can be simply detected by testing the case base for deficiency.

5.2.8. Learning Case-Based Knowledge

We can consider three parts for learning case-based knowledge: the case base, the similarity measure, and the weight function. In the following, we discuss these in more detail.

The Case Base

In general, we distinguish two typical situations: The case base is either constructed manually by acquiring case representatives, or the real cases are available, e.g., gathered by an already installed documentation system.

Once the system is in routine use, any query solved by the running knowledge system is also added to the case base. This growth of the case base can be interpreted as an ”automatic” learning of the system. However, with this naive approach the case base is continuously growing and becomes very large. For case-based reasoning this *swamping problem* was reported, e.g., by Francis and Ram [43] .

As mentioned before, a large case base can decrease the velocity of the case retrieval. Besides techniques for improving the retrieval step as sketched in Section 5.2.4, many researchers have proposed to apply additional maintenance steps after adding a number of new, solved cases to the working case base. This is especially useful, if the case base was constructed manually with case representatives.

For example, Smyth and Keane [123] propose a deletion policy, which removes cases from the case base while preserving its reasoning competence. The competence of a case base is based on the key concepts *coverage* and *reachability*: The coverage of a case refers to the set of cases in the case base that solve this case. The reachability of a case inversely describes the set of cases, that can be applied as a solution for this case.

Zhu and Yang [139] alternatively presented an approach based on adding existing cases to an initially empty case base until a given degree of competence is reached by the case base. In contrast to the deletion policy, a lower bound for the coverage of the case base can be guaranteed.

Learning the Similarity Measure

If no similarity function is available, then the default similarity in Equation 5.2 is applied.

$$sim_{def}(F_1, F_2) = \begin{cases} 1 & \text{if } F_1 = F_2\,, \\ 0 & \text{otherwise,} \end{cases} \tag{5.2}$$

where $F_1, F_2 \in \Omega_{\mathcal{F}}$. However, this boolean comparison of finding values will not provide satisfiable results for many applications.

For this reason, the developer usually has to manually define similarity measures for findings. To reduce this complex and costly task, (semi-)automatic methods can be applied, which try to learn accurate similarities. We therefore introduce an approach initially presented in Baumeister et al. [14].

Learning a similarity function is often presented by learning a corresponding *distance function*. In contrast to a similarity function a distance function does not describe the closeness of two findings, but inversely denotes the dissimilarity of the findings. Formally, a distance function is defined by

$$dist : \Omega_{\mathcal{F}} \times \Omega_{\mathcal{F}} \to [0, \infty)\,. \tag{5.3}$$

For practical reasons often the possible range of distances is restricted to the real interval $[0, 1]$. Given a distance function $dist$ the similarity of two findings $F_1, F_2 \in \Omega_{\mathcal{F}}$ can be computed according to Richter [111] as follows

$$sim(F_1, F_2) = 1 - \frac{dist(F_1, F_2)}{1 + dist(F_1, F_2)} = \frac{1}{1 + dist(F_1, F_2)}\,. \tag{5.4}$$

For findings with bounded distance range, i.e., $dist : \Omega_{\mathcal{F}} \times \Omega_{\mathcal{F}} \to [0, K]$, an alternative relation can be defined according to [111, 99]:

$$sim(F_1, F_2) = 1 - \frac{dist(F_1, F_2)}{K}\,, \tag{5.5}$$

where K is maximum constant. In the following, we present an approach for learning distance functions for arbitrary finding types, and then we will refine this approach by methods that apply additional domain knowledge.

General Approach for Learning Distance Functions When learning distance functions we have to distinguish between the different question types.

For numerical questions there exist well-known standard distance functions, like the *Euclidean* distance function, the *Manhattan* distance function, or the *Minkowski* distance function. A discussion of various kinds of distance measures can be found, e.g., in [52].

For findings assigned from choice questions we propose the use of the *Value Distance Metric* (VDM) introduced by Stanfill and Waltz [127], and improved by Wilson and Martinez [137]. Given two findings $F_1, F_2 \in \Omega_{\mathcal{F}}$ the VDM defines their distance as follows:

$$vdm(F_1, F_2) = \frac{1}{|\Omega_{\mathcal{D}}|} \cdot \sum_{D \in \Omega_{\mathcal{D}}} \left| \frac{N(F_1|D)}{N(F_1)} - \frac{N(F_2|D)}{N(F_2)} \right|, \tag{5.6}$$

where $N(F)$ is the number of cases $c \in CB$, for which the finding F is observed, i.e., $F \in \mathcal{F}_c$, and $N(F|D)$ is the number of cases $c \in CB$, in which F is observed and D is in the solution part of c, i.e., $F \in \mathcal{F}_c \wedge D \in \mathcal{D}_c$.

Then, given the value distance metric two findings are considered to be more similar, if they have more similar correlations with the diagnoses they occur with.

In general, we propose the following distance function for arbitrary finding types. For simplicity we assume that findings $F_1, F_2 \in \Omega_{\mathcal{F}}$ are assigned to the same question $Q \in \Omega_Q$, i.e., $a(F_1) = a(F_2) = Q$.

$$dist(F_1, F_2) = \begin{cases} vdm(F_1, F_2) & \text{if } Q \text{ choice question,} \\ vdm\big(discrete(F_1), discrete(F_2)\big) & \text{if } Q \text{ numerical question.} \end{cases} \tag{5.7}$$

We propose to (temporarily) discretize numerical findings for obtaining their distance, e.g., neighbouring numerical values may have no distance between.

Figure 5.7 summarizes the applied methods for learning similarities for the different kinds of questions.

Question	**Method**
Choice	For a choice question Q we fill the similarity matrix for the values $dom(Q)$ of the question by simply computing the distance between all values $v, v' \in dom(Q)$ as defined in Equation 5.7, and then transform the calculated distance to a similarity as defined in Equations 5.4 and 5.5.
Yes/No	As a special case for yes/no questions we simply apply the default similarity sim_{def} as given in Equation 5.2.
Numerical	Numerical questions are temporarily discretized for the comparison and the same method as for choice questions is applied.

Figure 5.7: Summary of learning similarity knowledge for different types of questions.

Improving Learning with Abnormality Information For many application domains it is possible to define abnormality information for finding values of choice questions. Then, each value of a choice question is attached with a label that explains, if the value is describing a normal or an abnormal state of the question. The abnormality function for finding values was introduced in Definition 5.2.3 (p. 74).

For example, consider the choice question ”temperature” with the value range dom*(temperature)* = {*normal, marginal, high, very high*}: The values {*normal, marginal*} denote normal values of the question, and the values {*high, very high*} describe abnormal states of the question.

Practical applications have shown that in contrast to a real valued categorization as shown in Definition 5.2.3, a symbolic categorization of abnormal values is simpler to understand for users. Thus, we can define the following abnormality function for finding values:

$$abn : \Omega_{val} \rightarrow \Omega_{Abn} \,, \tag{5.8}$$

where $\Omega_{Abn} = \{A0, A1, A2, A3, A4, A5\}$ is defined as a set of symbols describing the abnormalities in ascending order; with $A0$ representing the normal state and $A5$ representing the most abnormal state.

Using this abnormality information we can divide the value range of a choice question into two partitions containing normal and abnormal values, respectively. Since it is very unlikely that normal values are similar to abnormal values, we can define a fixed distance function between normal and abnormal values. An exemplary distance function is given in Equation 5.9 for a normal finding value $F = Q{:}v$ and an abnormal value $F' = Q{:}v'$, i.e., $abn(v) = A0$ and $abn(v') \in \{A1, \ldots, A5\}$.

$$dist_{abn}(F, F') = \begin{cases} 0.6 & \text{if } abn(v') = A1\,, \\ 0.7 & \text{if } abn(v') = A2\,, \\ 0.8 & \text{if } abn(v') = A3\,, \\ 0.9 & \text{if } abn(v') = A4\,, \\ 1.0 & \text{if } abn(v') = A5\,. \end{cases} \tag{5.9}$$

Using this distance function for comparing an abnormal and a normal finding we obtain a maximum distance between a normal and a totally abnormal finding, i.e., $dist(Q{:}v, Q{:}v') = 1$, for $abn(v) = A0$ and $dist(v') = A5$.

The remaining similarities, i.e., similarities between normal findings and similarities comparing abnormal findings, are computed according to the general method given in Equation 5.7.

Interpolating Distances with Scalability Information Beyond abnormalities the expert may mark some of the parameters as *scaled* to characterize, that values, that are closer to each other, are more similar.

For example, we consider a question ”temperature” with the value range $dom(temperature) = \{normal, increased, high, very\,high\}$, which is scaled, whereas the question *color* with value range $dom(color) = \{green, black, red\}$ is not scaled.

We can utilize this flag by applying the general VDM method given in Equation 5.7 not for all distinct pairs of values within each partition, but only for adjacent values. We interpolate the remaining distances by the following equation

$$dist(Q{:}v_i, Q{:}v_{i+k}) = dist(Q{:}v_i, Q{:}v_{i+k-1}) + dist(Q{:}v_{i+k-1}, Q{:}v_{i+k})\,, \tag{5.10}$$

where $k \geq 2$ and $v_j \in dom(Q)$. After interpolating the remaining distances we have to normalize the whole distance matrix for each scaled question Q, so that for all values $v, v' \in dom(Q)$ it holds that $0 \leq dist(Q{:}v, Q{:}v') \leq 1$.

Learning Weights for Questions

As described in Definition 5.2.2 weights assign positive integers to questions or findings. For learning weights we only consider the global weight function w_g, which assigns positive integers to questions. Initially, if no knowledge about weights is available, then the (global) weight for all questions is equal, e.g., $w_g(Q) = 1$ for all $Q \in \Omega_Q$.

We simplify the interpretation of the learned weights by introducing predefined symbols for weights. Thus, we propose a fixed symbol range for weights $\Omega_{\mathcal{W}} = \{G0, G1, G2, G3, G4, G5, G6, G7\}$, which contains weights in ascending order. A question attached with weight $G0$ has no importance, and a question with weight $G7$ has the highest importance. It is easy to see that the symbolic weights can be easily transferred to positive integers as required by Definition 5.2.2. However, if the learned weights are presented as symbols, then the (manual) interpretation and adaptation of the results by the developer is quite simpler than a presentation of integer values. We initially presented the following approach for learning weights in [14].

General Approach for Learning Global Weights Our approach is inspired by a procedure mentioned in [136], when using the VDM method (see Equation 5.7) to discriminate the importance of findings. However, our interpretation also tries to integrate additional background knowledge like abnormalities and structural knowledge.

The general idea of the approach is as follows: A question Q is defined to be important, if Q has a high *selectivity* over the diagnoses contained in the case base CB, which is applied for the learning task. The degree of selectivity directly corresponds to the importance (weight) of the question. Thus, if different values of a question Q indicate different diagnoses, then the question is considered to be *selective* for the diagnostic process.

We define the *partial selectivity* of a question $Q \in \Omega_Q$ combined with a diagnosis $D \in \Omega_{\mathcal{D}}$ by the equation

$$sel(Q, D) = \frac{\sum\limits_{v,v' \in dom'(Q)} \left| \frac{N(Q{:}v \mid D)}{N(Q{:}v)} - \frac{N(Q{:}v' \mid D)}{N(Q{:}v')} \right|}{\binom{|dom'(Q)|}{2}} \tag{5.11}$$

where $v \neq v'$ and $dom'(Q) \subseteq dom(Q)$ contains only values, that actually occur in cases $c \in CB$.

To compute the global *selectivity* of a question Q, we average the partial selectivities $sel(Q, D)$

$$sel(Q) = \frac{\sum\limits_{D \in D^Q_{rel}} sel(Q, D)}{|D^Q_{rel}|}\,, \tag{5.12}$$

where

$$D^Q_{rel} = \left\{ D \in \Omega_{\mathcal{D}} \;\middle|\; \exists\, Q \in \Omega_Q, v \in dom(Q) : \frac{N(Q{:}v|D)}{|CB|} > \mathcal{T}_w \right\}.$$

We only investigate the selectivities between questions and diagnoses, whose combined frequency is larger than a given threshold $\mathcal{T}_w$. Since $sel(Q,D) \in [0,1]$ for all diagnoses $D \in D^Q_{rel}$ and all questions $Q \in \Omega_Q$, we see that $sel(Q) \in [0,1]$ for all questions $Q \in \Omega_Q$. The lower bound 0 is obtained, if question Q has no selectivity over the diagnoses contained in $\Omega_{\mathcal{D}}$; the upper bound 1 is obtained, if Q has a perfect selectivity over the diagnoses contained in $\Omega_{\mathcal{D}}$, i.e., each value $v \in dom(Q)$ occurs either always or never with the diagnosis.

After determining the selectivity of each question, we use the logarithmic conversion table depicted in Figure 5.8 to transform the numerical selectivity into a symbolic weight.

$sel(Q)$		$w_g(Q)$	$sel(Q)$		$w_g(Q)$
0	$\rightarrow$	G0	(0.08, 0.16]	$\rightarrow$	G4
(0, 0.02]	$\rightarrow$	G1	(0.16, 0.32]	$\rightarrow$	G5
(0.02, 0.04]	$\rightarrow$	G2	(0.32, 0.64]	$\rightarrow$	G6
(0.04, 0.08]	$\rightarrow$	G3	(0.64, 1.00]	$\rightarrow$	G7

Figure 5.8: Transformation table for converting numerical selectivities into symbolic weights.

As discussed above we accept the loss of information to facilitate a user-friendly adaptation of the learned weights by the developer in a subsequent step.

Utilizing Abnormalities for Learning Weights Abnormalities were defined above for improving the learning method for similarities, and can also be applied for learning abnormalities.

If there are abnormalities available for a given question Q, then we will adapt Equation 5.11 to consider only the selectivity between normal and abnormal question values.

$$sel(Q,D) = \frac{\sum\limits_{v \in abnormal(Q) \,\wedge\, v' \in normal(Q)} \left| \frac{N(Q{:}v \,|\, D)}{N(Q{:}v)} - \frac{N(Q{:}v' \,|\, D)}{N(Q{:}v')} \right|}{|\, abnormal(Q)\,| \cdot |\, normal(Q)\,|}\,, \tag{5.13}$$

where $abnormal(Q) = \{\, v \in dom(Q) \,|\, abn(v) \neq A0 \,\}$ is the set of values $v \in dom(Q)$ representing an abnormal state, and $normal(Q) = dom(Q) \setminus abnormal(Q)$.

Optimizing Parameter Weights by Ontological Knowledge If the knowledge base is highly structured, i.e., questions are structured by meaningful question sets, then we can use this knowledge for refining the learned weights. Question sets can contain an *examination weight* to mark their significance in the overall diagnostic process. These weights help to adjust the weights of the questions contained in the question set. Therefore, questions contained in dense question sets (i.e., containing many questions) will receive a decreased weight, whereas questions contained in sparse question sets with fewer questions will obtain an increased weight.

For a question Q contained in question set QS we will obtain an adjusted weight $w'_g(Q)$ defined by Equation 5.14.

$$w'_g(Q) = \frac{w_g(Q)}{\sum\limits_{Q' \in QS} w_g(Q')} \cdot w(QS) \tag{5.14}$$

where $w(QS)$ is the *examination weight* for question set QS (e.g., defined by a domain expert). The heuristic given in Equation 5.14 is motivated by the fact that in many (medical) domains single phenomena are structured in single question sets (examinations). Thus, if the examination contains many questions describing the phenomenon, then this examination is likely to contribute more weights than an examination with fewer questions. Nevertheless, each examination only describes one phenomenon. It is worth noticing, that this method is not reasonable in general, but can be applied, if a highly structured case base is available.

The presented methods were evaluated in [14] using a case base gathered from a real world application. In summary, the evaluation of the methods showed that the usage of the described background knowledge improves the learning results.

5.2.9. Summary

In the previous section, we have briefly introduced the application of case-based reasoning for diagnostic tasks. This approach often fits the mental model of the experts and can be easily applied by initially creating a case base. Often a case base is available and additional knowledge acquisition can be limited to the definition of appropriate similarity and weight measures. However, the gathering of meaningful measures can evolve to be a difficult and complex task, if the domain is large but only partly understood. Therefore, we described (semi-)automatic learning methods for similarity measures and weights. These methods apply additional background knowledge for improving the learning results.

Besides learning case-based knowledge we additionally presented testing and restructuring methods that are appropriate for the case-based knowledge container.

5.3. Categorical Knowledge

In well-understood domains often certain knowledge for determining a solution can be stated. Thus, domain experts commonly have experience of findings that, if observed, categorically point to a particular diagnosis. For this kind of knowledge categorical rules are the most efficient way to represent structural knowledge.

5.3.1. Categorical Knowledge Representation

The most intuitive way for representing categorical knowledge are categorical rules.

Definition 5.3.1 (Categorical Rule) A *categorical rule* r is denoted as follows:

$$r = cond(r) \rightarrow D \quad [except(r), context(r)]\,,$$

where $cond(r)$ is a rule condition containing disjunctions and/or conjunctions of arbitrary findings $F \in \Omega_{\mathcal{F}}$ and $D \in \Omega_{\mathcal{D}}$ is the targeted diagnosis. Optionally, a categorical rule can contain a rule exception $except(r)$ and a rule context $context(r)$.

The semantics of the rule condition, the rule exception, and the rule context were described earlier in Section 5.1.1 (p. 63).
If the rule fires, then the state of the diagnosis specified in the rule action is set to *probable*, i.e., the diagnosis is established.

5.3.2. Inference of Categorical Knowledge

Knowledge inference of categorical knowledge is very simple: For a new finding observed in a case, the categorical rule base is evaluated. If the rule condition of a currently inactive rule evaluates true and the rule exception and context also allows to fire, then the rule action is activated, i.e., the specified diagnosis is established. An exemplary inference action of categorical rules is depicted in Figure 5.9.
A new observation can cause a rule r to be drawn back, e.g., if the rule condition $cond(r)$ evaluates false, but has evaluated true before. Then, the action of r has to be drawn back by de-establishing the specified diagnosis. This in turn can cause other rules to be drawn back again.

5.3.3. Acquisition of Categorical Knowledge

Categorical knowledge is commonly acquired by diagnostic rules. As their main benefit categorical rules do not need to be considered in the context of other rules, since they are self-contained, i.e., each rule itself contains the necessary knowledge for deriving a specified diagnosis. This characteristics distinguish them from, e.g., scoring rules, that usually only add a specified certainty to a diagnosis.
For a more structured approach of rule acquisition we propose a specialized rule representation, the *categorical decision tables*. The benefit of applying a structured approach is

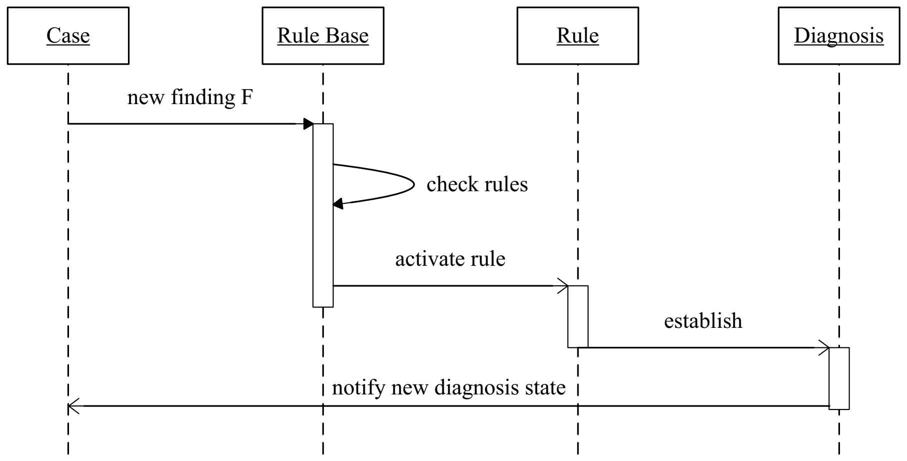


Figure 5.9: Sequence diagram of a categorical rule action.

that specialized editors can be utilized. Furthermore, experience has shown, that structured approaches significantly increase the clarity and maintainability of the rule base. We will discuss the approach in the following.

Categorical Decision Table If the confirmation of a diagnosis can be defined by the observation of a set of findings, then the *categorical decision table* can be applied, which describes a logical combination of a set of observed findings. A diagnosis is established, if a conditioned set of findings is observed; this set is combined by an *and* or an *or* top-condition.

Decision Table	r_1: Diagnosis D_1	r_2: Diagnosis D_2	r_3: Diagnosis D_3
Top-Condition:	*and*	*and*	*or*
choiceEqual(Q_1, v_1)	☒	☒	
choiceEqual(Q_2, v_2)		☒	
choiceEqual(Q_3, v_3)			☒
choiceEqual(Q_4, v_4)	☒		☒

Figure 5.10: A categorical decision table.

Figure 5.10 depicts an example of a categorical decision table. Given this table, diagnosis D_1 will be established by rule r_1, if the findings Q_1:v_1 and Q_4:v_4 are observed. Furthermore, the state of diagnosis D_3 will be probable, i.e., established by rule r_3, if the finding Q_3:v_3 or the finding Q_4:v_4 is observed ($D_i \in \Omega_Q, Q_i{:}v_i \in \Omega_{\mathcal{F}}$).

The categorical decision table can be extended by categorical rules represented in conjunctive or disjunctive normal form. Figure 5.11 depicts an example of an extended categorical decision table.

The table depicts the rule r_1 in conjunctive normal form and the rule r_2 represented in disjunctive normal form.

Decision Table	r_1: Diagnosis D_1		r_2: Diagnosis D_2	
Top-Condition:	*and*	*or*	*or*	*and*
choiceEqual(Q_1, v_1)		☒		☒
choiceEqual(Q_2, v_2)		☒		
choiceEqual(Q_3, v_3)				☒
		or		*and*
choiceEqual(Q_4, v_4)		☒		
choiceEqual(Q_5, v_5)				☒
choiceEqual(Q_6, v_6)		☒		☒

Figure 5.11: An extended categorical decision table.

Rules using the disjunctive normal form can sometimes be abbreviated by so-called *min/max-conditions*. Then, the condition of a rule r

$$cond(r) = min/max_{[2,3]}(c_1, c_2, c_3) \quad (c_i \text{ are sub-conditions})$$

states, that either two or three of the specified sub-conditions c_i need to be true, i.e.,

$$cond(r) = or\big(and(c_1, c_2), and(c_1, c_3), and(c_2, c_3)\big)\ .$$

Experience has shown that even more complex rule conditions are not suggestive, since the application of rules with arbitrary complex rule conditions yields unmaintainable and hardly interpretable knowledge bases.

Decision Trees The categorical decision table can be augmented by strategic knowledge for building *decision trees*. Then, conditions are represented as inner nodes of the tree and leafs of the trees are used for establishing diagnoses (with special diagnosis (–) for ”no solution”). In Figure 5.12 a decision tree is depicted using the same inferential knowledge presented in Figure 5.10. We can see, that firstly finding Q_1:v_1 is questioned. If Q_1:v_1 is observed, then the finding Q_2:v_2 is asked. If this finding is also observed, then the diagnosis D_2 is established. For this procedure, the right outer path of the decision tree is traversed. The other paths are traversed analogously. As mentioned before, decision trees embody strategic knowledge for controlling the dialog with the user. We will focus on this type of knowledge in Chapter 6 in more detail.

5.3.4. Testing and Restructuring Categorical Knowledge

For the agile development of knowledge systems using categorical knowledge it is also important to consider testing and restructuring methods. However, due to their similarities applied to categorical rules and score-based rules we present testing and restructuring methods together for both representations in Section 5.4.4 (p. 95).

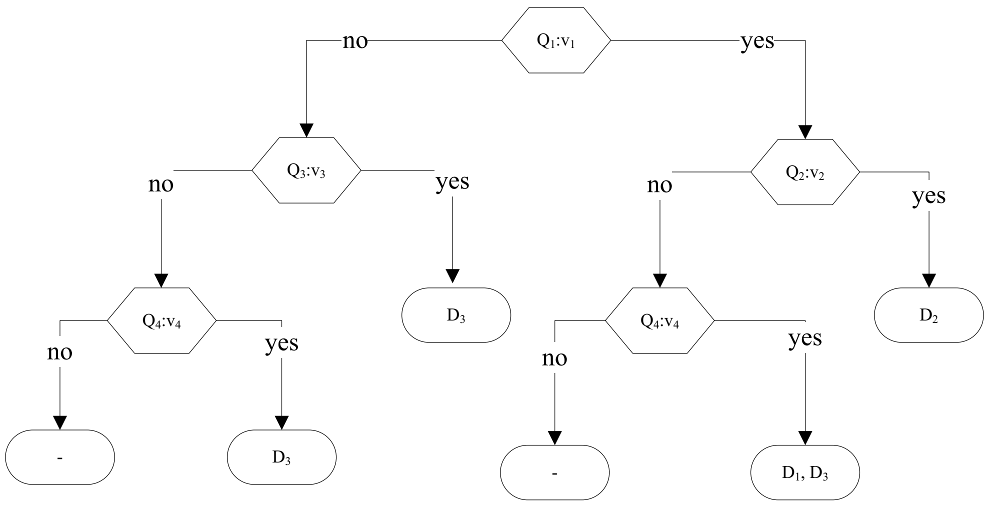


Figure 5.12: A simple example for a categorical decision tree.

5.3.5. Summary

In this section, we introduced a categorical representation for defining simple diagnostic knowledge. Categorical knowledge is suitable, if diagnoses can be inferred with certain knowledge.

Usually, categorical knowledge is represented by *categorical rules* inferring diagnoses. For the structured acquisition of categorical knowledge we presented the approaches *categorical decision tables* and *decision trees*.

5.4. Score-Based Knowledge

The preceding section introduced a rule-based formalism to capture categorical knowledge, which is able to solve simple diagnostic problems but fails for uncertain knowledge. In this section, we present a score-based formalism, which uses a rule-based representation augmented with *confirmation categories* to describe a symbolic uncertainty of the stated implication.

Scores are a well-known concept for diagnostic reasoning in medical decision making. For each diagnosis an account (score) is used for inferring the state of this diagnosis. In its simplest form, any observed finding can contribute to the score of a specified diagnosis. Then, the state of the diagnosis is determined by given threshold values. In its general form, not only isolated observations of findings can contribute to a diagnosis score, but also conditioned observations among findings. Rule-based approaches for implementing structural knowledge with uncertainty were mainly influenced by the work of the MYCIN project [24], and have been undergoing fruitful research for the last decades. For example, relations to Bayesian networks [88] were investigated by Heckerman [55].

Score-based approaches using rules go back to the INTERNIST/QMR project [76, 53]. Many researchers have adopted the ideas of the score-based representation and build (partially) successful systems. The LEXMED system [116] is a recent example of a successful medical application using scores within the PIT system. Other recent medical applications using scores are described in [39, 85].

5.4.1. Knowledge Representation

Score-based knowledge [99] usually is formulated by scoring rules, which are a specialization of general rules.

Definition 5.4.1 (Scoring Rule) A *scoring rule* r is denoted as follows:

$$r = cond(r) \xrightarrow{s} D \quad [except(r), context(r)]\,,$$

where $cond(r)$ is the rule condition of rule r, and $D \in \Omega_{\mathcal{D}}$ is the targeted diagnosis. For each rule a confirmation category $s \in \Omega_{scr}$ is attached with

$$\Omega_{scr} \in \{\, N7, N6, \ldots, N1, 0, P1, P2, \ldots, P7 \,\}\,.$$

Optionally, a rule exception $except(r)$ and a rule context $context(r)$ can be defined for a scoring rule r.

The semantics of the rule condition, rule exception, and the rule context were described earlier in Section 5.1.1. The confirmation categories are used to represent a qualitative degree of uncertainty. In contrast to quantitative approaches, e.g., Bayesian methods, symbolic categories state the degree of confirmation or disconfirmation for diagnoses. A category s expresses the uncertainty for which the observation of the findings in $cond(r)$ will confirm/disconfirm the diagnosis D. Whereas $s \in \{P1, \ldots, P7\}$ stand for confirming categories in ascending order, the symbols $s \in \{N1, \ldots, N7\}$ are ascending categories for disconfirming a diagnosis. A scoring rule with confirmation category 0 has no effect on the diagnosis' state, and therefore is usually omitted from the rule base. It is worth noticing, that the value range Ω_{scr} of the possible confirmation categories is not fixed. For a more detailed (or coarse) representation of confirmation the value range may be extended (or shortened). E.g., the PIT system simply uses integers. However, many projects applying score-based knowledge have shown the applicability of the presented detail of the value range, see e.g., [28, 106, 100].

In summary, using a score-based representation the structural knowledge container can be defined by a rule base $\mathcal{R}$ consisting of scoring rules as introduced above.

5.4.2. Knowledge Inference

Score-based knowledge is applied by evaluating the available rules with respect to the given observations. Then, if for a case c a new finding $F \in \Omega_{\mathcal{F}}$ is observed or the state of a diagnosis $D \in \Omega_{\mathcal{D}}$ has changed, then the rule base $\mathcal{R}$ is interpreted by evaluating each rule $r \in \mathcal{R}$ with the given observation: If the condition of a rule $r \in \mathcal{R}$ evaluates true (e.g., the conditioned findings are assigned to the postulated values), then the rule fires by executing the rule action: The specified category is attached to the diagnosis score. The final score of each diagnosis is determined by aggregating the attached categories in a predefined manner. According to the score and the threshold values the status of the current diagnosis is determined.

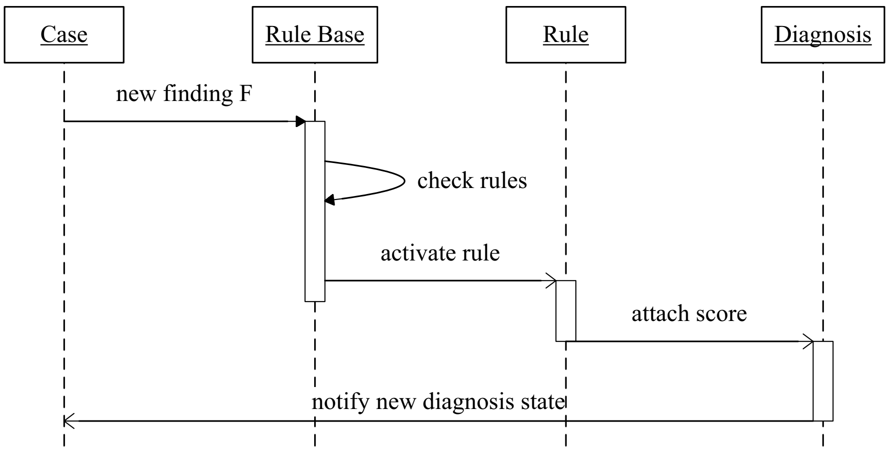


Figure 5.13: Sequence diagram of a score-based inference action.

The described inference structure is depicted in Figure 5.13. A new observation can cause a rule r to be drawn back, if e.g., the rule condition $cond(r)$ evaluates false, but has been evaluated true before. Then, the action of r has to be drawn back, by withdrawing the rules' confirmation category from the diagnosis and re-aggregating the diagnosis score. This in turn can cause other rules to be drawn back again, e.g., if the diagnosis is de-established due to the actually missing score threshold.

For the D3 system and its successor d3web the following aggregation procedure and threshold values have been proven to be suitable.

Aggregation of Scores In general, the score of a diagnosis $D \in \Omega_{\mathcal{D}}$ is calculated by summing the attached confirmation categories of D. To simplify the computation, the categories are translated to integer values by the function defined in Figure 5.14.

Rule Transformation				
Category	Integer		Category	Integer
0	0			
$N1$	-2		$P1$	2
$N2$	-5		$P2$	5
$N3$	-10		$P3$	10
$N4$	-20		$P4$	20
$N5$	-40		$P5$	40
$N6$	-80		$P6$	80
$N7$	$-INF$		$P7$	999

Figure 5.14: A simple transformation table for confirmation categories to integer values.

We can see, that the meaning of the categories is defined in a way, so that the aggregation of two equal categories result in a category of the next higher level (with exception of the boundary values $N7$ and $P7$).

Evaluating the Diagnosis Score For the evaluation of the diagnosis score the integers are summed to a final result, which is then translated to a symbolic category. To determine the diagnosis state

$$dom(D) = \{not\ probable, unclear, suggested, probable\}$$

of a diagnosis D the following thresholds given in Figure 5.15 are utilized.

State	**Threshold**
$not\ probable$	$< N5$
$unclear$	$in[N5, P3)$
$suggested$	$in[P3, P5]$
$probable$	$> P5$

Figure 5.15: Threshold values for the determination of a diagnosis state.

After the case has been finished the final states of the diagnoses can be retrieved. All diagnoses $D \in \Omega_{\mathcal{D}}$ with state $val(D) = probable$ are established, and are assumed to be part of the solution of case c. It is worth mentioning, that the translation of the boundary categories given in Figure 5.14 is not proportional to the remaining categories. Thus, the negative category $N7$ is translated to $-INF$ (commonly in practice realized by using the most negative value representable by the system). This is motivated by the fact, that often domain experts want to exclude a diagnosis for a given observation, which by no means should be become suggestive or probable during the remaining course of the case. The positive boundary value $P7$ has similar motivation: With such a category, experts are able to establish a diagnosis for a given observation, which hardly can be de-established by other rules possibly firing during the remaining course of the case (with exclusion of the category $N7$).

Although rules are an intuitive representation of structural knowledge, it is difficult for rule-based approaches to discriminate among the established diagnoses: When retrieving more than one established diagnosis, rule-based formalisms do not provide a built-in method to distinguish between composite versus differential diagnoses. This problem can be compensated by introducing problem areas (i.e., coarse diagnoses) defining sets of alternative diagnoses. Then, in a first step problem areas are established, and refined in a second step to more explicit diagnoses contained in the established problem areas (establish-refine strategy). Established problem areas state a composite solution for the given case providing fine-grained diagnoses as alternatives.

As claimed at the beginning of this section, scoring rules are a generalization of (categorical) diagnosis rules. It is easy to see, that for a fixed value range, e.g., $\Omega_{scr} = \{0, P7\}$, any scoring rule base simplifies to a categorical rule base.

5.4.3. Acquisition of Score-Based Knowledge

The acquisition of score-based knowledge is done by scoring rules. However, experience has shown that scoring rules with arbitrary complexity are very difficult to understand and to maintain. For this reason, we propose the use of simple scoring rules (rule with a simple rule condition), which is described by the DIAGNOSTIC SCORE pattern.

The DIAGNOSTIC SCORE Pattern

Experiences with developing large (score-based) knowledge systems have motivated the introduction of *knowledge formalization patterns* [102, 106]. Such patterns should not only support the implementation of score-based systems, but also provide a guideline for the developers of how to formalize their knowledge. Knowledge formalization patterns are described by the paragraphs *Name, Synopsis, Motivation, Applicability, Solution,* and *Consequences*.

The basic idea of the DIAGNOSTIC SCORE pattern is to rate all possible individual findings with respect to given diagnoses. Then, for each diagnosis we list all findings confirming or disconfirming the diagnosis and rate their confirmation strengths using symbolic categories. Typically, for each finding-diagnosis relation only a small confirmation category is defined. For using the DIAGNOSTIC SCORE pattern a specialized editor, the DIAGNOSTIC SCORE table, can be applied. An example is depicted in Figure 5.16.

For each cell in the table a simple scoring rule is generated with the finding specified in the

DIAGNOSTIC SCORE Table	Diagnosis D_1	Diagnosis D_2	Diagnosis D_3
choiceEqual(Q_1:v_1)	P1	P1	P2
choiceEqual(Q_2:v_2)		N2	
choiceEqual(Q_3:v_3)	P1		P2
choiceEqual(Q_4:v_4)	P3	P1	N1
$\vdots$	$\vdots$	$\vdots$	$\vdots$

Figure 5.16: An exemplary diagnostic score table.

table line as rule condition and the diagnosis in the table column as rule action. The confirmation category defined in the table cell denotes the confirmation category $s \in \Omega_{scr}$. E.g., for the finding Q_1:v_1 and the diagnosis D_1 the scoring rule $r = \text{choiceEqual}(Q_1, v_1) \xrightarrow{P1} D_1$ is generated.

Since DIAGNOSTIC SCORE applies only small confirmation categories, score-based knowledge using this pattern is usually very robust with respect to noisy or incomplete data input. However, sometimes simple scoring rules are not sufficient for describing the structural knowledge. Then, the use of abstractions, rule exceptions, and rule contexts can help to retain the simplicity of the knowledge representation, while increasing the power of expression. Abstractions, rule contexts, and rule exceptions were described in Section 5.1.

For example, the LIMPACT [83, 80] system was developed using score-based knowledge. To reduce the complexity of the rule base, the developer applied the rule context *established("suitable stream")*. Thus, structural knowledge determining the pollution level with respect to stream observations is only evaluated in context of a (previously derived) diagnosis "suitable stream".

Since the acquisition of scoring rules can be complex and costly, especially the estimation of appropriate confirmation categories, we provide a method for learning simple scoring rules, that easily can be modified manually, afterwards. See Section 5.4.6 (p. 102) for more details.

Using Complex Scoring Rules

If the combined observation of findings mean a disproportionate confirmation or disconfirmation of a diagnosis when compared to the single observation of the findings, then the plain application of the DIAGNOSTIC SCORE pattern may be not appropriate. In this case, the definition of sub-scores or the application of complex rules are suggestive to solve this conflict. However, the use of complex rules may break up the knowledge formalization pattern, and therefore the use of complex rules should be limited and well reasoned.

5.4.4. Testing Rule-Based Knowledge

In the following we present methods, which can investigate rule-based knowledge for anomalies, its robustness, and its understandability.

Static Verification for Rule Base Integrity

An anomaly describes an integrity violation within the implemented rule base, and can be the reason for erroneous behavior of the knowledge system.

Anomalies for rule-based knowledge systems have been thoroughly investigated in the literature, e.g., see [11, 33, 94].

According to Preece [94] rule-based anomalies can be classified into the following sub-categories:

- **Redundancy**
  The rule base contains redundant knowledge, e.g., unfirable rules, subsumed rules, unusable rule actions.
- **Ambivalence**
  The rule base contains contradictory rules with respect to the semantics of the rule actions, or the syntax of the rules.
- **Circularity**
  The rule base includes rules, that generate a circular rule execution.
- **Deficiency**
  The knowledge base contains findings, that are not used by any rule. Furthermore, deficiency is caused by missing rules, and for a given problem no solution can be inferred.

Methods for finding the several kinds of anomaly are commonly subsumed by the *static verification* [93] method. Preece [96] describes methods for finding anomalies in first-order logic rule bases. In [3] these methods were adapted for the application with the presented knowledge representation, i.e., the ontology introduced in Chapter 4. In the following, we concentrate on the application of static verification in context of the agile process, i.e., in which situations warnings and errors are reported.

Mechanics In the context of the agile development process we investigate the rule base for the following anomalies, and report errors or warnings as given by the tables.

► **Redundancy**	
Unusable Rule Condition A rule cannot fire since its condition can never evaluate true.	error
Unusable Rule Action A rule has no effect to a knowledge system run, since the fired rule action is not used by any other knowledge (e.g., unused abstraction).	error
Subsumed Rule A rules r is subsumed by another rule r', if the rule condition $cond(r)$ is more specific than the rule condition $cond(r')$, and both rules have the same rule action.	warning

► **Circularity**	
A rules base can produce circles by means of conditions and actions of rules, e.g., a diagnosis is scored by a rule, which itself only can fire, if the diagnosis is in a given state. Circles can also appear across multiple rules.	error
► **Ambivalence**	
Syntactic Ambivalence Two rules r, r' are syntactically ambivalent, if $cond(r) = cond(r')$, and the comparison of the two rule actions shows syntactical ambivalence as defined as follows: For scoring rules: Both rules have the same targeted diagnosis in their rule action, but with different confirmation categories. For abstraction rules: Both rules infer different values for the same abstracted question.	warning
► **Deficiency**	
Unused Finding An unused finding does not appear in any rule of the rule base and therefore is unnecessary for inferring diagnoses. However, often findings are implemented for documentary reasons, and are important even though they are not used by the structural knowledge. Therefore, the results of this method are not reasonable in any case, and should be carefully applied.	warning
Unreachable Diagnosis For an unreachable diagnosis there exists no scoring rule with this diagnosis in its rule action, i.e., the diagnosis cannot be inferred in any case. Additionally, the test can be refined in order to find diagnoses that have no appropriate scoring rules for establishing or suggesting the diagnosis.	warning
Unreachable Abstract Finding An unreachable abstract finding is an abstract finding, for which no abstraction rule exists with this finding in its rule action, i.e., no value for this finding can be inferred in any case.	warning

For a comprehensive description of the algorithmic details explaining the detection of anomalies we refer to Akin [3].

Usage The static verification method can be applied for finding anomalies. The method requires no additional test knowledge. Sometimes, anomalies can be intentionally contained in the knowledge base. For example, a *subsumed rule* can be implemented on purpose to intensify the diagnosis score due to a more general rule. Another example is the *unused finding*, which may not be required for inferring a diagnosis, but can be meaningful for other reasons, e.g., in a medical application the patient name for documenting the case.

Torture Tests for Rule Bases

Torture tests are investigating the robustness of the implemented knowledge, and are introduced in Section 5.6.4 (p. 130) as a black-box testing method. However, they can be also applied as a white-box test for rule-based knowledge. We distinguish between the white-box testing methods *reduce-knowledge* (decrease quantity of knowledge) and *modify-knowledge* (decrease quality of knowledge). Originally, torture tests were described by Groot et al. [49] for rule-based knowledge.

Mechanics Since the automated evaluation of the testing methods we need to set appropriate default values, which are usually determined beforehand by a manual inspection of the degradation studies.

For the *reduce-knowledge* approach we systematically reduce the number of rules contained in the knowledge base and measure the degree of the decreased diagnostic performance. Rules are typically removed at random. For an automated evaluation of this test we need to define the following thresholds: $\mathcal{T}_R$, the maximum percentage of removed rules of the knowledge base, and $\mathcal{T}_F$ the minimum average of the computed accuracy for the degraded test cases. If the computed accuracy falls below the threshold $\mathcal{T}_F$, then an error is reported.

The *modify-knowledge* approach systematically modifies the actions of implemented rules. Thus, the current confirmation strength of a scoring rule is changed; usually the strength is in-/decreased to an adjacent strength. Alternatively, for an abstraction rule the value of an abstracted question is modified to an adjacent value. For an automated evaluation of this method, a threshold for the minimum accuracy $\mathcal{T}_F$, and a maximum percentage of modified rules $\mathcal{T}_M$ is required. During the study the rules are randomly modified. If the computed accuracy falls below the threshold $\mathcal{T}_F$, then an error is reported. The implementation and evaluation of the test is analogous to the torture test that are presented in Section 5.6.4 (p. 130).

Usage With torture tests the robustness of the implemented structural knowledge can be tested. The presented *reduce-knowledge* and the *modify-knowledge* approaches allow for a finer gradation of testing the robustness when compared to the black-box torture is introduced in Section 5.6.4. The method requires test cases as test knowledge. Reasonable thresholds for the minimum accuracy $\mathcal{T}_F$, the maximum percentage of removed rules $\mathcal{T}_R$ of the knowledge base, and a maximum percentage of modified rules $\mathcal{T}_M$ of the knowledge base are required.

Static Rule Base Testing

Similar to the *static ontology testing* method presented in Section 4.5.1 the *static rule base testing* method tries to give an overview of the implemented rule base.

Mechanics The method generates a statistic of the implemented rules and highlights significant irregularities. Thus, the total numbers and percentages of implemented simple, medium and complex rules are displayed. Furthermore, the statistic distinguishes abstraction rules, categorical rule, and scoring rules. Additionally, the number and percentage of rules for each diagnosis and for each abstracted finding are calculated. The following irregularities are presented as warning:

- Diagnoses with an exceptionally large or small percentage of implemented rules, deriving the diagnosis
- Abstracted questions with an exceptionally large or small percentage of implemented rules, deriving values for the question

The calculated numbers and percentages of the remaining diagnoses and abstracted findings are presented to the user as information on demand.

Usage The static rule base testing method is used to investigate the understandability of the implemented rule base. For example, diagnoses and abstracted findings with exceptionally large derivation knowledge can point to parts in the knowledge base, that may require restructuring in the future. A smaller size of derivation knowledge for single entities mostly increases the understandability of the rule base.

Dynamic Rule Base Testing

With dynamic rule base testing the developer can determine the parts of the implemented rule knowledge, which is actually used under real world conditions.

Mechanics The method applies a (sufficiently) large number of test cases and runs them using the implemented knowledge system. During the test case evaluation the frequency of the used rules is counted. Rules with no or very seldom usage (defined by a threshold) are reported as a warning.

Usage Experience in various projects has shown, that knowledge bases often contain rules, which are never or nearly never used, and therefore have no significant impact on the practicability and accuracy of the rule base. However, large rule bases tend to be complex and to be hardly maintainable. For this reason, the *dynamic rule-base testing* method is very useful to detect so called *lazy* rules, that are never or very rarely applied. Thus, the understandability of the rule base is tested.

The method requires a sufficiently large case base with real cases as test knowledge and a threshold of minimum usage to detect lazy rules.

It is worth noticing, that the usage of diagnoses and findings for real world cases is already evaluated by the *Case-Based Ontology Testing* method described in Section 4.5.2 (p. 54).

5.4.5. Restructuring Rule-Based Knowledge

In this section, we briefly consider the propagation of ontological restructuring methods for rule-based knowledge. Furthermore, we introduce explicit restructurings for the rule-based representation.

Propagation of Ontological Restructuring Methods

The ontological restructuring methods sketched in Section 4.6 (p. 55) are also propagated to the rule-based interpretation of the structural knowledge container. For the methods TRANSFORMMCINTOYN and TRANSFORMYNINTOMC the transformed objects are simply updated in the conditions and actions of the implemented rules. In general, no conflict will be produced for these two restructurings.

TRANSFORMNUMINTOOC The method TRANSFORMNUMINTOOC is only executable, if the rule base only contains rules that partition the numerical value range in distinct partitions. Otherwise, no direct transformation is possible and the restructuring should be aborted.

TRANSFORMMCINTOOC The TRANSFORMMCINTOOC method similarly can cause conflicts, if the rule base contains rules with either multiple values of the specified multiple-choice question in their rule conditions (any kind of rules), or multiple values of the specified multiple-choice question in their rule action (abstraction rules). Then, the restructuring should be aborted, and the conflict needs to be manually resolved.

SHRINKVALUERANGE The method SHRINKVALUERANGE simply exchanges the original values with the values of the reduced value range, if contained in a rule condition or rule action (for abstraction rules). The method causes a conflict, if syntactically ambivalent rules are generated due to the execution of the restructuring method. Syntactic ambivalence of two rule is tested by static verification, introduced in Section 5.4.4. Then, the method is aborted and the rule base needs to be manually modified.

MOVEQUESTIONVALUE If the moved value is contained in a rule condition, then the condition is modified, so that the new targeted question is contained in the condition. However, the restructuring can cause conflicts, if e.g., the new targeted question additionally is contained in the rule condition, and is assigned to a value different to the moved value. For one-choice questions this condition can never evaluate true, since two values can never be assigned to an one-choice question at the same time. Then, the developer has to manually decide, which partial condition has to be removed from the condition. If the moved value is contained in the action of an abstraction rule, then the action has to be modified, so that the new targeted question is now assigned to the moved value.

INTRODUCEABSTRACTION For the method INTRODUCEABSTRACTION a corresponding abstraction rule needs to be inserted, which is often done automatically because of the specified abstraction condition. Then, all rules currently contained in the rule base are checked with respect to the new abstraction: If a rule condition contains the specified condition, then it is replaced by the newly created abstraction.

REMOVEQUESTION / REMOVEDIAGNOSIS The methods REMOVEQUESTION and REMOVEDIAGNOSIS can be executed automatically, if no knowledge is attached to the specified objects. Otherwise, defaults need to be used. For the rule-based representation we check, if the question or diagnosis is contained in a condition or in an action of an implemented rule. This simply can be tested by the deficiency tests introduced in Section 5.4.4. If rule-based knowledge is available for the specified object, then the rules are modified according to the given defaults: In general, three defaults are offered: 1) Delete all rules containing the considered object. 2) Only the corresponding sub-condition is removed, and the rule is only deleted, if the modified rule condition is syntactically deficient, e.g., the condition is empty. 3) The user is interactively asked, how to proceed with the modified rules.

Rule-Based Restructurings

Rule-based restructuring methods consider the modification of the existing rule base. Atomic changes of single rules can be hardly seen as a restructuring method, since only slight changes of the rule condition or rule action are made. However, due to nature of the rule-based representation side effects are possible, and therefore changes should be always validated by appropriate tests methods.

The manual restructuring of the *entire* rule base is a more interesting topic, which has not been investigated so far. An entire restructuring can become necessary, if the rule base has been constructed in a ”chaotic” manner. Missing rule base design often implies a missing understandability and maintainability of the implemented knowledge. For this reason, knowledge formalization patterns [102] have been introduced, in order to increase the understandability of rule bases. In general, they implement structured approaches and guidelines for representing the implemented rules. In the following, we sketch two restructuring methods for introducing the DIAGNOSTIC SCORE pattern and the HEURISTIC DECISION TABLE pattern, respectively.

INTRODUCEDIAGNOSTICSCORE The DIAGNOSTIC SCORE pattern states that the diagnostic knowledge is only defined using simple scoring rules. Any finding contributes with a negative or positive confirmation category to a given diagnosis. These categories are aggregated to a final score, which determines the state of the diagnosis. DIAGNOSTIC SCORE is very easy to implement and robust with respect to noisy input.

Consequently, only simple diagnostic rules are allowed for implementing the pattern. Therefore INTRODUCEDIAGNOSTICSCORE method firstly investigates the available rule base according to this characteristic. If only simple diagnostic rules are contained in the rule base, then the pattern is already implemented, and nothing has to be done. If the rule base contains one-level and multiple-level rules, then these rules need to be split up into rules with simple complexity. This task is very difficult, since the semantics of the original rules are changed in many ways.

We will sketch the transformation of one-level rules in the following. Multiple-level rules are transformed similarly to the scheme presented for one-level rules. If the terminal conditions of an one-level rule are connected by an AND condition, then for each terminal

condition a simple scoring rule is generated. The specified diagnosis of the original rule is adopted for the new rules. The original confirmation category is adjusted (decreased) for the new rules, so that the aggregation of the new categories will result in the original confirmation category. If the terminal conditions of the one-level rule are connected by an OR condition, then analogously for each terminal condition a simple scoring rule is generated. The action of new scoring rules contain the equal diagnosis and the confirmation category as specified in the original scoring rule. However, this may cause irregularities, since with the simple scoring rules a diagnosis can obtain a higher score than by using the one-level scoring rules.
For this reason, it is suggestive to perform the restructuring method interactively with the user for one complex rule per time only.

INTRODUCEHEURISTICDECISIONTABLE Using the HEURISTIC DECISION TABLE pattern the diagnostic knowledge is mainly implemented by one-level rules. Thus, one-level scoring rules are formulated that either establish or exclude diagnoses. The pattern is applied, if combinations of observed findings, when compared to single observations, have an intensified meaning for diagnoses. As a special characteristic the rules represented in the HEURISTIC DECISION TABLE are self-contained, i.e., they are evaluated separately and the specified confirmation categories are not aggregated to a single score. We can specify scoring rules for either excluding, suggesting or establishing a given diagnosis. If more than one scoring rule for a given diagnosis evaluates true in a given case, then the diagnosis' state is determined according to the following priority schema: The exclusion of the diagnosis has the highest priority compared to establishing the diagnosis, which is more important than suggesting the diagnosis.
The INTRODUCEHEURISTICDECISIONTABLE restructuring method consequently modifies the available rule base with the purpose to create only one-level scoring rules, establishing, suggesting or excluding diagnoses. Then, for each diagnosis all scoring rules are collected and divided into two partitions: The *positive partition* contains all rules with positive confirmation categories, whereas the *negative partition* consists of scoring rules with negative confirmation categories. For the positive partition, all combinations of the original rule conditions are joined, so that the aggregation of their corresponding confirmation category either suggests or establishes the given diagnosis. Excluding rules are generated using the negative partition. Rules are created analogously to the method sketched for the positive partition.

5.4.6. Learning Score-Based Knowledge

Although score-based knowledge is a convenient method for acquiring diagnostic knowledge, (semi-)automatic learning methods can help the developer to build knowledge systems rapidly. This section will present a method for inductively learning scoring rules according to the DIAGNOSTIC SCORE pattern. The simple scoring rules are generated with respect to the following criteria: Firstly, the learned rules should achieve a high accuracy for inferring the given diagnoses. Secondly, the complexity of the learned knowledge should be as simple as possible, i.e., the number of learned features and applied scores are as low as possible. Initially, this approach was presented by Atzmueller et al. [9], and was evaluated on a real world application.

For learning scoring rules the algorithm basically applies three steps: Starting with a statistical analysis of the case base the dependency associations between diagnoses and findings are determined, i.e., so-called frequency profiles are generated. Then, each dependency is rated with respect to its significance for the diagnostic process. In the third step, a quasi-probabilistic rating is computed for each significant diagnosis-finding association, and a scoring rule for each association is generated. The symbolic category of the rule is determined by a mapping function using the quasi-probabilistic rating. In order to simplify the computation and the results of the algorithm, only discrete choice questions are considered for the learning task. However, numerical questions can be handled easily by discretization in a preprocessing step. In the following ,we present the three basic steps of the algorithm in more detail:

Step 1: Generating Frequency Profiles

In the first step, the given case base CB is analyzed according to the frequencies of diagnoses and findings, and their associations, respectively. The results of this analysis are stored in frequency profiles. We define a frequency profile for a diagnosis as follows.

Definition 5.4.2 (Frequency Profile) A frequency profile $FP_{CB}(D)$ for a diagnosis $D \in \Omega_{\mathcal{D}}$ contained in a case base CB is defined as the set of tuples

$$FP_{CB}(D) = \left\{ \big(F, \mathit{freq}_{CB}(F,D)\big) \;\big|\; F \in \Omega_{\mathcal{F}} \wedge \mathit{freq}_{CB}(F,D) \in [0,1] \right\},$$

where F is a finding and $\mathit{freq}_{CB}(F,D) \in [0,1]$ represents the frequency the finding F occurs in conjunction with D in the case base CB, i.e.,

$$\mathit{freq}_{CB}(F,D) = \frac{\big|\{\, c \in CB \mid F \in \mathcal{F}c \wedge D \in \mathcal{D}c \}\big|}{\big|\{\, c \in CB \mid D \in \mathcal{D}c\}\,\big|}\,.$$

Thus, a frequency profile of a diagnosis contains all findings with attached frequencies, that co-occur with the diagnosis in the case base.

It is obvious, that a frequency profile initially can contain many findings, and therefore we apply statistical pruning in order to reduce the profiles to the most significant findings for the particular diagnoses.

Step 2: Pruning Frequency Profiles

We prune each frequency profile by creating a four-fold contingency table for each finding and diagnosis considered by the particular profile. The contingency table describes the frequencies of the finding-diagnosis association in more detail (for non-boolean findings contingencies can be defined analogously).

	D	$\neg D$
F	a	b
$\neg F$	c	d

Figure 5.17: Four-fold contingency table for diagnosis-finding associations.

The letters in the contingency table describe the following frequencies relative to the considered case base CB.

$$\begin{aligned} a &= N(D \wedge F)\,, & b &= N(\neg D \wedge F), \\ c &= N(D \wedge \neg F)\,, & d &= N(\neg D \wedge \neg F), \end{aligned}$$

where $N(cond)$ is the number of times the condition $cond$ is true for cases $c \in CB$, i.e., the diagnosis or finding is contained in the case base or not.
With the contingency table we can apply the χ^2-test for independence for determining the dependencies between the finding-diagnosis relations. For binary events the formula of the χ^2-test is defined as follows:

$$\chi^2(F, D) = \frac{(a+b+c+d)(ad-bc)^2}{(a+b)(c+d)(a+c)(b+d)}\,.$$

Especially, if the case base only allow for small sample sizes, then we can apply the Yates' correction for a more reasonable comparison. For all dependent tuples (F, D) we derive the quality of the dependency using the ϕ-coefficient

$$\phi(F, D) = \frac{ad - bc}{\sqrt{(a+b)(c+d)(a+c)(b+d)}}\,,$$

which measures the degree of association between two binary variables. The ϕ-coefficient can be used to detect positive and negative dependencies between the variables, respectively. However, we will only consider dependencies, for which the absolute value of the ϕ-coefficient exceeds a given threshold $\mathcal{T}_\phi$. Therefore, all dependencies between a diagnosis and a finding with an absolute ϕ-coefficient less than this threshold will be removed from the frequency profile. Consequently, the *reduced frequency profile* is defined as follows.

Definition 5.4.3 (Reduced Frequency Profile) The reduced frequency profile $FP^*_{CB}(D)$ for a diagnosis $D \in \Omega_{\mathcal{D}}$ contained in a case base CB is defined as a set of tuples

$$\begin{aligned} FP^*_{CB}(D) = \big\{ \big(F, \mathit{freq}_{CB}(F, D)\big) \;\big|\; & F \in \Omega_{\mathcal{F}} \\ & \wedge\ \mathit{freq}_{CB}(F, D) \in [0, 1] \ \wedge\ |\phi(F, D)| > \mathcal{T}_\phi \big\}\,, \end{aligned}$$

As a result of this second step we arrive at a set of reduced frequency profiles, containing only significant dependencies between diagnoses and findings.

Step 3: Rule Generation

In the last step, each association between a diagnosis $D \in \Omega_{\mathcal{D}}$ and a finding $F \in \Omega_{\mathcal{F}}$ is used for the generation of a scoring rule $F \xrightarrow{s} D$. An association between a diagnosis D and a finding F is given, if F is contained in the reduced frequency profile $FP^*_{CB}(D)$.
The applied confirmation category $s \in \Omega_{scr}$ is calculated by the *precision* and the *false alarm rate* of the considered scoring rule.
Then, the *precision* of a rule is defined as

$$precision(r) = \frac{TP}{TP + FP},$$

and the *false alarm rate* (*far*) is given by

$$far(r) = \frac{FP}{FP + TN},$$

where TP, TN, FP are the number of *true positives, true negatives*, and *false positives,* respectively. It is worth noticing, that the specificity is defined by $specificy = 1 - far$.
These can easily be extracted from the contingency table defined in Figure 5.17. For a positive dependency ($\phi(F, D) > 0$) between finding F and D, $TP = a$, $TN = d$ and $FP = b$. For a negative dependency ($\phi(F, D) < 0$) we have to predict the absence of the diagnosis, and therefore we need to define $TP = b$, $TN = c$ and $FP = a$.
We rate the dependency by computing a *quasi probabilistic score (qps)*, which is defined as follows:

$$qps(r) = sgn\big(\phi(D, F)\big) * precision(r)\big(1 - far(r)\big) \tag{5.15}$$

Finally, this quasi-probabilistic score is mapped to a symbolic confirmation category. The continuous values of $qps(r)$ are mapped with respect to the function given in Figure 5.18.

qps(r)		**category(r)**	**qps(r)**		**category(r)**
[-1.0, -0.9)	$\rightarrow$	$N6$	(0.0, 0.5)	$\rightarrow$	$P1$
[-0.9, -0.5)	$\rightarrow$	$N3$	[0.5, 0.9)	$\rightarrow$	$P3$
[-0.5, 0.0)	$\rightarrow$	$N1$	[0.9, 1.0]	$\rightarrow$	$P6$

Figure 5.18: Mapping function for quasi-probabilistic scores to symbolic confirmation categories.

It is easy to see that the mapping function given in Figure 5.18 can be refined in order to cover a larger or smaller value range of confirmation categories. However, the presented value range is appropriate for a user-friendly interpretation and understanding of the learned scoring rules by the user. The understandability of the learned rule base can be further improved by reducing its size. We achieve this by merging rules for continuous attributes. Thus, if two rules are conditioning neighboring partitions of the same question, and have the equal confirmation category, then we combine the two rules to one rule with an extended condition. This simple method can reduce the number of learned rules without loosing accuracy or changing the semantics of the learned rule base.

In [9] the presented algorithm was refined in order to process abnormality and partition class information, which was shown to further reduce the complexity of the learning results. Further, the learning algorithm was evaluated with a real life case base.

5.4.7. Summary

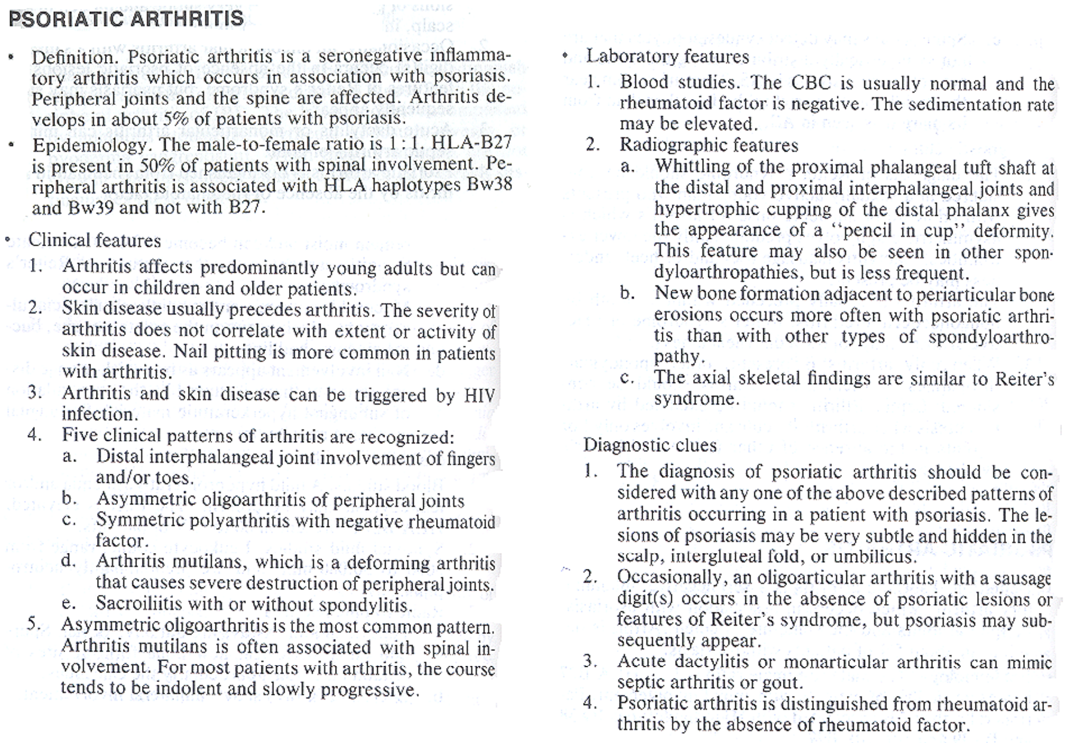
In this section, we have introduced score-based knowledge represented by scoring rules. Scoring rules attach symbolic confirmation categories to diagnoses, which are aggregated in order to determine the state of the given diagnoses. We have described the knowledge representation and inference in detail, and described the acquisition of score-based knowledge using the knowledge formalization pattern DIAGNOSTIC SCORE. Finally, we presented an approach for learning scoring rules, which tries to learn compact and understandable rule bases for a given case base.

5.5. Causal Set-Covering Knowledge

In this section, we introduce causal set-covering models for representing structural knowledge. Set-covering models can be incrementally constructed, first starting with symbolic confirmation strengths for stating uncertainty and can be later refined by an even more precise formalism for uncertainty, i.e., quantitative uncertainty. Furthermore, we describe testing methods, restructuring methods, and a learning approach for set-covering models.

5.5.1. Symbolic Confirmation Strengths

Set-covering models are an intuitive representation for diagnostic reasoning in many domains, e.g., in medicine and biology. In the literature knowledge is often formulated based on a list of typical effects of a diagnosis. Medical textbooks commonly describe diseases by their observable findings. Analogously, in biology plants are characterized by their observable properties. Figure 5.19 depicts the exemplary description of the rheumatic disease ”psoriatic arthritis” taken from a medical textbook [38].



PSORIATIC ARTHRITIS

- Definition. Psoriatic arthritis is a seronegative inflammatory arthritis which occurs in association with psoriasis. Peripheral joints and the spine are affected. Arthritis develops in about 5% of patients with psoriasis.
- Epidemiology. The male-to-female ratio is 1 : 1. HLA-B27 is present in 50% of patients with spinal involvement. Peripheral arthritis is associated with HLA haplotypes Bw38 and Bw39 and not with B27.
- Clinical features
  1. Arthritis affects predominantly young adults but can occur in children and older patients.
  2. Skin disease usually precedes arthritis. The severity of arthritis does not correlate with extent or activity of skin disease. Nail pitting is more common in patients with arthritis.
  3. Arthritis and skin disease can be triggered by HIV infection.
  4. Five clinical patterns of arthritis are recognized:
     a. Distal interphalangeal joint involvement of fingers and/or toes.
     b. Asymmetric oligoarthritis of peripheral joints
     c. Symmetric polyarthritis with negative rheumatoid factor.
     d. Arthritis mutilans, which is a deforming arthritis that causes severe destruction of peripheral joints.
     e. Sacroiliitis with or without spondylitis.
  5. Asymmetric oligoarthritis is the most common pattern. Arthritis mutilans is often associated with spinal involvement. For most patients with arthritis, the course tends to be indolent and slowly progressive.
- Laboratory features
  1. Blood studies. The CBC is usually normal and the rheumatoid factor is negative. The sedimentation rate may be elevated.
  2. Radiographic features
     a. Whittling of the proximal phalangeal tuft shaft at the distal and proximal interphalangeal joints and hypertrophic cupping of the distal phalanx gives the appearance of a “pencil in cup” deformity. This feature may also be seen in other spondyloarthropathies, but is less frequent.
     b. New bone formation adjacent to periarticular bone erosions occurs more often with psoriatic arthritis than with other types of spondyloarthropathy.
     c. The axial skeletal findings are similar to Reiter’s syndrome.

Diagnostic clues
1. The diagnosis of psoriatic arthritis should be considered with any one of the above described patterns of arthritis occurring in a patient with psoriasis. The lesions of psoriasis may be very subtle and hidden in the scalp, intergluteal fold, or umbilicus.
2. Occasionally, an oligoarticular arthritis with a sausage digit(s) occurs in the absence of psoriatic lesions or features of Reiter’s syndrome, but psoriasis may subsequently appear.
3. Acute dactylitis or monarticular arthritis can mimic septic arthritis or gout.
4. Psoriatic arthritis is distinguished from rheumatoid arthritis by the absence of rheumatoid factor.

Figure 5.19: Textbook definition [38] of the rheumatic disease ”psoriatic arthritis”.

Set-covering knowledge states causal knowledge between diagnoses and findings. A set-covering model consists of set-covering relations of the following form:

> If a diagnosis D is true, then the questions $Q_1, \ldots, Q_n$ are observed with corresponding values $v_1, \ldots, v_n$.

A single set-covering relation r is denoted by $r = D \rightarrow Q{:}v$; we say that the finding $Q{:}v$ is *covered* by the diagnosis D.
For example, Figure 5.20 shows a set-covering model $\mathcal{R}$ with 5 set-covering relations for two diagnoses D_1 and D_2. An edge from a diagnosis D to a finding $Q{:}v$ with the label r indicates a set-covering relation $r = D \rightarrow Q{:}v$.

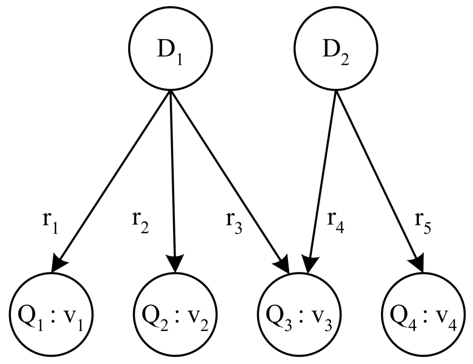


Figure 5.20: Set-covering model for the diagnoses D_1 and D_2 and the questions Q_1, Q_2, Q_3 and Q_4.

Textbook knowledge as depicted in Figure 5.19 can be easily translated into set-covering relations. For a more powerful representation a covering strength can be attached to each set-covering relation, which state a symbolic frequency of the observation of the specified finding. Definition 5.5.1 gives a formal description of a set-covering relation.

Definition 5.5.1 (Set-Covering Relation, Set-Covering Model) A *set-covering relation* r between a diagnosis $D \in \Omega_{\mathcal{D}}$ and a finding $F \in \Omega_{\mathcal{F}}$ is denoted by $r = D \rightarrow F$. We say that "D covers F". $\Omega_{\mathcal{R}}$ denotes the universe of all set-covering relations. A *set-covering model* is a set $\mathcal{R} \subseteq \Omega_{\mathcal{R}}$ of set-covering relations.
To each set-covering relation $r \in \Omega_{\mathcal{R}}$ a symbolic covering strength $cs(r)$ can be attached. Let $\Omega_{CS} = \{P3, P2, P1, 0, N1, N2, N3\}$ be the universe of all possible covering strengths, which states the frequencies of observation of specified finding in descending order, e.g., $P3$ means "very often", 0 means "unclear", and $N3$ means "very seldom".

We can see that set-covering models using symbolic categories allow for a simple and intuitive translation of diagnostic textbook knowledge.

5.5.2. Symbolic Diagnostic Inference

In general, set-covering diagnosis can be described as the task of finding an appropriate hypothesis that can explain a given observation $\mathcal{F}_{\mathcal{O}}$. A hypothesis is defined as a set of diagnoses that together constitute a covering of $\mathcal{F}_{\mathcal{O}}$.

Definition 5.5.2 (Hypothesis) We call a set $\mathcal{H} \subseteq \Omega_{\mathcal{D}}$ of diagnoses a *hypothesis*. A hypothesis $\mathcal{H} = \{ D_1, \ldots, D_n \}$ can be interpreted as a conjunction $D_1 \wedge \cdots \wedge D_n$ of diagnoses, which tries to explain a given observation.

Given a set $\mathcal{F}_\mathcal{O} \subset \Omega_\mathcal{F}$ of *observed findings*. The goal is to find a hypothesis $\mathcal{H}$ which is able to explain the observed findings $\mathcal{F}_\mathcal{O}$. This is the case if $\mathcal{H}$ covers all observed findings, where a finding F is *covered* by $\mathcal{H}$, iff F is covered by at least one diagnosis $D \in \mathcal{H}$. If a finding is not covered by $\mathcal{H}$, then it is called *isolated* w.r.t. $\mathcal{H}$; the set of all observed findings that are not covered by (isolated w.r.t.) $\mathcal{H}$ will be denoted by $\mathcal{F}_\mathcal{O}^{iso}$.
A set of findings $\mathcal{F}$ is called *functional*, iff for all questions $Q \in \Omega_Q$ there exists at most one finding $Q{:}v \in \mathcal{F}$, i.e., with Q in the first component. In that case we can define the partial function $val_\mathcal{F} : \Omega_Q \to \Omega_{val}$ which returns the value v assigned to a given question $Q \in \Omega_Q$: $val_\mathcal{F}(Q) = v$, if $Q{:}v \in \mathcal{F}$, and $val_\mathcal{F}(Q) = \perp$, otherwise. Furthermore, we define $a(\mathcal{F}) = \{\, Q \in \Omega_Q \mid \exists\, Q{:}v \in \mathcal{F} \,\}$ to be the set of questions in a set $\mathcal{F}$ of findings.

Definition 5.5.3 (Predicted Findings) Given a set-covering model $\mathcal{R}$, a diagnosis $D \in \Omega_\mathcal{D}$, and a hypothesis $\mathcal{H} \subseteq \Omega_\mathcal{D}$. Then

$$\mathcal{F}_D = \{\, F \in \Omega_\mathcal{F} \mid D \to F \in \mathcal{R} \,\}$$

is the set of all findings that are covered by D. The set $\mathcal{F}_D$ is commonly called the set of *predicted findings* for a diagnosis D.

We select a functional set $\mathcal{F}_\mathcal{H} \subseteq \bigcup_{D\in\mathcal{H}} \mathcal{F}_D$ of findings that are covered by $\mathcal{H}$, i.e., $\mathcal{F}_\mathcal{H}$ should contain only one finding for each question Q. If two diagnoses in $\mathcal{H}$ cover the same question Q with different values, then $\bigcup_{D\in\mathcal{H}} \mathcal{F}_D$ is not functional. In this case, we select one of the conflicting findings $Q{:}v \in \bigcup_{D\in\mathcal{H}} \mathcal{F}_D$ as follows:

1 If one of the conflicting findings is contained in $\mathcal{F}_\mathcal{O}$, then we select it.
2 If probabilities are defined, then we select the finding with the most probable set-covering relation in $\mathcal{F}_\mathcal{H}$.
3 Otherwise, we randomly select one finding.

We remark, that additional knowledge can enhance the conflict resolution strategy. E.g., in [14] abnormality information was used to select the most abnormal parameter value. If abnormality information for parameter values is defined in the model, then we can apply this easily for conflict resolution.
Given a set $\mathcal{F}_\mathcal{O}$ of observed findings and a set $\mathcal{F}_\mathcal{H}$ of predicted findings. We say that a question $Q \in \Omega_Q$ is observed, if there exists a finding $Q{:}v \in \mathcal{F}_\mathcal{O}$ for Q. For a comparison between $\mathcal{F}_\mathcal{H}$ and $\mathcal{F}_\mathcal{O}$ we introduce two subsets of $\mathcal{F}_\mathcal{H}$:

1 The set $\mathcal{F}_{\mathcal{H},\mathcal{O}}$ of *parametrically predicted findings* consists of all predicted findings $Q{:}v \in \mathcal{F}_\mathcal{H}$ for which the question Q is observed.
2 The set $\mathcal{F}^+_{\mathcal{H},\mathcal{O}} = \mathcal{F}_\mathcal{H} \cap \mathcal{F}_\mathcal{O}$ of *positively predicted findings* consists of all predicted findings which are observed with the predicted value.
Analogously, we define $\mathcal{F}^-_{\mathcal{H},\mathcal{O}} = \mathcal{F}_{\mathcal{H},\mathcal{O}} \setminus \mathcal{F}^+_{\mathcal{H},\mathcal{O}}$ to be the set of *negatively predicted findings*.

Weighted Questions in Set-Covering Models In many cases not all findings describe the same importance for the derivation of a solution. For this reason we introduce weights for findings, which are represented by global weight functions as defined in Definition 5.2.2 (p. 73). Then, a global weight function

$$w_g : \Omega_Q \to I\!N_+$$

defines the absolute importance of each question contained in the domain. For a convenient usage we also define the global weight function w_g for findings. For a finding $F = Q{:}v$ we say that

$$w_g(Q{:}v) = w_g(Q)\,,$$

i.e., the global weight of a finding is equal to the global weight of the embodied question.

Whereas, the evaluation of a given hypothesis is simple, the generation of appropriate hypotheses is a difficult task. It is easy to see that an exhaustive examination of all possible hypotheses has exponential complexity. Therefore, reasonable heuristics need to be applied.

Hypothesis Generation: Full Elimination

In the following, we describe a simple greedy-like algorithm for generating hypotheses based on a set of observed findings.

Algorithm 1 HYPOTHESIS GENERATION: FULL ELIMINATION

```
1: generateHyp( Hypothesis $\mathcal{H}$, Set $unexplained$ )
2: if terminate? then
3:    return
4: else
5:    $F$ = selectMaxFinding($unexplained$)
6:    $\{D_1, \ldots, D_k\}$ = bestKDiagnosesFor($F$)
7:    for all $D_i \in \{D_1, \ldots, D_k\}$ do
8:       $\mathcal{H}' = \mathcal{H} \cup \{D_i\}$
9:       addToGlobalHypList($\mathcal{H}'$)
10:      generateHyp($\mathcal{H}'$, $unexplained \setminus pred(D_i)$)
11:   end for
12: end if
```

The recursive algorithm is called with an empty hypothesis $\mathcal{H} = \{\}$, $unexplained = \mathcal{F}_\mathcal{O}$, and terminates (line 3), if at least one of the following conditions is true:

- All diagnoses have been considered for hypothesis generation, i.e., $\mathcal{H} = \Omega_\mathcal{D}$.
- The set $unexplained$ is empty, i.e., all observations have been explained by at least one diagnosis contained in the hypothesis $\mathcal{H}$.
- The apriori probability of the current hypothesis falls below a given threshold, i.e., the hypothesis must fulfill a specified likelihood.
- The current size of the hypothesis $\mathcal{H}$ exceeds a given threshold, i.e., the size of the hypothesis is restricted beforehand.
- The size of the already generated hypotheses (contained in the globalHypList) exceeds a given threshold, i.e., the number of hypotheses to be generated can be restricted beforehand.

The methods in lines 5 and 6 are responsible for the control of the greedy algorithm. In line 5 we select the most appropriate finding from the observation set, which has not been considered before. Mostly, the finding F with the largest global weight $w_g(F)$ is selected, but other strategies are possible, e.g., selecting the finding, which is covered by the most probable diagnosis. In line 6, we retrieve the k best diagnoses, for the selected finding F. Typically, we choose the diagnoses covering the finding F, for which the set-covering relations have the highest covering strength.

Data Structures For an efficient implementation of the algorithm each finding F needs to embody a list of set-covering relations $r = D \rightarrow F$ sorted according to the covering strength (used by method bestKDiagnosesFor(), line 6). Each diagnosis D needs to store a list of its covered findings in order to implement an efficient implementation of the set operation in the recursive call (line 10). Furthermore, the set of observed findings $unexplained$ needs to be sorted according to the weights of the included findings (a linked list will speed up the selection in line 5 and the remove operation in line 10). The global hypothesis list may be implemented as a linked list for an efficient and sorted insertion of new diagnoses.

Hypothesis Generation: Partial Elimination

The full elimination algorithm may not be appropriate for a feasible evaluation of the observation set. For this reason, we present a partial elimination algorithm, which takes into account the covering strength of each applied set-covering relation. Then, findings are not fully eliminated, but for each finding in the set $unexplained$ an explanation account is created, which represents a certain degree to which the finding is explained by the currently considered hypothesis.

Algorithm 2 HYPOTHESIS GENERATION: PARTIAL ELIMINATION

```
 1: generateHyp( Hypothesis $\mathcal{H}$, Set $unexplained$ )
 2: if terminate? then
 3:     return
 4: else
 5:     $F$ = selectMaxFinding($unexplained$)
 6:     $\{D_1, \ldots, D_k\}$ = bestKDiagnosesFor($F$)
 7:     for all $D_i \in \{D_1, \ldots, D_k\}$ do
 8:         $\mathcal{H}' = \mathcal{H} \cup \{D_i\}$
 9:         addToGlobalHypList($\mathcal{H}'$)
10:         update($D_i$, $unexplained$)
11:         generateHyp($\mathcal{H}'$, $unexplained$)
12:     end for
13: end if
```

We can see that the algorithm differs in the previously presented algorithm by one method: The method update(D_i, obs) considers the findings contained in obs and attaches to each finding $F \in unexplained$ the covering strength of the set-covering relation $D_i \rightarrow F$, if it is greater than the currently stored covering strength. Initially, all findings are assigned to the default covering strength 0. Then, each finding that has a greater covering strength than a given threshold value is removed from the set $unexplained$. With the threshold value we can predefine the desired degree of explanation strength. This feature highlights the main difference to the previously presented algorithm using full elimination, which virtually assumes a threshold value $\epsilon = 0$.

Data Structures In addition to the previously discussed data structures, the observation set $unexplained$ needs to implement a hash set, which stores the current covering strengths for the contained findings (with finding as the key and the strength as the value).

Alternative Generation Approaches

The presented approaches for generating set-covering hypotheses are quite simple and obviously can be improved by more sophisticated search methods, e.g., beam search. Furthermore, the consideration of background knowledge, e.g., negative covering strengths, can further shrink the hypothesis space. Advanced methods for reducing the hypothesis space were discussed, e.g., by Stefik [128, Ch. 9].

Hypothesis Evaluation

During the hypothesis generation step, new hypotheses are added to a global hypothesis list. This list is sorted with respect to the explanation quality of the included hypotheses according to the given observation. The explanation *quality* of a hypothesis $\mathcal{H}$ for a given observation $\mathcal{F}_{\mathcal{O}}$ by the following equation:

$$\varrho(\mathcal{F}_{\mathcal{H}}, \mathcal{F}_{\mathcal{O}}) = \frac{\sum\limits_{F \in \mathcal{F}^{+}_{\mathcal{H},\mathcal{O}}} max\_cs(F, \mathcal{H}) \cdot w_g(F)}{\sum\limits_{F \in \mathcal{F}_{\mathcal{O}}} w_g(F)}\,, \tag{5.16}$$

where $\mathcal{F}_{\mathcal{H}}$ are the covered findings of $\mathcal{H}$ and

$$max\_cs(F, \mathcal{H}) = max\big(\, cs(r) \,\big|\, r = D \rightarrow F \text{ with } D \in \mathcal{H} \,\big)$$

is the maximum covering strength of finding F w.r.t. a diagnosis $D \in \mathcal{H}$ covering F. Besides the quality a hypothesis $\mathcal{H}$ can be also measured by its incorrect prediction of findings. For this reason, we introduce a *false prediction account fp*, which is defined in the following equation:

$$fp(\mathcal{F}_{\mathcal{H}}, \mathcal{F}_{\mathcal{O}}) = \frac{\sum\limits_{F \in \mathcal{F}^{-}_{\mathcal{H},\mathcal{O}}} max\_cs(F, \mathcal{H}) \cdot w_g(F)}{\sum\limits_{F \in \mathcal{F}_{\mathcal{O}}} w_g(F)}\,, \tag{5.17}$$

where $\mathcal{F}^{-}_{\mathcal{H},\mathcal{O}}$ is the set of parametrically observed findings, which are predicted by a diagnosis $D \in \mathcal{H}$ but not observed with the predicted value. It is easy to see that an optimal hypothesis, i.e., a hypothesis best explaining the observed findings, tries to maximize the quality ϱ and to minimize the false prediction account fp.

Data Structures For an efficient evaluation of the hypothesis, we need to consider additional data structures. Any hypothesis contains two sets of findings, $\mathcal{F}^{+}_{\mathcal{H},\mathcal{O}}$ and $\mathcal{F}^{-}_{\mathcal{H},\mathcal{O}}$ for the following reasons: If a diagnosis D is added to a hypothesis $\mathcal{H}$ (line 8), then both sets are updated. The set $\mathcal{F}^{+}_{\mathcal{H},\mathcal{O}}$ contains all findings, for which the predicted value is also observed. Analogously, the set $\mathcal{F}^{-}_{\mathcal{H},\mathcal{O}}$ contains all observed findings $F = Q{:}v$, for which the question Q is observed but with a different value v' than predicted by a diagnosis $D \in \mathcal{H}$. It is worth noticing, that a finding previously contained in the set $\mathcal{F}^{-}_{\mathcal{H},\mathcal{O}}$ can be moved to set $\mathcal{F}^{+}_{\mathcal{H},\mathcal{O}}$, if a diagnosis is added, which covers the finding with the observed value. For example, for a given observed finding $Q{:}v_2$ we define a hypothesis $\mathcal{H} = \{D_1\}$, and D_1 covers finding $Q{:}v_1$. Then, for an evaluation finding $Q{:}v_1$ is added to the set $\mathcal{F}^{-}_{\mathcal{H},\mathcal{O}}$. If a new diagnosis D_2 covering the finding $Q{:}v_2$ is added to the hypothesis $\mathcal{H}$, then the observation $Q{:}v_2$ is moved to the set $\mathcal{F}^{+}_{\mathcal{H},\mathcal{O}}$.

5.5.3. Quantitative Uncertainty in Set-Covering Relations

As mentioned in the introduction of this section, it is difficult for humans to give precise estimations of numerical probabilities. Thus, (untrained) experts fail to differ between small probabilities for an event, e.g., $P(F|D) = 0.1$ vs. $P(F|D) = 0.01$, though they show significant statistical differences. For this reason qualitative covering strengths were introduced in the previous section in order to give an intuitive substitute for probabilities.

However, covering strengths are not sufficient for a formal analysis of set-covering models handling uncertainty. We will therefore introduce an approach, which transforms qualitative covering strengths into numerical probabilities. Then, a given covering strength cs for a covering relation $D \rightarrow F$ is transformed to a conditional covering probability $P(F|D)$. In Figure 5.21 the transformation is depicted graphically.

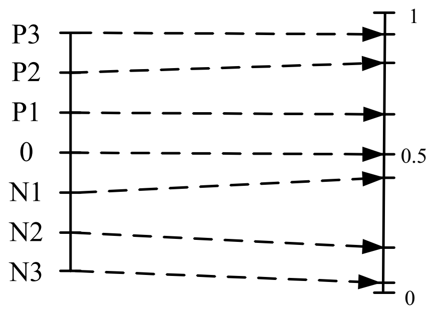


Figure 5.21: Transformation of qualitative covering strengths to numerical probabilities.

For an exact transformation function we have to consider the following issues:

- the covering strength 0 should be mapped to a default probability, which does not significantly alter the probability of the diagnosis.
- positive covering strengths should yield probabilities increasing the probability of the diagnosis.

- negative covering strength should result in probabilities, decreasing the probability of the diagnosis.

The following table depicts an ad-hoc conversion function for covering strengths.

Covering Strength	P3	P2	P1	0	N1	N2	N3
Probability	0.9	0.75	0.6	0.5	0.25	0.1	0.01

Then, qualitative set-covering models can easily transformed into set-covering models with quantitative uncertainty, i.e., set-covering relations with probabilities. In the following, we discuss the handling of probabilities in set-covering models.

Probabilities in Set-Covering Models

We explain the handling of probabilistic uncertainty in set-covering relations by introducing events and causation events, which were inspired by Peng and Reggia [89].

Definition 5.5.4 (Basic Set-Covering Events) For a set-covering model we define the following basic events:

1 A *cause event* $D \in \Omega_{\mathcal{D}}$ is defined as the event that a diagnosis D is present. This event contains no information about the presence/absence of any other diagnosis $D' \in \Omega_{\mathcal{D}} \setminus \{D\}$. We analogously define $\neg D$ as the event that D is absent.

2 An *effect event* $F \in \Omega_{\mathcal{F}}$ defines the event that a finding F is present (observed). The effect event F contains no information about the presence/absence of any other finding $F' \in \Omega_{\mathcal{F}} \setminus \{F\}$. We analogously define $\neg F$ as the event that F is absent (not observed). In this way, an observation set $\mathcal{F}_{\mathcal{O}} = \{F_1, \ldots, F_n\}$ can be interpreted as a conjunction of effect events $F_1 \wedge \cdots \wedge F_n$.

3 A *causation event* $\overrightarrow{DF}$ denotes the event that the finding $F \in \Omega_{\mathcal{F}}$ is actually observed and caused by the diagnosis $D \in \Omega_{\mathcal{D}}$. For a set-covering relation $r = D \rightarrow F$ the causation event $\overrightarrow{DF}$ denotes the realization of r. Analogously we define $\neg\overrightarrow{DF}$ as the absence of $\overrightarrow{DF}$.

We remark, that a causation event $\overrightarrow{DF}$ may still be false, even if the effect event F is true (observed) and the cause event D is true (hypothesized). In this case, F is caused by another diagnosis D' possibly contained in the hypothesis, too. On the other hand, a true causation event $\overrightarrow{DF}$ implies cause event D and effect event F to be true. As we mentioned before, we are able to attach uncertainty to set-covering relations, which are called *conditional causal probabilities*.

Definition 5.5.5 (Conditional Causal Probabilities) The *conditional causal probability* $P(\overrightarrow{DF}|D)$ is the probability, that $F \in \Omega_{\mathcal{F}}$ is caused by $D \in \Omega_{\mathcal{D}}$ utilizing the causation event $\overrightarrow{DF}$. The probability of the absence of a causation event $\neg\overrightarrow{DF}$ is defined as

$$P(\neg\overrightarrow{DF}|D) = 1 - P(\overrightarrow{DF}|D)\,.$$

When we attach uncertainty to set-covering models, we define conditional causal probabilities to set-covering relations.

Definition 5.5.6 (Contexts of Causation Events) Let $\overrightarrow{DF}$ be a causation event for $D \in \Omega_{\mathcal{D}}$ and $F \in \Omega_{\mathcal{F}}$. Then, we call C a context of $\overrightarrow{DF}$, if C is a conjunction of arbitrary cause events and causation events other than $\overrightarrow{DF}$ and $\neg\overrightarrow{DF}$.

Additional Knowledge and Assumptions To facilitate such propositions and their integration in the computation of covering models we need to make the following assumptions.

1 *Independence of cause events:* For each diagnosis $D \in \Omega_{\mathcal{D}}$ the apriori probability $P(D) \in (0;1]$ is given and the cause event D is independent from any other cause event. As a consequence, the apriori probability for a hypothesis $\mathcal{H} = \{D_1, \dots, D_n\}$ is the product of the single probabilities, i.e.,

$$P(\mathcal{H}) = \prod_{D \in \mathcal{H}} P(D) \cdot \prod_{D' \in \Omega_{\mathcal{D}} \setminus \mathcal{H}} \big(1 - P(D')\big)\,.$$

We remark that for a hypothesis $\mathcal{H}$ all diagnoses $D' \in \Omega_{\mathcal{D}} \setminus \mathcal{H}$ are assumed to be absent.

2 *Independence of causation events:* For each covering relation $r = D \to F$ defined in the set-covering model the non-zero conditional causal probability $P(\overrightarrow{DF}|D) \in (0;1]$ is given. If a cause event $D \in \Omega_{\mathcal{D}}$ happens, then the causation event $\overrightarrow{DF}$ occurs independently of any context C with $P(D \wedge C) > 0$, i.e.,

$$P(\overrightarrow{DF}|D \wedge C) = P(\overrightarrow{DF}|D).$$

3 *Completeness assumption:* For each question $Q \in \Omega_Q$ there need to exist at least one set-covering relation $r = D \to Q{:}v \in \mathcal{R}$ with $v \in dom(Q)$.

It is easy to see that a set-covering model applying uncertainty information requires $|\Omega_{\mathcal{D}}| + |\mathcal{R}|$ probabilities. With the completeness assumption we can guarantee that there exists a causation event for any observed question. For handling multiple faults (i.e. a set of diagnoses is explaining an observation set), we have to consider *composite causation events*.

Definition 5.5.7 (Composite Causation Events) Let $F \in \Omega_{\mathcal{F}}$ be an observed finding and let $\mathcal{H} \subseteq \Omega_{\mathcal{D}}$ be a hypothesis. Then, $\overrightarrow{\mathcal{H}F}$ is defined as the *composite causation event*. The event $\overrightarrow{\mathcal{H}F}$ describes that F is caused by at least one diagnosis $D \in \mathcal{H}$.

When computing the probability of a composite causation event $\overrightarrow{\mathcal{H}F}$ we have to take the disjunction of every subset of $\mathcal{H}$ for causing F into account.

For example, for the set-covering model given in Figure 5.20, the hypothesis $\mathcal{H} = \{D_1, D_2\}$ and the finding $F = Q{:}v_3$ we want to compute the conditional probability of the composite causation event $\overrightarrow{\mathcal{H}F}$. Thus, we have to consider

$$\begin{aligned} P(\overrightarrow{\mathcal{H}F}|\mathcal{H}) &= P(\overrightarrow{D_1\neg D_2F}|\mathcal{H}) + P(\overrightarrow{\neg D_1D_2F}|\mathcal{H}) + P(\overrightarrow{D_1D_2F}|\mathcal{H}) = \\ &= P(\overrightarrow{D_1F}|\mathcal{H}) \cdot \big(1 - P(\overrightarrow{D_2F}|\mathcal{H})\big) + \\ &\quad + P(\overrightarrow{D_2F}|\mathcal{H}) \cdot \big(1 - P(\overrightarrow{D_1F}|\mathcal{H})\big) + \\ &\quad + P(\overrightarrow{D_1F}|\mathcal{H}) \cdot P(\overrightarrow{D_2F}|\mathcal{H})\,. \end{aligned}$$

In general, the probability for a composite causation event is computed as follows:

$$P(\overrightarrow{\mathcal{H}F}|\mathcal{H}) = \sum_{\mathcal{H}'\subseteq\mathcal{H},\mathcal{H}'\neq\{\}} \Big(\prod_{D\in\mathcal{H}'} P(\overrightarrow{DF}|D) \cdot \prod_{D\in\mathcal{H}\setminus\mathcal{H}'} \big(1 - P(\overrightarrow{DF}|D)\big)\Big) \tag{5.18}$$

Hypothesis Evaluation For the evaluation of a hypothesis $\mathcal{H}$ for a given observation $\mathcal{F}_\mathcal{O}$ we define the quality measure for set-covering models with probabilities.

The quality measure applies not only probabilities to set-covering relations but also additional knowledge components like similarities and (global) weights. Each component supplies an additional support for the calculation of the quality measure, if it is available in the given model. However, if one component does not appear, it cannot contribute to the quality of a hypothesis and therefore will not appear in the calculation. For this reason we will introduce the abbreviating functions wc, sc, and pc. If the corresponding knowledge is available, then we set

$$\begin{aligned} wc(Q) &= w_g(Q)\,, \\ sc(Q) &= sim\big(f(Q,\mathcal{F}_\mathcal{H}), f(Q,\mathcal{F}_\mathcal{O})\big)\,, \\ pc(Q) &= P(\mathcal{H}) \cdot P_{\mathcal{H},\mathcal{F}_\mathcal{O}}(Q{:}v) \quad \text{with } Q \in a(\mathcal{F}_\mathcal{H} \cap \mathcal{F}_\mathcal{O}) \wedge v = val_{\mathcal{F}_\mathcal{O}}(Q)\,, \end{aligned}$$

where $f(Q,\mathcal{F})$ is a function that returns the finding $F \in \mathcal{F}$ containing the question Q, and function $P_{\mathcal{H},\mathcal{F}_\mathcal{O}}$ is defined as follows:

$$P_{\mathcal{H},\mathcal{F}_\mathcal{O}}(F) = \begin{cases} P(\overrightarrow{\mathcal{H}F}|\mathcal{H}) & \text{for } F \in \mathcal{F}_\mathcal{O}\,, \\ 1 - P(\overrightarrow{\mathcal{H}F}|\mathcal{H}) & \text{otherwise}\,. \end{cases} \tag{5.19}$$

The (local) similarity function sim is defined according to Definition 5.2.1.

If the corresponding knowledge is not available, then we set $wc(Q) = sc(Q) = pc(Q) = 1$, i.e., 1 is the default value.

The quality of a hypothesis is given by the following equation

$$\varrho(\mathcal{F}_\mathcal{H}, \mathcal{F}_\mathcal{O}) = \frac{\sum_{Q\in a(\mathcal{F}_{\mathcal{H},\mathcal{O}})} wc(Q) \cdot pc(Q) \cdot sc(Q)}{\sum_{Q\in a(\mathcal{F}_\mathcal{O})} wc(Q)}\,. \tag{5.20}$$

For the expert these equations serve as an intuitive understanding of the model. A more appropriate procedure for handling uncertainty would be the introduction of a *leak-diagnosis*. A leak-diagnosis D_l captures the idea that no model can be a complete view of the domain, and that there are always other reasons that can cause a given finding. These “other reasons” are collected in the leak-diagnosis, which is categorically connected to all available findings. To shrink the emerging number of probabilities, we assume a constant probability for all set-covering relations between the leak-diagnosis and a finding. If the model contains weights, then it is easy to see that the leak probabilities can be adapted with respect to the weights. Large weights will decrease the leak probability whereas small weights increase the probability. As a consequence, for every hypothesis we have to consider the leak-diagnosis to be included in the hypothesis as well. It is easy to understand that with the usage of the leak-diagnosis there will be no isolated observed parameters because the leak-diagnosis holds covering relations to all findings by default. So $\mathcal{F}_{\mathcal{O}}^{iso}$ will be empty for all $\mathcal{H}$ and all $\mathcal{F}_{\mathcal{O}}$. In [16] we presented an incremental approach augmenting set-covering models with additional knowledge components.

5.5.4. A Bayesian Transformation for Set-Covering Models

Set-covering models allow for an intuitive representation of uncertainty for diagnostic reasoning. Furthermore, the knowledge acquisition costs can be step-wise increased according to the required reasoning accuracy, e.g., by introducing additional knowledge like similarities and weights.

Bayesian networks [88] are the state-of-the-art representation of uncertain knowledge in current artificial intelligence research. The main advantage of Bayesian networks are their well defined semantics when considering the reasoning capabilities and acquisition methods. However, if we postulate the self-acquisition of knowledge systems, then the application of Bayesian networks is still problematic. Experience has shown, that mathematically untrained domain specialists find it very difficult to represent their diagnostic knowledge in the Bayesian formalism. For this reason, we propose to start with a set-covering model and, if required, switch to a Bayesian network. E.g., a representation change can become necessary, if the knowledge should be shard with other Bayesian knowledge systems or an even more exact and powerful specification of uncertain knowledge is required.

Consequently, we sketch an approach for transforming a set-covering model into a Bayesian network. If we compare the representational approaches of set-covering models and Bayesian networks, we identify the following similarities:

Graphical representation In both approaches, diagnoses and findings are represented as nodes in a graph-like structure (mostly, a DAG), and the dependencies between the nodes are described by arcs. Set-covering models only allow for a boolean representation of nodes.

Uncertainty In Bayesian networks uncertainty is represented by the exact specification of conditional probability tables (CPT); the complexity of CPTs can be reduced by simplifying assumptions using NOISY-functions [37].

Independency In Bayesian networks independence between nodes is determined according to the d-separation criterion, e.g., see [88]. For set-covering models we categorically assume that the occurrence of multiple diagnoses not connected by causal arcs is independent, as well as the observation of arbitrary findings for different questions given no diagnosis at all.

The general idea of transforming set-covering models into Bayesian networks is as follows: Questions contained in the set-covering model are mapped to discrete nodes in the corresponding Bayesian network, and diagnoses are mapped to boolean nodes included in the network. Existing set-covering relations are mapped to arcs in the Bayesian network, and the attached uncertainties of the set-covering relations are inserted into the corresponding cells of the CPTs. The remaining probabilities are computed according to the composite causation event for probabilities for combined diagnoses, and by using similarity knowledge for values of the question not specified in a set-covering relation. For Bayesian networks a probability can be specified for the event that a value for the question is observed although no defined diagnoses is occurring. This *leak*-probability is not explicitly defined in set-covering models, but we apply defined weights and abnormality knowledge of questions for deriving the leak-probabilities of the corresponding findings. Then, the leak-probability of a finding is the smaller the higher the weight of the corresponding question is, and the higher the abnormality of the finding value is.

In the best case, either all possible or no set-covering relations between a diagnosis and a question are defined, i.e., for each value of question there exists a diagnosis–finding relation. Then, a direct transformation of the probabilities into a CPT is possible. For incompletely specified set-covering models several approaches were described and evaluated in [23], in which the considerations sketched above are also explained and motivated in more detail.

5.5.5. Acquisition of Set-Covering Models

The typical procedure for acquiring set-covering knowledge is given as follows: Often it is useful to firstly define an abstraction layer as described in Section 5.1.2. Abstractions for set-covering models are often defined by abstraction rules. Then, set-covering knowledge is stepwise acquired by a diagnosis-centered approach: For each diagnosis corresponding set-covering relations are defined. For the definition of the set-covering model we propose an incremental process as introduced in [16]: We start with a simple model describing the coarse structure between diagnoses and findings. Further, we define relations between the diagnosis and its typical findings, i.e., the findings that are mostly observed for the given diagnosis. The set-covering relations for the remaining question values of the specified findings are interpolated by similarity knowledge between finding values. Of course, the developer can manually specify these set-covering relations in order to increase the accuracy of the model. Subsequently, the simple model can be enhanced by weight knowledge and confirmation strengths to further increase the accuracy of the model and the resulting system.

In order to reduce knowledge acquisition costs we provide learning methods for the automatic generation of set-covering relations, covering strengths, similarity functions, and weights. For more details we refer to Section 5.5.8.

Since the development of the set-covering model is an evolutionary process, we propose restructuring methods and test measures. E.g., restructuring methods for the set-covering model consider the propagation of ontological restructurings, and the introduction and deletion of diagnosis layers in the set-covering model (see Section 5.5.7). A model can be tested for its correctness, for anomalies (redundancy, ambivalence, circularity, deficiency), for its robustness, and for its understandability. For more details we refer to Section 5.5.6. Finally, we offer a transformation method for converting a set-covering model to a Bayesian network. This probabilistic representation rarely fits the mental model of a domain expert, but provides many advantages: There are various well-investigated learning methods for Bayesian networks available, which can be applied to improve an existing model if a sufficient number of cases become available due to a productional use. Furthermore, Bayesian networks have a clearly defined semantics of uncertainty, and efficient reasoning methods are available.

5.5.6. Testing Set-Covering Knowledge

In this section we discuss methods for testing set-covering knowledge for its understandability, anomalies and its robustness.

Static SC-Model Analysis

With the *Static SC-Model Analysis* method the developer can investigate the understandability of the implemented set-covering model.

Mechanics The method simply counts the number of set-covering relations for each diagnosis and calculates a mean value with standard deviation. If the standard deviation exceeds a specified threshold, then diagnoses with border values are reported as a warning. Furthermore, the method computes for each finding the mean value (with standard deviation) of diagnoses predicting the finding. If the standard deviation exceeds a given threshold, then the findings with border values are reported as a warning.

Usage The method is used for testing the balance of the set-covering model. The balance of a set-covering model can increase the understandability of the implemented knowledge. Thus, diagnoses and findings with significant few or significant many relations are presented to the user. For this method no additional test knowledge is required, except for the specification of threshold values for die maximum standard deviation of set-covering relations and covering diagnoses.

Static SC-Model Verification

The static verification method was presented for rule-based knowledge in Section 5.4.4, and can be also applied for set-covering models. Analogously to the rule-based method anomalies in the structural knowledge container are detected. For set-covering models we can distinguish the following anomalies:

Redundancy	The set-covering model contains redundant knowledge, e.g., subsumed set-covering relations, redundant set-covering relations.
Ambivalence	The model contains set-covering relations that are contradictory with respect to constraints contained in the model.
Circularity	The set-covering model contains set-covering relations that produce a circle.
Deficiency	The knowledge base contains findings or diagnoses that are not used by any set-covering relation. Alternatively, a deficiency is caused by missing similarity or weight knowledge.

Mechanics The following table gives an overview of the anomalies, that can be detected by the static sc-model verification.

► **Redundancy**	
Subsumed Set-Covering Relation A set-covering relation $r = D \rightarrow F$ is subsumed by another set-covering relation $r' = D' \rightarrow F$, if D' is (transitively) covered by D.	warning
Redundant Set-Covering Relation (exact) A set-covering relation $r = D \rightarrow F$ is redundant, if there also exists another set-covering relation $r' = D \rightarrow F$.	warning
Redundant Set-Covering Relation (fuzzy) A set-covering relation $r = D \rightarrow F$ is fuzzy-redundant, if there exists a set-covering relation $r' = D \rightarrow F'$ with $sim(F, F') \geq \mathcal{T}$, where $\mathcal{T}$ is a given threshold.	warning
► **Ambivalence**	
A set-covering relation $r = D \rightarrow F$ is ambivalent, if there exists an exclusion condition with $(\neg D \wedge F)$.	error
► **Circularity**	
There exists a set of set-covering relations, so that for a two diagnoses $D, D' \in \Omega_D$ the diagnosis D is (transitively) covered by D' and the diagnosis D' is (transitively) covered by D.	error
► **Deficiency**	
Missing Weight/Similarity There exists a question with no weight knowledge or no similarity knowledge, but the question is assigned in at least one set-covering relation.	warning
Missing Set-Covering Relation There exists a diagnosis or a question, which is not contained in any set-covering relation of the model.	warning

Usage With the static verification method anomalies can be detected in set-covering models. Most of the method are quite simple, but provide a valuable contribution to the agile development process. For example, it can be difficult to detect incompletely defined objects for a large structural knowledge container. However, incompleteness means a deficiency of the implemented structural knowledge container. For the static verification of set-covering models no additional test knowledge is needed, with exception to the detection of fuzzy-redundant set-covering relations, which requires a threshold for the maximum similarity of redundant findings.

Torture Tests for Set-Covering Models

Analogously to the rule-based approach, we can define torture tests for a set-covering model to determine its robustness. Then, we define the *reduce-knowledge* and the *modify-knowledge* methods for set-covering models.

Mechanics Starting with the complete set-covering model the *reduce-knowledge* approach systematically removes set-covering relations until a given threshold $\mathcal{T}_{SR}$, defining the maximum percentage of (randomly) removed relations, is reached. After each degradation, the currently modified knowledge base is applied to test cases and the mean accuracy is calculated to document the diagnostic performance during the degradation. If the mean value of the calculated accuracy falls below a defined threshold $\mathcal{T}_F$, then an error is reported.
The *modify-knowledge* approach does not reduce the size of the set-covering model, but modifies single set-covering relations contained in the model. Thus, confirmation strengths of contained set-covering relations are randomly selected and slightly increased or decreased. Then, the diagnostic performance of the modified system is determined by the accuracy during the degradation. For this approach, a defined threshold $\mathcal{T}_{MR}$ describing the maximum number of modified set-covering relations, and a defined threshold $\mathcal{T}_F$ is required. If the mean accuracy computed by the degradation study falls below the threshold $\mathcal{T}_F$, then an error is reported. The implementation and evaluation of torture tests for set-covering models is analogous to the torture tests presented in Section 5.6.4.

Usage This methods provides a fine-grained method for testing the robustness of structural knowledge formalized by set-covering models.
Then, a sufficiently large collection of test cases is required, as well as the thresholds $\mathcal{T}_F$, $\mathcal{T}_{SR}$, and $\mathcal{T}_{MR}$ describing the minimum diagnostic accuracy, the maximum number of removed set-covering relations, and the maximum number of modified set-covering relations, respectively.

5.5.7. Restructuring of Set-Covering Models

In the following, we sketch the propagation of ontological restructuring methods with respect to set-covering models. In Section 4.6 we introduced restructuring methods for the ontological knowledge container. These methods also trigger corresponding methods for restructuring the implemented set-covering model.

The method TRANSFORMYNINTOMC is simply propagated by removing all set-covering relations to the original yes/no questions and by creating a new set-covering linking the newly created multiple-choice question. Analogously, the TRANSFORMMCINTOYN is propagated by removing all set-covering relations to the original multiple-choice question from the set-covering model, and by inserting new set-covering relations linking the newly created yes/no questions. Both methods will cause no conflict in general.

Also the SHRINKVALUERANGE method is always applicable: The set-covering relations with findings of the specified question are modified according to the transformation matrix defined for the method.

The method TRANSFORMNUMINTOOC can only be executed, if the original numerical question can be split up into distinct partitions, that are representing the new choice answers. Then, the set-covering relations specified for the original numerical questions are replaced by set-covering relations linking the newly created one-choice findings. If the numerical value range cannot be partitioned, then the method needs to be aborted.

The TRANSFORMMCINTOOC method can be executed without conflicts, if no set-covered finding of the multiple-choice question contains multiple values. Then, the multiple choice question is simply replaced with the newly created one-choice question. Otherwise, the method needs to be aborted.

The methods REMOVEDIAGNOSIS and REMOVEQUESTION can only be applied, if the specified diagnosis or question is contained in no set-covering relation of the model. This is simply detected as deficiency by the static verification of set-covering models introduced in Section 5.5.6. Otherwise, the conflicting set-covering relations need to be interactively removed or re-linked in collaboration with the developer.

5.5.8. Learning Set-Covering Models

In the following, we describe an algorithm for learning set-covering models from cases, which initially was presented by Baumeister et al. in [14], and was evaluated on a real life case base.

A set-covering model usually contains a set of set-covering relations between diagnoses and findings, and additional knowledge about finding similarities and weights. Since the similarity function and weight measure for set-covering models is equivalent to the functions defined for case-based reasoning, we refer to the algorithms given in Section 5.2.8 (p. 81). In this section we focus on learning set-covering relations. In general, the algorithm is divided into three main steps: Firstly, diagnostic profiles for each diagnosis are generated according to the case base given as training samples. Diagnostic profiles are augmented frequency profiles, containing all co-occurring findings for each diagnosis together with co-occurring diagnoses. Then, the profiles are refined and filtered so that only the most frequent findings for the particular diagnoses are contained in the profiles. The reduced profiles are applied to build set-covering relations in the third step. For each finding contained in a diagnostic profile a set-covering relation is generated.

Building Diagnostic Profiles

A diagnostic profile is an augmented frequency profile, which was defined in Definition 5.4.2.

Definition 5.5.8 (Diagnostic Profile) A diagnostic profile $DP_{CB}(D)$ for a diagnosis $D \in \Omega_{\mathcal{D}}$ contained in a case base CB is defined as a tuple

$$DP_{CB}(D) = \big(\; FP_{CB}(D), \mathcal{D}_{cor} \,\big)\,,$$

where $FP_{CB}(D)$ is the frequency profile for diagnosis D, and $\mathcal{D}_{cor}$ is a set of tuples containing diagnoses D',

$$\mathcal{D}_{cor} = \big\{ \big(D', freq_{CB}(D', D) \big) \,\big|\, D' \in \Omega_{\mathcal{D}} \,\wedge\, freq_{CB}(D', D) \in (0, 1] \big\}\,,$$

that are co-occurring with D, attached with their frequencies.

The set of corresponding diagnoses can help the developer to compare similar set-covering models for frequently corresponding diagnoses, since coherent diagnoses typically cover the similar findings.

Refining the Diagnostic Profiles

Initially, the diagnostic profiles contain *all* findings the particular diagnoses co-occur with. Obviously, it is reasonable to filter out very unfrequent associations between the diagnoses and the findings.

Before the filter step, we will consider similarities between the findings contained in the profile. For example, if a profile for diagnosis D includes the finding $temperature{:}high$ ($T{:}h$) with frequency 0.4 and the finding $temperature{:}very\ high$ ($T{:}vh$) with frequency 0.4, then both findings might be too unfrequent to remain in the profile. But since both findings are very similar to each other, an adapted frequency may be sufficiently frequent to remain in the profile. Then, if $sim(T{:}h, T{:}vh) = 0.8$, then an adapted frequency $freq'_{CB}(D, T{:}h)$, concerning similar findings is

$$\begin{aligned} freq'_{CB}(D, T{:}h) &= freq_{CB}(D, T{:}h) + \big(freq_{CB}(D, T{:}vh) \cdot sim(T{:}h, T{:}vh)\big) \\ &= 0.4 + (0.8 \cdot 0.4) = 0.72\,. \end{aligned}$$

We adapt this idea when we firstly compute an adapted frequency $freq'_{CB}(D, F)$ for each finding F in the profile regarding similarities between finding values of the same question. After adapting the frequencies we remove all findings from the profile, which are still too unfrequent with respect to a given threshold $\mathcal{T}_{\lambda}$. We remark that a given diagnostic profile may contain more than one finding for a given question, if their adapted frequencies exceed the specified threshold. The resulting *reduced diagnostic profile* is defined as follows:

Definition 5.5.9 (Reduced Diagnostic Profile) A reduced diagnostic profile $DP^{*}_{CB}(D)$ for a diagnosis $D \in \Omega_{\mathcal{D}}$ contained in a case base CB is defined as a tuple

$$DP^{*}_{CB}(D) = \big(\; FP^{\lambda}_{CB}(D), \mathcal{D}_{cor} \,\big)\,,$$

where $\mathcal{D}_{cor}$ is a set of corresponding diagnoses of D, and $FP^{\lambda}_{CB}(D)$ is the augmented frequency profile with

$$FP^{\lambda}_{CB}(D) = \big\{ \big(F, \mathit{freq}'_{CB}(D,F)\big) | F \in \Omega_{\mathcal{F}} \wedge \mathit{freq}'_{CB}(D,F) > \mathcal{T}_{\lambda} \big\} \,.$$

Formally, the adapted frequency $\mathit{freq}'_{CB}(D,F)$ is computed by

$$\mathit{freq}'_{CB}(D,F) = \mathit{freq}_{CB}(D,F) + \sum_{F' \in \mathcal{F}_{FP_{CB}(D)}} \mathit{freq}_{CB}(D,F') \cdot sim(F,F') \,,$$

where $\mathcal{F}_{FP_{CB}(D)}$ is the set of all finding contained in the frequency profile of diagnosis D.

If the similarity function is not defined for two findings $F, F' \in \Omega_{\mathcal{F}}$, then we set the default similarity $sim(F,F') = 0$.

Generating Set-Covering Relations

Based on the reduced diagnostic profiles computed in the previous step we generate set-covering relations between diagnoses and findings. For each reduced diagnostic profile we generate a set-covering relation $D \rightarrow F$ for each finding F contained in the diagnostic profile for D. The frequency of the association between the diagnosis D and the finding F can be attached as numerical covering probability defined in Section 5.5.3, if quantitative uncertainty should be applied to the set-covering model. To increase the understandability of the learned results we alternatively can transform the frequency into a symbolic covering strength as defined in Section 5.5.2. An appropriate mapping function is required, which e.g., was presented in Figure 5.18 for score-based knowledge.

It is worth mentioning that the frequency threshold $\mathcal{T}_{\lambda}$ directly corresponds to the number of generated set-covering relations. For a high threshold we will generate a sparse set-covering model with only few set-covering relations. A low threshold will result in a dense set-covering model containing many set-covering relations. Therefore, the developer has to find a trade-off between too special and too general set-covering models.

5.5.9. Summary

In this section, we have introduced set-covering models for representing diagnostic structural knowledge. Set-covering models allow for an intuitive mapping of expert's mental models into a processable representation. They can handle similarity measures and weights for findings, and they are able to state uncertainty either symbolically or numerically. As a special characteristic set-covering models are suitable for handling domains with multiple faults. We concluded this section by introducing a method for learning set-covering models from a case base.

Set-covering models can be extended by introducing an additional layer for abstract findings. Thus, not only diagnoses can cover findings, but also abstract findings can cover more specific findings defining an *is-a* relationship. Diagnoses may cover only abstract findings that are observed if at least one of the covered specific findings is observed. Using such an abstraction layer more general observable concepts can be represented with set-covering models.

5.6. Black-Box Testing of Structural Knowledge

The evaluation of structural knowledge is one of the most important tasks during knowledge system development, since it uncovers errors and checks the inferential completeness of the knowledge system. In the preceding sections we have presented several approaches for representing structural knowledge. Consequently, we identified and discussed different testing methods for the particular approaches. However, some testing methods can be generally applied and do not rely on the underlying knowledge representation. These kinds of methods are commonly called *black-box* testing methods (functional testing), whereas methods linked with the underlying knowledge representation are called *white-box* testing methods (structural testing).

In the following, we introduce black-box testing methods, that can be generally applied. The following black-box testing methods for structural knowledge are discussed: empirical testing, sequentialized empirical testing, inferential constraints, and torture tests. These methods do not rely on the underlying knowledge representation of structural knowledge.

5.6.1. Empirical Testing

The most popular kind of black-box testing is *empirical testing*: In a first step previously solved test cases are selected. In a second step the implemented knowledge system runs the test cases and infers a solution for each case. The inferred solution is compared with the stored solution of the case and differences are presented to the user.

Mechanics The task can be decomposed in three sub-tasks: Test case selection, test case application, and evaluation of the results. The scheme is depicted in Figure 5.22. In the first step an appropriate collection of test cases is selected. On the one hand, for

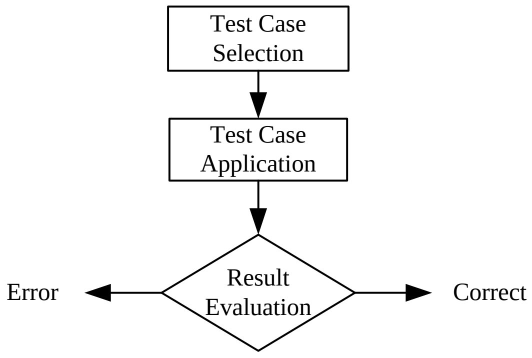


Figure 5.22: Scheme for the empirical testing task.

the agile development we suggest to include a number of test cases for each diagnoses included in the system. On the other hand, the size of the resulting case base should not exceed a given threshold, since running large empirical tests after each change of the knowledge base can evolve to be impractical and time consuming. Therefore, it is suggestive to include a small number of test cases covering all diagnoses into the working test suite and to add the remaining test cases into the integration test suite, which is only executed before the integration step.

The test case application task is simple: For each test case c a knowledge system run is started with a new case c'. The new case c' receives the problem description $\mathcal{F}_c$ of the stored case and infers a solution $\mathcal{D}_{c'}$ using the implemented inferential knowledge.

After the test case application has been finished, the result is analyzed: For each solved test case c' the inferred solution $\mathcal{D}_{c'}$ is compared with the stored solution $\mathcal{D}_c$. This analysis reveals the correctness of the knowledge system, and several approaches for interpreting the results have been proposed.

The measures *precision* and *recall* are well-known approaches for determining the accuracy of the system.

Definition 5.6.1 (Precision and Recall) Let $c = (\mathcal{F}_c, \mathcal{D}_c, \mathcal{I}_c)$ be the stored case and $c' = (\mathcal{F}_{c'}, \mathcal{D}_{c'}, \mathcal{I}_{c'})$ the new inferred case, with $\mathcal{F}_c = \mathcal{F}_{c'}$. Then, the *precision* is defined by

$$precision(\mathcal{D}_c, \mathcal{D}_{c'}) = \begin{cases} \frac{|\mathcal{D}_c \cap \mathcal{D}_{c'}|}{|\mathcal{D}_{c'}|} & \text{if } D_{c'} \neq \{\}\,, \\ 1 & \text{if } D_{c'} = \{\} \text{ and } D_c = \{\}\,, \\ 0 & \text{otherwise.} \end{cases}$$

The precision calculates the degree of inferred diagnoses that are actually correct. The *recall* is defined by

$$recall(\mathcal{D}_c, \mathcal{D}_{c'}) = \begin{cases} \frac{|\mathcal{D}_c \cap \mathcal{D}_{c'}|}{|\mathcal{D}_c|} & \text{if } D_c \neq \{\}\,, \\ 1 & \text{otherwise.} \end{cases}$$

The recall measures the degree of correct solutions that are actually inferred.

In the medical domain the measures *sensitivity* and *specificity* are commonly used. The sensitivity or *true-positive rate* is defined as the likelihood that the disease of a patient is actually derived by the system. The specificity or *true-negative rate* is defined as the likelihood that for a healthy patient actually no diagnosis is derived. Commonly, these two measures are defined for single diagnoses, e.g., Shortliffe et al. [121, p.90ff]; in the following we adapt the measures to the multiple-fault problem in diagnostic knowledge systems.

Definition 5.6.2 (Sensitivity and Specificity) Let $c = (\mathcal{F}_c, \mathcal{D}_c, \mathcal{I}_c)$ be the stored case and $c' = (\mathcal{F}_{c'}, \mathcal{D}_{c'}, \mathcal{I}_{c'})$ the new inferred case, with $\mathcal{F}_c = \mathcal{F}_{c'}$. Then, the *sensitivity* measures the count of correctly derived diagnoses, and is defined by

$$sensitivity(\mathcal{D}_c, \mathcal{D}_{c'}) = \begin{cases} \frac{|\mathcal{D}_c \cap \mathcal{D}_{c'}|}{|\mathcal{D}_c|} & \text{if } \mathcal{D}_c \neq \{\}\,, \\ 1 & \text{otherwise.} \end{cases}$$

The sensitivity is equivalent to the previously defined recall measure. The *specificity* measures the count of incorrect diagnoses that are also not derived, and is defined by

$$specificity(\mathcal{D}_c, \mathcal{D}_{c'}) = \frac{|\Omega_{\mathcal{D}} \setminus \mathcal{D}_c|}{|\Omega_{\mathcal{D}} \setminus \mathcal{D}_c| + |\mathcal{D}_c' \setminus \mathcal{D}_c|}\,.$$

Thompson and Mooney [36] propose the *intersection accuracy* for the evaluation of cases with multiple diagnoses in the case.

Definition 5.6.3 (Intersection Accuracy) Let $c = (\mathcal{F}_c, \mathcal{D}_c, \mathcal{I}_c)$ be the stored case and $c' = (\mathcal{F}_{c'}, \mathcal{D}_{c'}, \mathcal{I}_{c'})$ the new inferred case, with $\mathcal{F}_c = \mathcal{F}_{c'}$. The *intersection accuracy iacc* is defined by

$$iacc(\mathcal{D}_c, \mathcal{D}_{c'}) = \frac{precision(\mathcal{D}_c, \mathcal{D}_{c'}) + recall(\mathcal{D}_c, \mathcal{D}_{c'})}{2},$$

which averages the precision and recall of a retrieved solution.

However, experience has shown that the *F-measure* known from the information extraction theory and data mining, e.g., [138, p. 146], is appropriate for comparing solutions with multiple faults.

Definition 5.6.4 (F-Measure) Let $c = (\mathcal{F}_c, \mathcal{D}_c, \mathcal{I}_c)$ be the stored case and $c' = (\mathcal{F}_{c'}, \mathcal{D}_{c'}, \mathcal{I}_{c'})$ the new inferred case, with $\mathcal{F}_c = \mathcal{F}_{c'}$ and let β be a constant. The *F-measure* is defined as follows:

$$f(\mathcal{D}_c, \mathcal{D}_{c'}) = \frac{(\beta^2 + 1) \cdot precision(\mathcal{D}_c, \mathcal{D}_{c'}) \cdot recall(\mathcal{D}_c, \mathcal{D}_{c'})}{\beta^2 \cdot precision(\mathcal{D}_c, \mathcal{D}_{c'}) + recall(\mathcal{D}_c, \mathcal{D}_{c'})}$$

The F-measure computes the geometric mean of the precision and the recall of a retrieved solution. In the context of the F-measure commonly $\beta = 1$.

The following example will clarify the difference between the presented measures: Let $c = (\mathcal{F}_c, \{D_1, D_2, D_3\}, \mathcal{I}_c)$ be the test case, and let $c' = (\mathcal{F}_{c'}, \{D_1, D_2\}, \mathcal{I}_{c'})$ be the inferred case, i.e., only the two first diagnoses have been inferred by the knowledge system. Then, for $|\mathcal{D}_c| = 3$, $|\mathcal{D}_{c'}| = 2$, and $\beta = 1$ we obtain the following measures: $precision(\mathcal{D}_c, \mathcal{D}_{c'}) = 2/2 = 1$, $recall(\mathcal{D}_c, \mathcal{D}_{c'}) = 2/3 = 0.66$, $iacc(\mathcal{D}_c, \mathcal{D}_{c'}) = 0.5 \cdot (1 + 0.66) = 0.83$, and $f(\mathcal{D}_c, \mathcal{D}_{c'}) = (2 \cdot 1 \cdot 0.66)/(1 \cdot 1 + 0.66) = 0.795$.
The example shows that neither the precision nor the recall is an appropriate measure for the evaluation of case solutions; it is worth noticing, that the recall would yield improper results, if the solutions set had been exchanged against each other. The intersection accuracy also is not appropriate, since the arithmetic mean yield no meaningful measure, e.g., for $precision(\mathcal{D}_c, \mathcal{D}_{c'}) = 1$ and $recall(\mathcal{D}_c, \mathcal{D}_{c'}) = 0$ would still yield the $iacc(\mathcal{D}_c, \mathcal{D}_{c'}) = 0.5$. The F-measure finally denotes an appropriate measure. For this reason, the presented agile process model uses the F-measure for evaluating the results of the empirical testing.
As depicted in Figure 5.22 the analysis concludes with an error or correctness message. An error is reported, if one of the test cases yields a F-measure less than 1, i.e., if at least one test case has not been correctly solved. Whereas this behavior can sensible for the working test suite during the development of the knowledge base, we admit a less strict guideline for the integration test suite. Since, the integration test suite usually contains real life cases, it is very common that not all cases can be solved correctly by the knowledge system. Therefore, we introduce a threshold value for the F-measure. If the F-measure for the test cases contained in the integration test suite does not fall below the given threshold, then no error is reported. Otherwise, we present the falsely solved cases in the error message.

Hierarchical Relations of Diagnosis Set As described earlier in Chapter 4 the ontological knowledge container usually contains hierarchical relationships between diagnoses; mostly these relationships describe parent-child relations with type *is-a*. This concept can be used for an extended definition of the presented accuracy measures. Then, we do not simply compare the occurrences of a diagnosis in the retrieved and the stored solution set, respectively, but also consider similarities between different diagnoses.

For computing the measures we do not simply interset the solutions sets, but compare the included diagnoses according to the following similarity function. Let $\mathcal{D}$ be the set of correct diagnoses contained in the stored case, and $\mathcal{D}'$ be the set of derived diagnoses contained in the current case. We define the similarity-based intersection as follows.

Definition 5.6.5 (Similarity-Based Intersection) Let $\mathcal{D}, \mathcal{D}' \subseteq \Omega_{\mathcal{D}}$ two diagnosis sets; the *similarity-based intersection* is defined by

$$\bigcap_{sim}(\mathcal{D}, \mathcal{D}') = \frac{\sum\limits_{D \in \mathcal{D} \cup \mathcal{D}'} dsim(D, \mathcal{D}, \mathcal{D}')}{|\mathcal{D} \cup \mathcal{D}'|} ,$$

where $dsim : \Omega_{\mathcal{D}} \times \Omega_{\mathcal{D}} \rightarrow [0, 1]$ is function comparing two diagnoses. A possible definition for $dsim$ is

$$dsim(D, \mathcal{D}, \mathcal{D}') = \begin{cases} 1 & \text{if } D \in \mathcal{D} \cap \mathcal{D}' , \\ 0.5 & \text{if } D \in \mathcal{D}' \wedge D \in p(\mathcal{D}') , \\ 0.5 & \text{if } D \in \mathcal{D}' \wedge D \in c(\mathcal{D}') , \\ 0 & \text{otherwise.} \end{cases}$$

The function p returns the set of parent diagnoses of the specified diagnosis, and the function c determines the set of diagnoses representing the children of the specified diagnosis.

The function $dsim$ may be adjusted with respect to the requirements of the application domain. If the knowledge system not only derives established diagnoses but also suggested diagnoses, then these can be compared with the expected solution set in a similar manner.

Usage The empirical testing method is applied for testing the correctness of the diagnostic system. Then, malfunction of the system is detected by inferring wrong or less diagnoses than predicted by the test cases. We can distinguish empirical testing in context of the working test suite and included in the integration test suite. For the working test suite we commonly assume to achieve fully correct behavior with a total F-measure equal to 1. For the integration test we can loosen this assumption by asking for a F-measure less than 1. Obviously, empirical testing requires test cases as test knowledge.

5.6.2. Sequentialized Empirical Testing

Usual empirical testing as introduced above focusses on the accuracy of the solved cases. The cases are completely processed by the knowledge system, and only the final result of the case is compared with the expected solution of the test case. *Sequentialized empirical testing* is not only interested in the comparison of the final solution, but also considers the inspection of intermediate results during case processing. In general, sequentialized empirical testing is a generalization of the previously presented empirical testing method.

Mechanics Sequentialized empirical testing applies a set of sequentialized cases. A sequentialized test case itself is an ordered sequence of cases.

Definition 5.6.6 (Sequentialized Case) A sequentialized case is defined as an ordered list of cases

$$c^* = (c_1, \ldots, c_n)\,,$$

where $c_i \in \Omega_C$ are (sub-)cases describing the sequence of observations stored in the case c^*. For each (sub-)case $c_i \in c^*$ an intermediate solution $\mathcal{D}_{c_i}$ is given, that defines the temporary solution of the sequentialized case derivable at this point of problem-solving.

With sequentialized cases the developer does not only define an ordinary test case as applied to normal empirical testing, but splits up the test case into an ordered sequence. For each sub-case of the sequence an intermediate solution can be defined. The execution of the test is similar to normal empirical testing: For each sequentialized case c^* the sub-cases $c_i \in c^*$ are passed in the defined order to the knowledge system. After the entry of each sub-case c_i the (intermediate) solution of the knowledge system is compared with the expected intermediate solution of the sub-case $\mathcal{D}_{c_i}$. If for any of the sub-cases c_i the expected solution $\mathcal{D}_{c_i}$ differs from the currently inferred solution of the knowledge system, then an error is reported for this sequentialized case c^*.

Usage The sequentialized empirical testing method is applied for thoroughly testing the correctness of the implemented structural knowledge. In contrast to empirical testing, test cases additionally need to be structured by sequenced sub-cases containing a partial observation set and an intermediate solution. The construction of sequentialized cases denotes an increased effort of case acquisition, which need to be done manually. However, the application of the test method has a significant advantage compared to normal empirical testing: Thus, the evaluation of a test case is more precise since the method not only reports the fact that a case has been falsely solved, but also will point to the sub-case which is likely to cause the error.

5.6.3. Inferential Constraints

The empirical testing method presented in the previous section is very simple and can be easily applied, but has one main drawback: For an effective application of the method the developer has to define a collection of appropriate test cases for each diagnosis he wants to be included in the knowledge system. This task is often done manually and denote a complex and time-consuming task. Inferential constraints simplify the empirical testing method by the following way: The developer does not define full test cases, but only collects necessary findings and attaches inferential constraints to this collection. These constraints contain a set of diagnoses, that must appear in conjunction with the set of findings or/and a set of diagnoses, that must not be inferred if the given finding set is observed. In this way, a *unit case* is defined which is usually smaller than a real case and is evaluated in a slightly different way.

Mechanics For the *Inferential Constraints* method we utilize unit cases which are defined as follows.

Definition 5.6.7 (Unit Case) A *unit case* uc is defined as a tuple

$$uc = (\mathcal{F}_c, \mathcal{D}_{req}, \mathcal{D}_{excl})\,,$$

where $\mathcal{F}_c \subseteq \Omega_{\mathcal{F}}$ is a set of observed findings in the case. The set $\mathcal{D}_{req} \subseteq \Omega_{\mathcal{D}}$ consists of diagnoses, which are required to be inferred for the given problem description $\mathcal{F}_c$, and the set $\mathcal{D}_{excl} \subseteq \Omega_{\mathcal{D}}$ denotes a set of diagnoses, which must not inferred for the given problem description $\mathcal{F}_c$.

For each new diagnosis D the developer defines a set of unit cases with $D \in \mathcal{D}_{req}$ and $D \in \mathcal{D}_{excl}$. The method is executed by running the unit cases with the knowledge system. For each unit case it is checked, if one of its inferential constraints has been violated, i.e., if a required diagnosis has not been inferred, or if an excluded diagnosis has been derived by the knowledge system. Any violation of an inferential constraint is reported as an error.

Usage The *Inferential Constraints* method is applied for testing the correctness of structural knowledge, since it directly checks the derivation of diagnoses for given observations. The method requires unit cases as test knowledge.

5.6.4. Torture Tests

Another interesting way to test structural knowledge is implemented by *torture tests*. With this method the input of the knowledge system is gradually decreased or the quality of the available knowledge is reduced. It is measured how the quality of the knowledge system output, i.e., the inferred diagnoses, is affected by the decrements. The change of the knowledge system behavior is formally given by its robustness, which can be defined according to the IEEE Standard Glossary [59]:

Definition 5.6.8 (Robustness) The degree to which a system or component can function correctly in presence of invalid inputs or stressful environmental conditions.

The robustness of a knowledge system can be systematically investigated by degradation studies, which were introduced by Groot et al. [49, 48].
In a degradation study the quality of the knowledge system input is systematically and gradually decreased, and it is measured how the inferred output of the knowledge system decreases as a result.
Groot et al. consider three kinds of decrements with respect to the knowledge system input:

- *Decrease input data quality*: For each test case $c = (\mathcal{F}_c, \mathcal{D}_c, \mathcal{I}_c)$ the size of observed findings $\mathcal{F}_c$ is decreased, i.e., a number of findings contained in the problem description is modified or removed. With this study the system can be tested with respect to incomplete input data.
- *Decrease completeness of knowledge*: The test cases are not modified, but structural knowledge is systematically removed from the knowledge base. This approach requires some but little knowledge about the underlying knowledge representation of the structural knowledge. This study investigates the behavior of the knowledge system with respect to incomplete knowledge.
- *Decrease quality of knowledge*: This approach does not remove knowledge from the structural knowledge container, but systematically modifies existing knowledge. It is easy to see that this approach requires knowledge about the underlying knowledge representation and cannot be classified as a black-box test anymore. With this study the system can be tested with respect to the valuation quality of the developer, e.g., the behavior of the knowledge system that was developed by a biased expert.

For each study the accuracy is measured for each decrement of the input quality, the completeness of the knowledge, or the quality of the knowledge. Within this section we only discuss the first approach, i.e., decrease input quality, since the remaining approaches rely on the representation of the structural knowledge container. We discuss these approaches together with the appropriate knowledge representation.

Mechanics Originally, torture tests are not intended to be used as automated tests but are manually analyzed using degradation studies that display the mutation of the robustness by precision/recall diagrams. For the agile development process the mechanics of torture tests can be easily automated by introducing degradation thresholds for each study. However, in order to find and adjust appropriate thresholds the results of the degradation studies usually need to be inspected manually before starting the automated torture tests.

For an automated analysis of the *Decrease Input Data Quality* study we simply need to define the following thresholds:

$\mathcal{T}_F$:	The minimum average of the computed F-measure for the degraded test cases.
$\mathcal{T}_N$:	The minimum number of findings contained in a degraded test case.

As discussed earlier, we use the F-measure in the context of our process model. The implementation of a degradation study is quite simple: For each study we take the test cases and incrementally decrease the number of observed findings for each case until the number of remaining findings is greater than or equal to $\mathcal{T}_N$. The removed findings are randomly selected. Each degraded case is used for a run with the knowledge system, and we compute the F-measure. The F-measures of all decreased cases are averaged to a total average F-measure of the processed study. Of course, the study is executed many times in order to receive a sound result of the study. If the average of the total F-measures is less than $\mathcal{T}_F$, then an error is reported.

Usage Torture tests are applied for testing the robustness of the implemented structural knowledge. Robustness can be classified as a kind of correctness in presence of incomplete input data or incorrect knowledge. The method requires test cases as test knowledge and reasonable threshold values for the minimum number of required observations per case and for the minimum F-measure for a solved case.

5.6.5. Summary

This section introduced various methods for testing structural knowledge. In particular, we considered black-box testing methods that are independent from the underlying knowledge representation. In general, black-box testing methods are suitable for simply testing the correctness and robustness of the implemented knowledge, and require (often costly) test knowledge, e.g., test cases. Specialized (white-box) tests, introduced in the preceding chapters, are often used to investigate the understandability of the available knowledge, and for detecting hidden anomalies. For white-box test often no or cheap test knowledge is required, i.e., threshold values.

5.7. Conclusion

In this chapter, we presented different approaches for implementing structural knowledge. All these approaches were inspired by mental models of domain experts, and have been proven to be suitable for real world applications in the past. Thus, we introduced abstraction knowledge, case-based knowledge, rule-based knowledge and set-covering models for filling the structural knowledge container. For all approaches presented in this chapter we also explained their implementation in the context of the agile process model; consequently, we introduced appropriate test measures, learning algorithms, and restructuring methods.

6. The Strategic Knowledge Container

The development of real world knowledge systems often embodies the definition of a large set of diagnoses. For each diagnosis or group of diagnoses a number of tests (questions) are contained in the knowledge base for establishing or excluding them. It is obvious that prompting all available tests for all diagnoses denotes a complex and time-consuming task, which is commonly not accepted by end-users. Furthermore, sometimes tests are risky and should be only made under specific circumstances, e.g., if the utility of deriving a specific diagnosis is higher than the involved risks or costs.

Therefore, large systems mostly contain strategic knowledge to guide the questionary, and to avoid unnecessary questions for the particular cases. Strategic knowledge is used to focus on relevant questions, and to capture all necessary information in a consultation.

6.1. Knowledge Representation

The knowledge included in the strategic knowledge container determines the dialog behavior of the knowledge system. In general, we distinguish between a local and a global dialog strategy. Whereas the *global dialog strategy* mainly guides the dialog by selecting question sets appropriate for the current case, the *local dialog strategy* refines the selected question sets by indicating additional follow-up questions. The indication of questions and question sets, respectively, is performed by indication rules.

Definition 6.1.1 (Indication Rule) *Indication rules* are denoted by

$$r = cond(r) \rightarrow indicate(Q_1, \ldots, Q_n)\,,$$
$$r' = cond(r') \rightarrow \neg indicate(Q_1, \ldots, Q_n)\,,$$

where $cond(r)$ is the rule condition of rule r as defined in Figure 5.1 (p. 65) and the list $(Q_1, \ldots, Q_n)$ either denotes an ordered collection of questions (local indication rule) or a sequence of question sets (global indication rule), i.e., $Q_i \in \Omega_Q \vee Q_i \in \Omega_{QS}$.

The rule action *indicate* adds the specified questions (sets) to a dialog agenda in order to be asked to the user. If an indication rule is withdrawn, then it is removed from the agenda. The rule action ¬*indicate* has the contrary meaning: The specified questions (sets) are indicated not to be asked to the user, i.e., they are blocked at the dialog agenda. Rules with action ¬*indicate* are also called *contra-indication rules*.

Indication rules can be sub-classified into normal indication rules, clarification rules, and refinement rules. Whereas normal indication rules activate additional question sets according to the observation of specified findings, clarification and refinement rules depend on

the current state of a diagnosis. *Clarification rules* indicate additional tests for a suggested diagnosis in order to decrease or increase the evidence of the diagnosis. Question sets indicated by *refinement rules* are used to approve the state of an already established diagnosis and to refine the established diagnoses with respect to underlying sub-diagnoses. Therefore, the conditions of clarification and refinement rules consists of conditions among the states of diagnoses.
Furthermore, the dialog interface can be optimized by suppressing answer alternatives of specified questions, e.g., answers are hidden if they are not reasonable in the context of the current case. Answer alternatives are hidden using suppress rules.

Definition 6.1.2 (Suppress Rule) A *suppress rule* is denoted by

$$r = cond(r) \rightarrow suppress(Q, v)\,,$$

where $cond(r)$ is the rule condition of rule r as defined in Figure 5.1 (p. 65), $Q \in \Omega_Q$ is a (choice-)question, and $v \in dom(Q)$ is a value of Q. If the suppress rule is activated, then the dialog is notified not to display the specified question value v.

Indication rules implement local as well as global strategic knowledge, whereas suppress rules are used for local strategic knowledge.
If more than one question set is indicated by rules and therefore included in the dialog agenda, then it is not clear which question set should be primarily presented to the user. The simple solution would select the question sets ordered by their indication order, i.e., the first indicated question set is presented at first. A more sophisticated solution for competing question sets is the definition of priorities defined by a cost function.

Definition 6.1.3 (Cost Function) The *cost function* for question sets is defined by

$$cost : \Omega_{QS} \rightarrow \Omega_{cost}\,,$$

where Ω_{QS} is the universe of question sets and Ω_{cost} is the universe representing costs. Using the cost function the risks, the required duration time, and the monetary costs of a question set can be represented.

For simplicity the universe Ω_{cost} can be mapped to $\mathbb{R}$. However, in practical applications a more abstract universe with symbolic categories is simpler to acquire, e.g., $\Omega_{cost} = \{c_1, \ldots, c_{10}\}$. In the following section, we present approaches for the local and the global dialog strategy using indication rules and cost functions.

6.2. Knowledge Acquisition

Commonly, the usage of strategic knowledge follows a simple top-level pattern: When starting a new dialog an initial questionary (*init-question pattern*) consisting of one or more question sets is presented to the user. Based on this initial questionary further question sets are indicated to be asked in a specified order. For the indication of further questions and question sets we identify different approaches.

6.2.1. Local Strategic Knowledge

Local strategic knowledge refines the local behavior of the dialog and is implemented using local (contra-)indication rules and suppress rules. In this manner, a dialog tree is constructed that performs tests appropriate for the current case and allows for an economic acquisition of the case data. Questions that are activated by indication rules are appended to the *local indication agenda*, and questions are added to the *local contra-indication agenda*, if they were activated by contra-indication rules. Accordingly, questions are removed from the agendas if the corresponding rules are drawn back.

In the user dialog questions included in the local indication agenda are immediately presented, if not also included in the contra-indication agenda. Before the rule is actually presented to the user suppress rules are checked in order to possibly hide answer alternatives, see suppress rules.

6.2.2. Global Strategic Knowledge

The task of the global strategic knowledge is the guidance of the overall dialog behavior, i.e., the indication of question sets appropriate for the current case. The implementation is carried out using global (contra-)indication rules and cost functions. For the acquisition of global strategic knowledge we classify different approaches. Whereas the standardized indication can be used only using strategic knowledge, the other approaches require (arbitrary) structural knowledge for deriving diagnoses.

Standardized Indication The standardized indication simply uses indication rules and contra-indication rules for defining a dialog behavior that is predefined by the developer. Indication rules are applied implementing a dialog tree covering all aspects of a possible case in the considered domain. Typically, no cost function for question sets is used. No additional structural knowledge is used, since question sets are presented due to indication rules that depend on the currently gathered observations. The inference of the standardized indication is quite simple: Indicated question sets are appended to a dialog agenda and are subsequently presented by the dialog. However, already indicated question sets are not presented if marked by a contra-indication rule.

Establish-Refine Strategy In contrast to the static definition of the dialog behavior using the standardized indication the establish-refine approach dynamically builds a dialog with questions sets (tests) appropriate for the current case. The main idea of the approach is the establishment of coarse diagnoses, i.e., problem areas, using the init question pattern. Then, further question sets are indicated in order to refine the established problem areas, i.e., deriving underlying diagnoses with a more detailed meaning. The establish-refine strategy is implemented with refinement rules. Due to its dynamic indication the order of indicated question sets is not predictable. The application of a cost function can help to define a priority of indicated question sets. With a defined cost function the indication agenda is sorted (stable sort) in ascending order according to the defined costs.

The establish-refine strategy requires a well-elaborated diagnosis hierarchy and additional structural knowledge for deriving diagnoses, i.e., problem areas.

Hypothesize-and-Test Strategy The hypothesize-test strategy also uses the init question pattern, to derive diagnoses that are used to indicate further question sets (tests). Compared to the establish-refine strategy question sets are not indicated because of established problem areas, but question sets are indicated in order to confirm or disconfirm suggested diagnoses. Consequently, the hypothesize-test strategy is implemented using clarification rules. For the implementation we can distinguish two approaches: The first approach selects the most probable diagnosis for which indication rules are available. Only question sets relevant for this diagnosis are indicated. The second approach selects all suggested diagnoses and their indication rules, respectively. Due to the multiple selection of different question sets it is reasonable to use a cost function defined for question sets. With a cost function the indicated question sets are sorted (stable sort) in ascending order according to their defined costs. In addition to the defined cost function the indication of question sets can be ordered by their utilities. Then, question sets relevant for more suggested diagnoses are indicated before question sets only relevant for fewer suggested diagnoses.

The hypothesize-test strategy requires additional structural knowledge for inferring suggested diagnoses in the current case and a defined cost function.

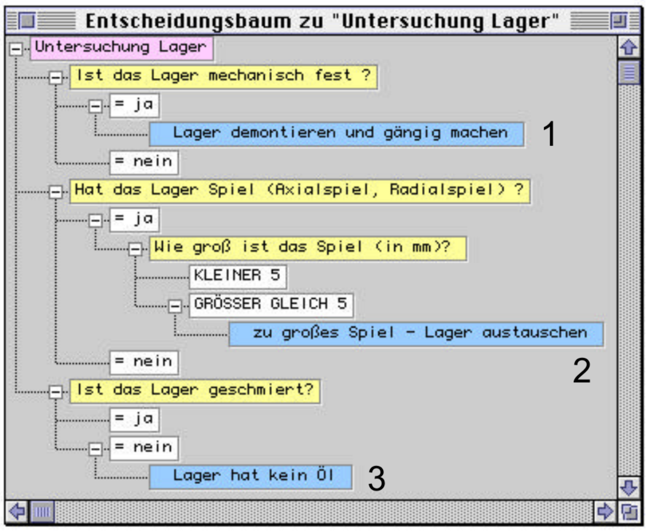
Utility-Based Strategy The utility-based strategy selects questions and question sets, respectively, according to their relevance for currently suggested diagnoses. The init question pattern is used to prompt an initial set of questions. Based on these questions diagnoses are becoming suggested or probable. Then, additional questions or question sets with the highest utility for the suggested diagnoses are selected. The utility of a question is defined by its usefulness in order to establish or exclude the given diagnosis. The utility of questions or question sets can be derived by background knowledge containing the sensitivity/specificity for each diagnosis/question pair.

Alternatively, such knowledge can be inferred approximately by structural knowledge. For example, the KMS.HT [89, 107] system applies the utility-based strategy. The knowledge system uses set-covering knowledge and presents the question that is predicted most frequently by the suggested diagnosis and that is currently not answered by the user. After the question has been answered hypotheses are regenerated and the system again selects the question that is predicted most frequently by the currently suggested diagnosis.

This strategy can be simply implemented and no additional strategic knowledge is required. However, since the dynamic indication yields an incoherent dialog behavior frequently switching between different questions sets, it is seldom used in large applications.

Case-Based Strategy The case-based strategy requires a case base which consists of highly structured cases, i.e., solved cases with a stored indication sequence of the processed question sets. For any new indication decision the strategy selects a case of the case base that is most similar to the currently entered case. Then, the order of the indicated question sets is determined by the stored indication order of the retrieved case.

Mixing Strategy The strategies discussed above can be mixed, e.g., a standardized indication is extended by a utility-based indication. However, the use of different strategies can decrease the clarity and structure of the strategic knowledge container. A more structured approach of a mixed strategy is the heuristic decision tree.
Heuristic Decision Tree Using a mixed approach strategic knowledge is jointly acquired together with structural knowledge. We propose the use of (heuristic) decision trees that are easy to understand and commonly applied in technical domains. This type of strategic knowledge is only focussed on retrieving a meaningful solution for a problem and does not consider a standardized documentation. The development of heuristic decision trees is described by the HEURISTIC DECISION TREE pattern [102]. Heuristic decision trees weaken the criteria of normal decision trees by handling noisy or missing data, and by providing not only established diagnoses for a given problem but also suggested solutions. In summary, the pattern describes the use of an entry investigation (init question pattern) that is used to determine the focussed problem area. Based on the located problem area all refining diagnoses contained in the problem area are suggested. Then, question sets corresponding to specific decision trees are activated in order to exclude or to confirm the suggested diagnoses. In Figure 6.1 an example of a heuristic decision tree in a technical domain is depicted. The decision tree determines the exact diagnosis of a bearing block (”Lager”). Final diagnoses are connected as leafs of the tree (attached with numbers 1–3).



```
Entscheidungsbaum zu "Untersuchung Lager"
Untersuchung Lager
  Ist das Lager mechanisch fest ?
    = ja
      Lager demontieren und gängig machen   1
    = nein
  Hat das Lager Spiel (Axialspiel, Radialspiel) ?
    = ja
      Wie groß ist das Spiel (in mm)?
        KLEINER 5
        GRÖSSER GLEICH 5
          zu großes Spiel - Lager austauschen
    = nein                                    2
  Ist das Lager geschmiert?
    = ja
    = nein
      Lager hat kein Öl   3
```

Figure 6.1: Example of a heuristic decision tree for diagnosis faults in bearing blocks (”Lager”).

For a more detailed discussion of the HEURISTIC DECISION TREE pattern together with the description of variants we refer to Puppe et al. [106, Ch. 2].

6.3. Testing Strategic Knowledge

Testing strategic knowledge ensures that the knowledge system implements the intended dialog paths. For a thoroughly testing of strategic knowledge it would be necessary to define an exhaustive dialog graph containing all reasonable dialog paths. It is obvious that this task can evolve to be very costly and time consuming, and is therefore only tractable for small systems.

In the following, we introduce alternative methods that try to minimize the acquisition costs of test knowledge.

6.3.1. Partial-Ordered Question Sets

The *Partial-Ordered Question Sets* method evaluates the correctness of the implemented strategic knowledge: It can be applied statically for standardized knowledge and dynamically with test cases for arbitrary strategic knowledge. As implied by the name, the method requires a set of partial-ordered question sets as test knowledge. Such sets define typical dialog sequences of real cases.

Definition 6.3.1 (Partial-Ordered Question Set) Let Ω_{QS} be the universe of all question sets. A *partial-ordered question set* $POQS$ is a sequence

$$POQS = (\, S_1, \ldots, S_n \,)\,,$$

where $S_i \subseteq \Omega_{QS}$ is a set of question sets and $n \geq 1$. The $POQS$ defines the desired order of a collection of question sets. In a typical dialog the question sets contained in S_i should be appear before the question sets specified in S_{i+1}. Further, question sets contained within a S_i can appear in the dialog in an arbitrary order.

The following example may clarify the definition: For a medical application the $POQS = \big(\{\text{entry examination}\}, \{\text{exam1, exam2}\}, \{\text{lab exam}\}\big)$ is defined, which states that the question set "entry examination" should appear before the question sets "exam1" and "exam2" followed by question set "lab exam". The order of the sets "exam1" and "exam2" is denoted to be irrelevant.

Using a collection of $POQS$ the developer of the knowledge system is able to specify a set of typical (and intended) dialog paths in a convenient and compact manner.

Mechanics For the static evaluation of the test knowledge contained in $POQS$ strategic knowledge needs to be represented by the standardized indication. A transitive indication graph starting from the initial question sets is build. This is done by recursively tracing the implemented indication rules contained in questions of the initial question sets. During the recursive traversal after each new indication of a question set we test, if the order of any of the defined $POQS$ has been violated, and we report an error in the case of a violation. A warning is given for a $POQS$, if the $POQS$ cannot be completely evaluated, i.e., the order of the first part of the $POQS$ is correct, but the last question sets of the $POQS$ cannot be indicated by any strategic knowledge.

For an arbitrary representation of strategic knowledge, $POQS$ can be applied for a dynamic evaluation: Then, test cases are used to generate knowledge system runs. For each test case the dialog path is logged by means of an ordered sequence of indicated question sets. These sets are compared with the implemented $POQS$ and significant differences are analogously reported by errors and warnings as described for the static evaluation.

Usage The method evaluates the correctness of the implemented strategic knowledge using pre-defined typical dialog paths, called $POQS$. If question sets contained in a $POQS$ are indicated by the knowledge system in inverse order, then there exists evidence for an incorrect behavior of the implemented strategic knowledge, and consequently an error is reported. If the knowledge system indicate the correct order of the questions sets contained in a $POQS$ but the last question sets are not indicated, then this irregular behavior is reported as a warning. For the static evaluation only a set of $POQS$ is required. The dynamic evaluation of the method additionally requires a set of test cases.

6.3.2. Diagnosis-Related Question Sets

Typically, the indication of question sets is strongly related with the diagnoses to be derived. Then, specialized tests (question sets) are presented if a given diagnosis is suggested or established. In such cases, the *Diagnosis-Related Question Sets* (DRQS) method can be applied. We can define for each diagnosis a list of necessary question sets (partially ordered) that have to be indicated and answered by the user if the diagnosis has been established.

Definition 6.3.2 (Diagnosis-Related Question Set) Let Ω_{QS} be the universe of all question sets and $\Omega_{\mathcal{D}}$ is the universe of all diagnoses. A *diagnosis-related question set* $DRQS$ is a set

$$DRQS(C) = \{\, P_1 \ldots, P_n \,; \mathit{evalFlag} \,\}\,,$$

where $C = (D_1 \wedge D_n) \wedge (D_1 \vee \mathcal{D}_m)$ is a condition consisting of diagnoses with $D_i \in \Omega_{\mathcal{D}}$, and $P_i \subseteq \Omega_{QS}$ are partial-ordered question sets ($POQS$); a $DRQS(C)$ defines a set of alternative $POQS$. The condition C is true, if the constrained diagnoses are indicated in a given case. Then, at least one P_i contained in $\{P_1, \ldots, P_n\}$ need to be fulfilled, i.e., the question sets in P_i need to be indicated in the defined order. For the evaluation of the $POQS$ we consider a given $\mathit{evalFlag} \in \{\mathit{after}, \mathit{before}, \mathit{free}\}$: For the values $\mathit{after}/\mathit{before}$ the given question sets need to be indicated after/before the diagnoses in C have been derived; no order of indication of the question sets and diagnoses need to be considered for the value free.

It is easy to see that the $DRQS$ method is an extension of the POQS method, and now additionally considers a set of derived diagnoses.

Mechanics For the evaluation of the test method we apply a set of test cases to the knowledge system. Similarly to the Partial-Ordered Question Set method the dialog path is logged for each test case. The logged dialog path is represented by a sequence of indicated and answered question sets. For each test case the appropriate $DRQS$ is selected, i.e., $DRQS$ containing the diagnoses derived by the test case. The indication log is compared with the corresponding $POQS$ defined in the $DRQS$. An error is reported, if the stored dialog sequence cannot be mapped to at least one $POQS$ contained in $DRQS$. For values of $evalFlag$ other than $free$ we additionally check, if the question sets were indicated before or after the diagnoses has been derived.

With the evaluation of $DRQS$ the following problem can arise: The indication order of the specified question sets can be violated, if a subsequent question set is accidently indicated because of another derived diagnosis. In this case a simple comparison of the indication log is not sufficient, and we additionally have to parse the activated indication rules of the presented question sets.

Usage The method is used to test the correctness of the implemented strategic knowledge together with existing structural knowledge. Therefore, this method can only be applied, if structural knowledge is available for the diagnoses contained in the implemented $DRQS$. We additionally expect test cases to be present as test knowledge.

6.3.3. Test Case Duration

The *Test Case Duration* method evaluates an important aspect, which should be considered besides the correctness of the strategic knowledge. For documentation and consultation systems the duration time of a dialog is a very significant measure for the quality and acceptance of the system. Thus, users of the system are usually short in time and do not want to enter useless and time-consuming information. For this reason, the practical success of a knowledge system project also depends on the duration of the implemented dialog. Benchmarks are generated by preliminary studies that capture the actual state of the domain before bringing the system into routine usage. Mainly, the benchmarks contain information about the duration and quality of examinations (e.g., in a medical domain).

Mechanics The typical duration of a dialog can be measured by manually quantifying the time during a user is running a case. For a sample of cases a statistic can be generated yielding the mean duration of a case. Obviously, a short duration increases the practicability of the system. The main drawback of this method is its incapability of automation, which declines it from the continuous application during the development of the knowledge system.

An approximate method for this test is *question counting*: Test cases are applied and passed to the system. For each finished case the answered questions and indicated question sets are counted. The mean number of answered questions and indicated question sets for all test cases can give an approximation of the typical duration time, when assuming an estimated and constant duration for each answered question (e.g., 5 seconds). This simple measure can be improved by an adapted quality function not only counting the number of questions and question sets, respectively. The proposed measure may include different duration weights for the different types of questions, e.g., choice question have typically a lower duration weight than numerical questions. Furthermore, a choice question can obtain an adapted weight depending on the number of available answer alternatives (additionally regarding the length of the presented texts of the question and answers). The hierarchical structure of the questions and question sets, respectively, also can affect the dialog duration, e.g., follow-up questions not presented to the user until becoming necessary may reduce the dialog complexity and consequently the duration time.

The test reports an error, if the calculated duration exceeds a value that is greater than 105% of the provided benchmark. If the calculated time is greater than 100% and less than 105%, then the method reports a warning. Of course, the threshold can be adapted with respect to the specific requirements of the knowledge system project. Sometimes an average benchmark for the duration time is not sufficient for a reasonable test of the dialog efficiency, e.g., the test suite contains very simple but also very complex and time consuming cases. Then, the method can be improved by providing individual duration benchmarks for each test case.

Usage The method checks the practicability of the knowledge system by approximating its efficiency during system usage. The test can be automated and needs a collection of test cases as test knowledge. Additionally, the test needs a benchmark value, which assumes the desired duration time of a typical dialog. This benchmark value is often acquired by preliminary studies preceding the knowledge system development.

6.4. Restructuring Strategic Knowledge

This section briefly discusses the application of restructuring methods to the strategic knowledge container. Mainly, restructurings of the strategic knowledge container are methods corresponding with the ontological container. However, we will also present methods for restructuring elements of the strategic knowledge container.

6.4.1. Propagation of Ontological Restructuring Methods

Changes caused by restructuring methods of the ontological knowledge container also need to be propagated to the strategic knowledge container. In the following, we summarize the changes of ontological changes on indication rules.

TransformMCIntoYN If the multiple-choice question is contained in the rule condition of an indication rule, then it is replaced by the yes/no questions corresponding to the original values of the multiple-choice question. If the multiple-choice question was originally indicated by the indication rule, i.e., the restructured question is contained in the rule action, then it is replaced by the generated yes/no questions.

TransformYNIntoMC If one of the yes/no questions is contained in a rule condition, then it is replaced by the multiple-choice question with the corresponding value. For an indication rule with one of the yes/no questions in the rule action, the multiple-choice question is now indicated instead of the original question.

TRANSFORMNUMINTOOC The restructuring method is only applicable, if the numerical value range can be discretized into distinct sub-partitions. Then, the condition of an indication rule containing the original numerical question is modified, so that the one-choice question with the discretized value is contained. If the numerical question was indicated by a rule action, then the new one-choice question is indicated, instead.

SHRINKVALUERANGE If the restructured choice-question is contained in a rule condition, then the conditioned value is modified with respect to the transformation matrix. No change is required for indication rules containing the restructured question in the rule action.

INTRODUCEABSTRACTION If the condition of an indication rule contains all abstracted findings, then replace these findings by the newly created abstraction. The rule action of indication rules is never affected by this method.

MOVEQUESTIONVALUE For this restructuring we only consider the condition of indication rules. If the condition of an indication rule contains the moved value assigned to the original question, then the condition is modified so that the new targeted question is now assigned to the moved value. However, the method can cause conflicts, e.g., by generating cyclic indication graphs. In this case, the user has to decide manually, if a rule has to be removed from the strategic knowledge container.

REMOVEQUESTION If the removed question is contained in a rule condition of an indication rule, then we have to distinguish two cases:

1 Only the removed question is contained in the condition. Then, we remove the rule from the strategic knowledge container.

2 The removed question is contained in the rule condition among other questions. Then, the user has to decide about

 2.1 only removing the sub-condition containing the removed question, or

 2.2 removing the entire rule.

If default values are specified for case 2, then this restructuring method can be applied automatically; otherwise an interaction with the user is necessary. For indication rules containing the removed question in the rule action, we have to similarly decide as described above: The rule is removed, if only the question is contained in the rule action. For a rule action containing more than the removed question, we have to consider the decisions described in case 2.

REMOVEDIAGNOSIS If a diagnosis was removed from the ontological knowledge container, then we analogously have to proceed as described for removing a question. However, we only have to consider the event, that the diagnosis is contained in a rule condition, since diagnoses cannot be indicated, in general.

EXTRACTQUESTIONSET If a question set was extracted from another question set, then we have to consider indication rules containing the original question set in their rule action. All rules indicating the original question will also indicate the extracted question set. It is worth noticing, that the ordering of the indicated question sets is determined by defaults.

COMPOSEQUESTIONSETS Similarly to the EXTRACTQUESTIONSET method, only the rule actions of indications rules need to be considered, since question sets cannot be included in rule conditions. Then, all rules containing one of the combined question sets in their rule action are modified, so that they now indicate the composed question set.

6.4.2. Restructuring Methods of Strategic Knowledge

We briefly introduce restructuring methods that consider the change of implemented strategic knowledge.

COMBINEINDICATION The developer of the knowledge system usually defines a question set (test) asking specialized findings, which can refine the state of a given diagnosis. Sometimes additional tests for specific diagnoses become necessary, and then often a question set containing new findings is implemented. The new question set should be indicated, whenever the old question sets are indicated. The restructuring COMBINEINDICATION provides a comfortable method for combining indications of several question sets.

The developer specifies a set $\mathcal{Q}$ of question sets, for which their indication should be combined. Then, all strategic rules indicating at least one question set $QS \in \mathcal{Q}$ are modified, so that they also indicate the remaining question sets $QS' \in \mathcal{Q} \setminus \{QS\}$.

MOVEINDICATION Due to the evolutionary design of the knowledge base a question set may have become redundant, e.g., all important questions contained in the question set have been moved by preceding restructuring methods. Then, with the MOVEINDICATION method all rules that are originally indicating the redundant question set are modified, so that these rules are now indicating another question set.

Restructuring of Strategies If the implementation of the strategic knowledge container does not follow an organized approach, then the restructuring of the entire strategic knowledge may become necessary. Such a restructuring can imply the definition of cost knowledge or the introduction of a strategic approach, e.g., a hypothesize-and-test strategy. Also a mixed approach including the establish-refine and the hypothesize-and-test strategy can be reasonable. The modification of indication rules is a complex task and it is difficult to provide automated methods for this. Nevertheless, the developer can be supported by appropriate tools with specialized editors for indication rules. Furthermore, a sufficient test suite is also necessary. Although, this type of restructuring was not considered in the context of this thesis it is a promising direction for future work.

6.5. Summary

In this chapter, we have introduced the strategic knowledge container. Strategic knowledge is an essential container for large knowledge systems, when not all available questions are required for inferring an appropriate solution or acquiring all questions is too complex and costly for the user. For the indication of questions and question sets, respectively, we distinguish local and global strategic knowledge, and we have presented different approaches for their implementation. According to the agile process model we have described testing methods and restructurings of the strategic knowledge container.

7. The Support Knowledge Container

Often the development of a diagnostic knowledge system also includes the application of informal knowledge for specified entities of the knowledge base, i.e., diagnoses or findings. The term *informal support knowledge* comprises unstructured or semi-structured text and multimedia content like text book entries, images, movies or sounds. It is used to informally describe diagnoses or findings in more detail. Then, the ontological entities are connected with support knowledge by a collection of *links*. In the following, we describe situations in which support knowledge can be used, and we present methods for testing structural knowledge.

7.1. Roles of Support Knowledge

Additional, informal knowledge can support the user of the knowledge system and is applied for the following reasons:

- support the user during data acquisition
- support the user with the retrieved problem solution

We now will discuss these *roles* of support knowledge in more detail.

7.1.1. Support during Data Acquisition

The appearance of support knowledge for data acquisition should be optional and only be present on demand, since it can increase the dialog size significantly, which is not desirable with respect to the dialog efficiency. When supporting the user during data acquisition we distinguish several support tasks.

Support for Performing an Examination In technical and medical domains often specialized tasks need to be performed, before a finding can be acquired. For example, the ECHODOC system supports the user during the transesophageal echocardiography (TEE) examination. Performing this examination is difficult for unexperienced users, especially the correct adjustment of the examination device, which is required for a qualitative acquisition of findings. Therefore, the system embodies a well-elaborated support container, which provides multimedia based support of device adjustment for each finding to gather. Support is given by additional texts describing the correct examination step including a correct setting of the device. Besides the presented informal content even more structured knowledge can be offered: The integration of *wizards* can interactively clarify the correct execution of an examination. Then, a set of control questions connected with structural knowledge can determine the correct setting of the device.

Identification of Findings Providing the actual value of a question can be difficult for the user of the system. E.g., the user may not feel certain about the correct actions about determining a finding. Support knowledge can help the user by additional information (e.g., images and movies) depicting exemplary characteristics of the explained finding. For example, in the ECHODOC system the user is supported by images that are used for a visual interpretation of findings.

Description of Examinations Besides aiding the executing of an examination, support knowledge can be also used for describing the purpose and context of the performed examinations. This kind of support knowledge is mainly represented by texts, and is often useful when an alternative list of optional examinations are offered, and the user has to select manually the suitable examination.

7.1.2. Support the Retrieved Solution

When the system has derived a solution for a given problem support knowledge can be used to help the user with interpreting the derived solution.

Describing an Inferred Solution Support knowledge can be used to describe a represented diagnosis in more detail. For example, text-book knowledge about the diagnosis can be offered by hyper-media texts that explain the pathology or therapy possibilities in more detail. Additionally, the description of the inferred solution can contain an estimation about the correlated costs and affiliated risks. However, this type of support knowledge can be extended in order to substitute a (possibly weak) structural container: In some cases a well-elaborated structural knowledge container is not reasonable or not possible. In these cases knowledge base design probably will focus on a weak structural content only inferring coarse diagnoses, that are linked with support knowledge. The user of the system is able to browse through a network of hyper-linked texts including images and movies, in order to refine the previously inferred coarse diagnosis. On the one hand, such a system does not require the costly formalization of structural knowledge and thus simplify the development process. On the other hand, in such a system the user is expected to be sufficiently experienced to find the correct and final solution in a manual way. As mentioned in Section 4.2 (p. 47) therapies are treated as diagnoses, and therefore support knowledge describing therapies in more detail is defined analogously.

Support Therapy Execution In contrast to diagnoses, therapies can contain support knowledge for performing the derived therapy. Then, the support knowledge, e.g., contains advices and guidelines about the dosage and the application duration of the inferred therapies. This kind of support knowledge can be enhanced by interactive *wizards*, that, e.g., compute the correct dosage for the specified patient.

7.2. Acquisition of Support Knowledge

The acquisition of support knowledge is quite simple: For each diagnosis and question that should be extended by support knowledge a set of links is specified. Each link represents a kind of content supporting the corresponding diagnosis or question.

Before using the support knowledge container it is important to decide about the technical aspects of storing the support knowledge. Additional content like hypertext, images or movies can be stored in a (structured) file system or a data base. Furthermore, for the construction of a large support knowledge container it is reasonable to consider the use of a content management system, which allows for a structured approach of managing various kinds of support knowledge.

7.3. Testing Support Knowledge

Since the support knowledge container mostly consists of unstructured data testing can be hardly automated. However, we present methods for testing the links between the ontological entities and the informal support knowledge.

7.3.1. Plain Link Testing

A very simple but useful test method is *Plain Link Testing*. For all available links it is tested, if the linked content is accessible.

Mechanics The method simply checks in a straightforward way, if the specified links are available, i.e., if the linked content can be retrieved. If not all links lead to content, then an error is reported. The test is simple but useful, when non-local content is linked to the system, that is not administered by the developer of the knowledge system. Web-content like web-pages or data bases available by a web-interface are examples for non-local support knowledge.

Usage The method checks the correctness of the implemented support knowledge, since unavailable content yield to an unpredicted behavior of the knowledge system. For this method no additional test knowledge is required. However, for checking non-local content an appropriate infrastructure is needed, e.g., an internet connection.

7.3.2. Static Link Testing

The *Static Link Testing* method provides an overview of the available links to the particular support knowledge entries.

Mechanics The method checks for each (final) diagnosis and for each question, if links to support knowledge are available and counts the number of links for each entity for statical use. If for one entity no support knowledge is available, then an error is reported. A warning is given, if for an entity a disproportionate number of links has been attached, e.g., significantly more or less than the average.

Usage The method checks for anomalies contained in support knowledge, since missing links or entities with a disproportionate number of links represent irregularities in the knowledge base. For the method no additional test knowledge is required.

7.3.3. Ambivalent Link Testing

Often, the same entry of support knowledge (e.g., text book chapter) is linked to more than one ontological entity. Then, the *Ambivalent Link Testing* method can be applied in order to test, if the linkage has an ambivalent meaning.

Mechanics For each support knowledge entry linked to more than one entity we check the semantical relationship between these entities. It is easy to see, that for this task ontological knowledge is needed as test knowledge, that enables for a semantical check between ontological objects. If the support knowledge entry is linked to entities which have no semantical relationship, then this is reported as a warning.

Usage The method evaluates the knowledge for anomalies, which can be present by ambivalent links. The test method requires ontological knowledge, which contains semantical relationships between the implemented ontological entities.

7.4. Restructuring Support Knowledge

Due to its informal characteristics restructuring of support knowledge can be hardly supported by semi-automatic methods. However, for the ontological restructuring methods REMOVEQUESTION and REMOVEDIAGNOSIS we need to consider the deletion of the attached support knowledge.

7.5. Summary

In this chapter, we have introduced the support knowledge container, which consists of (mostly) informal knowledge like texts, images or movies but also can contain structured knowledge used by interactive wizards. Support knowledge can be added to diagnoses and findings for aiding the user during data acquisition, and for helping the user with a retrieved problem solution. Although support knowledge is mostly unstructured and automated testing is hardly possible, we presented (simple) methods for evaluating the support knowledge container.

Part III.

Practical Aspects

8. Implementation of the Agile Process Model with d3web

In this chapter, we present the system d3web– a shell-kit for developing and using diagnostic knowledge systems. We sketch the historical evolution of the D3 system, the predecessor of d3web, and we give an overview of the d3web system and its components. Then, the components of the d3web system will be introduced in more detail. A brief comparison of the knowledge acquisition environments of the predecessor D3 and d3web.KnowME concludes this chapter.

8.1. Overview

The d3web system originates from the D3 system, which has been applied since 1989 in many medical, biological, and technical domains; e.g., Puppe [100] gives an overview of applications until 1998. D3 itself was the successor of the systems MED1 [105], MED2 [98], and CLASSIKA [103], and provides a monolithic program for the acquisition and the use of diagnostic knowledge systems. D3 was initially running on Apple systems©, and later was ported to Microsoft Windows© systems. Since 1996 the system D3 is commercially distributed and maintained by the IISY AG, which so far has established many successful systems especially in the technical domain. Actually, the commercial system was renamed to SOLVATIO, including many extensions like multi-user support and customizable web interfaces. For a detailed historical overview till 1996 we refer to [104, App. B].
The presented d3web system is a complete Java-based re-implementation of the D3 system started in 2001. Many colleges and students at the Department of Computer Science VI at the Würzburg University were involved in this implementation project. The current distribution of the d3web system can be downloaded from

`http://www.d3web.de`.

Actually, d3web is a system family consisting of a set of separate components, all based on the root component d3web.Kernel. Figure 8.1 shows the main components of the d3web system family.

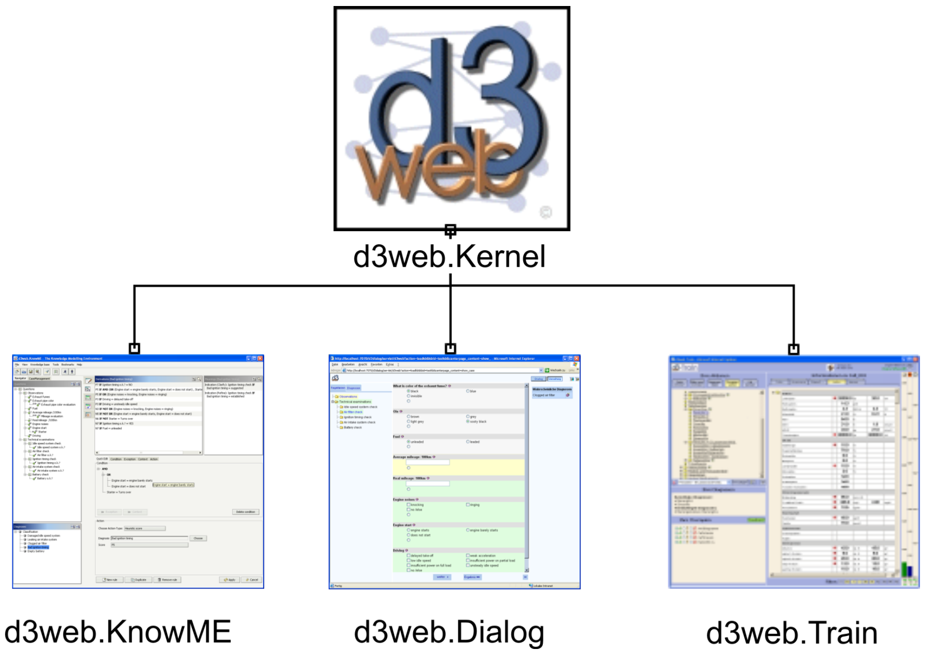


Figure 8.1: The main components of the d3web system family, including the kernel, a knowledge modeling environment, a web-based dialog, and a web-based tutoring system.

The component d3web.Kernel provides the basic functionality of a diagnostic knowledge system, e.g.,

- the representation of knowledge bases and cases,
- the persistence management of knowledge bases and cases,
- a set of problem-solvers handling
  - structural knowledge (abstraction, case-based knowledge, categorical knowledge, score-based knowledge, symbolic set-covering knowledge)
  - strategic knowledge (standardized indication)
- a multi-user architecture, which facilitates simultaneous problem solving sessions

The component d3web.KnowME represents the knowledge modeling environment for the development, maintenance, and validation of diagnostic knowledge systems. We describe the components of this component in the following sections in more detail. The component d3web.Dialog represents the user interface of an implemented knowledge system, providing a web-based server for running cases and retrieving problem solutions. The component d3web.Train [57] allows for the development and the application of knowledge-based tutoring systems. This component is not considered in the presented work. As an important aspect of this framework, the previous monolith was split up into three independent applications only based on the same kernel, i.e., each component linked with the kernel represents a stand-alone program.

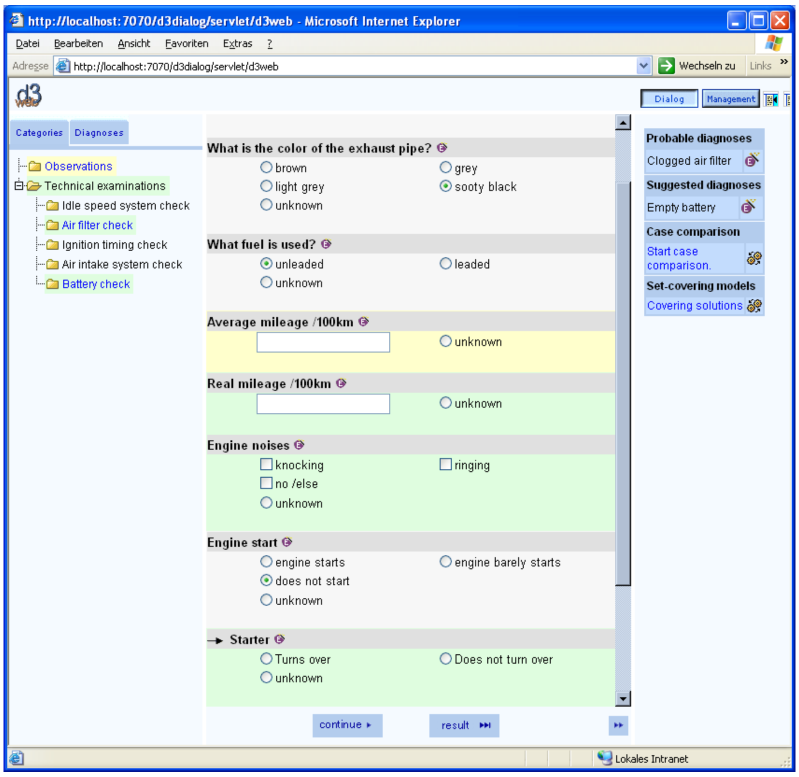
In this work, we want to introduce the most important aspects of the application d3web.Dialog for using the knowledge systems and the knowledge modeling application d3web.KnowME in more detail.

8.2. Using Knowledge Systems: d3web.Dialog

The d3web.Dialog is a server application written in Java using the popular servlet container APACHE TOMCAT [143]. The server application requires a Java JRE[1] 1.4 or better. We have deployed the server on Microsoft Windows© and Linux systems, but in principle the server can be installed on any operating system, for which the required JRE is available. For the user-client a web-browser is required. Currently, the implementation of the client is mainly optimized for Microsoft Internet Explorer 5.5 or better, but Netscape 7/Mozilla 1.5 or better is also possible.

8.2.1. Running a Case

In Figure 8.2 an instance d3web.Dialog is shown. In general, the dialog is divided into three main panes: The navigation pane, the acquisition pane, and the result pane.



Observations
Technical examinations
Idle speed system check
Air filter check
Ignition timing check
Air intake system check
Battery check

What is the color of the exhaust pipe?
brown, grey, light grey, [✓ sooty black], unknown

What fuel is used?
[✓ unleaded], leaded, unknown

Average mileage /100km
unknown

Real mileage /100km
unknown

Engine noises
knocking, ringing, no /else, unknown

Engine start
engine starts, engine barely starts, [✓ does not start], unknown

Starter
Turns over, Does not turn over, unknown

Probable diagnoses
Clogged air filter

Suggested diagnoses
Empty battery

Case comparison
Start case comparison.

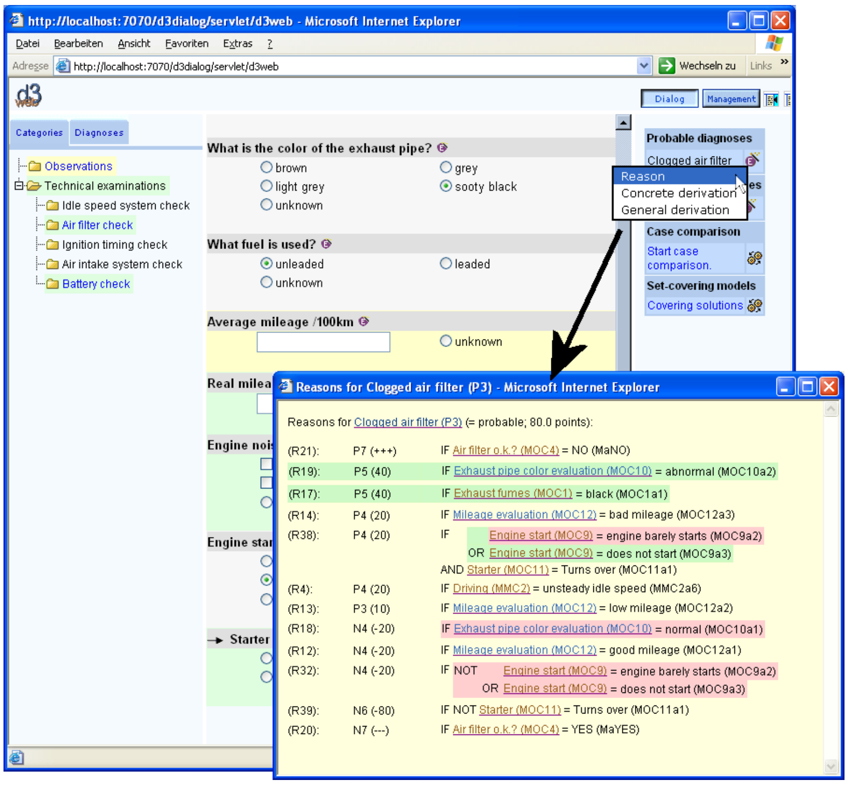
Set-covering models
Covering solutions

Figure 8.2: An instance user dialog of d3web.Dialog.

The Navigation Pane The left pane shows the navigation pane, which basically provides the question set hierarchy and the diagnosis hierarchy, respectively. Indicated question sets, i.e., question sets to be asked to the user, are colored in green. The currently

[1]Java Runtime Environment

processed question set is colored in yellow and presented in the center pane, i.e., the acquisition pane. Already answered question sets are colored in gray. Typically, the system guides the user dialog by indicating question sets according to the specified strategic knowledge. However, the user is also able to manually indicate question sets, by simply clicking the particular question set. Then, the selected question set will be presented in the center pane, immediately.



What is the color of the exhaust pipe?
brown / grey / light grey / [✓ sooty black] / unknown

What fuel is used?
[✓ unleaded] / leaded / unknown

Average mileage /100km
unknown

Reasons for Clogged air filter (P3) - Microsoft Internet Explorer

Reasons for Clogged air filter (P3) (= probable; 80.0 points):

```
(R21):   P7 (+++)   IF Air filter o.k.? (MOC4) = NO (MaNO)
(R19):   P5 (40)    IF Exhaust pipe color evaluation (MOC10) = abnormal (MOC10a2)
(R17):   P5 (40)    IF Exhaust fumes (MOC1) = black (MOC1a1)
(R14):   P4 (20)    IF Mileage evaluation (MOC12) = bad mileage (MOC12a3)
(R38):   P4 (20)    IF        Engine start (MOC9) = engine barely starts (MOC9a2)
                          OR  Engine start (MOC9) = does not start (MOC9a3)
                    AND Starter (MOC11) = Turns over (MOC11a1)
(R4):    P4 (20)    IF Driving (MMC2) = unsteady idle speed (MMC2a6)
(R13):   P3 (10)    IF Mileage evaluation (MOC12) = low mileage (MOC12a2)
(R18):   N4 (-20)   IF Exhaust pipe color evaluation (MOC10) = normal (MOC10a1)
(R12):   N4 (-20)   IF Mileage evaluation (MOC12) = good mileage (MOC12a1)
(R32):   N4 (-20)   IF NOT    Engine start (MOC9) = engine barely starts (MOC9a2)
                          OR  Engine start (MOC9) = does not start (MOC9a3)
(R39):   N6 (-80)   IF NOT Starter (MOC11) = Turns over (MOC11a1)
(R20):   N7 (---)   IF Air filter o.k.? (MOC4) = YES (MaYES)
```

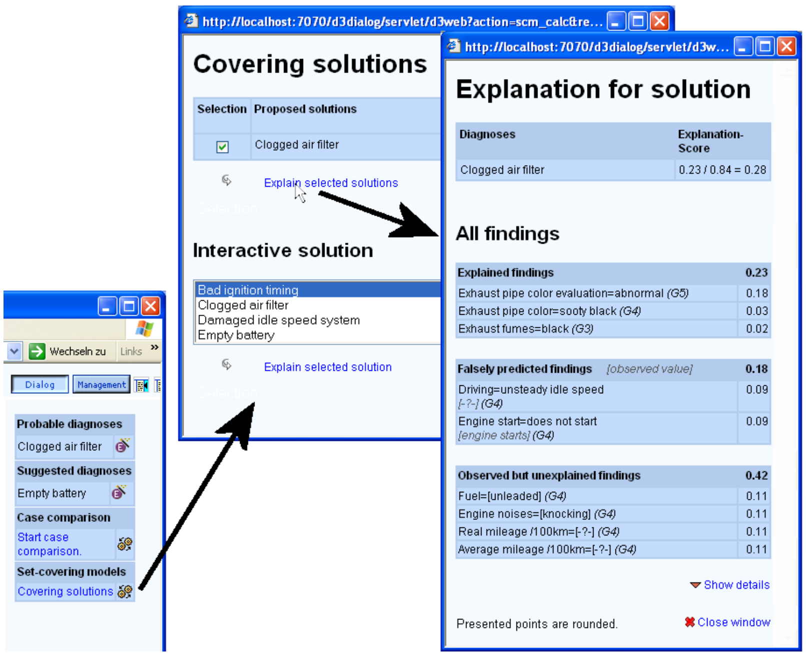
Figure 8.3: The derivation of a given diagnosis can be explained: For the score-based knowledge representation, the fired scoring rules are shown in green color. Rules with a false evaluation of their condition are colored in red.

The Acquisition Pane The acquisition pane is located in the center of the application, and presents the currently asked questions to be answered by the user. Analogously to the navigation pane, the already answered questions are colored in gray, the pending questions are colored in green, and the currently asked question is colored in yellow. Follow-up questions can be subsequently indicated and are inserted below the primary question, marked by an arrow → (e.g., question *Starter* shown in Figure 8.2). Answered question sets are completed the continue button. The case is instantly finished without answering pending questions by the result button.

The Result Pane The result pane at the right shows the currently derived diagnoses with respect to the collected findings; suggested and already established diagnoses are displayed. The probable diagnoses are the diagnoses, for which the score-based or categorical structural knowledge has derived an established state; the suggested diagnoses are the diagnoses, for the score-based structural knowledge has derived a suggested state. The derivation of a single diagnosis can be explained by clicking the corresponding *E* button.

The derivation of the diagnoses in Figure 8.3 is based on available score-based knowledge. Furthermore, the current case can be evaluated by using a case-based reasoning plug-in (cases are required), and a set-covering reasoning plug-in (set-covering knowledge required).

Figure 8.4 shows the derived solutions based on set-covering knowledge (*Covering solutions*). Then, solutions can be explained in more detail or further diagnoses can be interactively evaluated using set-covering knowledge.



http://localhost:7070/d3dialog/servlet/d3web?action=scm_calc&re...
Wechseln zu
Links
Dialog
Management
http://localhost:7070/d3dialog/servlet/d3w...

Probable diagnoses
Clogged air filter
Suggested diagnoses
Empty battery
Case comparison
Start case comparison.
Set-covering models
Covering solutions

Covering solutions

Selection	Proposed solutions
✓	Clogged air filter

Explain selected solutions

Interactive solution

Bad ignition timing
Clogged air filter
Damaged idle speed system
Empty battery

Explain selected solution

Explanation for solution

Diagnoses	Explanation-Score
Clogged air filter	0.23 / 0.84 = 0.28

All findings

Explained findings	0.23
Exhaust pipe color evaluation=abnormal *(G5)*	0.18
Exhaust pipe color=sooty black *(G4)*	0.03
Exhaust fumes=black *(G3)*	0.02

Falsely predicted findings *[observed value]*	0.18
Driving=unsteady idle speed *[-?-] (G4)*	0.09
Engine start=does not start *[engine starts] (G4)*	0.09

Observed but unexplained findings	0.42
Fuel=[unleaded] *(G4)*	0.11
Engine noises=[knocking] *(G4)*	0.11
Real mileage /100km=[-?-] *(G4)*	0.11
Average mileage /100km=[-?-] *(G4)*	0.11

Show details

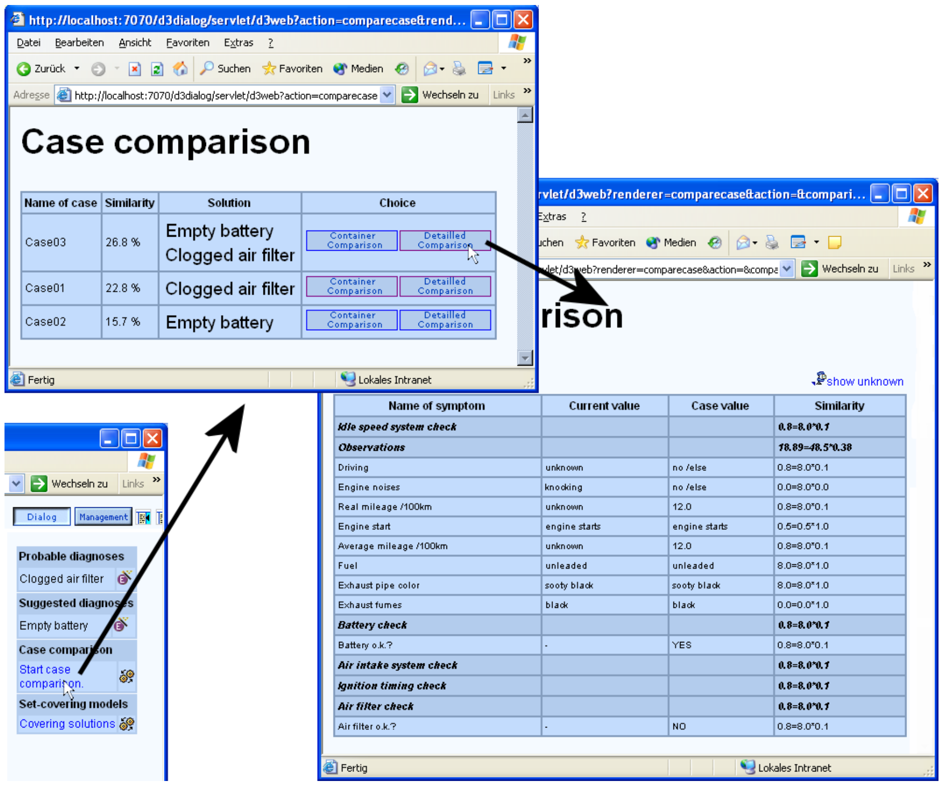
Presented points are rounded.

Close window

Figure 8.4: The solutions based on set-covering knowledge can be derived on demand; additional diagnoses can be evaluated interactively.

Alternatively, the most similar cases for the current case can be retrieved using case-based structural knowledge (*Start case comparison*). If cases are available, then the retrieved cases are listed in a table, and detailed comparisons of the particular cases can be requested as shown in Figure 8.5.

At the top of the result pane, the user can access a menu for switching from the dialog to the admin menu. This menu provides menus for starting a new case, saving the current case, and loading a previously saved case. Additionally, there are menus for uploading a new knowledge base, switching to another available knowledge base, and for accessing the advanced settings of the dialog.


http://localhost:7070/d3dialog/servlet/d3web?action=comparecase&rend...
Datei Bearbeiten Ansicht Favoriten Extras ?
Zurück Suchen Favoriten Medien
Adresse http://localhost:7070/d3dialog/servlet/d3web?action=comparecase
Wechseln zu Links
Fertig
Lokales Intranet
rvlet/d3web?renderer=comparecase&action=&compari...
show unknown
Dialog Management
Probable diagnoses
Clogged air filter
Suggested diagnoses
Empty battery
Case comparison
Start case comparison.
Set-covering models
Covering solutions

Case comparison

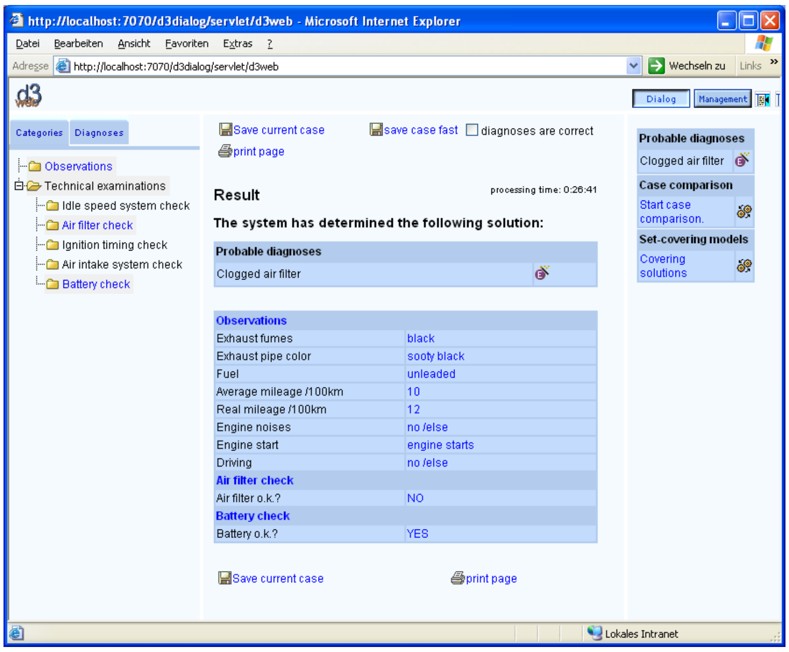
Name of case	Similarity	Solution	Choice
Case03	26.8 %	Empty battery Clogged air filter	Container Comparison / Detailled Comparison
Case01	22.8 %	Clogged air filter	Container Comparison / Detailled Comparison
Case02	15.7 %	Empty battery	Container Comparison / Detailled Comparison

Name of symptom	Current value	Case value	Similarity
Idle speed system check			**0.8=8.0*0.1**
Observations			**18.89=48.5*0.38**
Driving	unknown	no /else	0.8=8.0*0.1
Engine noises	knocking	no /else	0.0=8.0*0.0
Real mileage /100km	unknown	12.0	0.8=8.0*0.1
Engine start	engine starts	engine starts	0.5=0.5*1.0
Average mileage /100km	unknown	12.0	0.8=8.0*0.1
Fuel	unleaded	unleaded	8.0=8.0*1.0
Exhaust pipe color	sooty black	sooty black	8.0=8.0*1.0
Exhaust fumes	black	black	0.0=0.0*1.0
Battery check			**0.8=8.0*0.1**
Battery o.k.?	-	YES	0.8=8.0*0.1
Air intake system check			**0.8=8.0*0.1**
Ignition timing check			**0.8=8.0*0.1**
Air filter check			**0.8=8.0*0.1**
Air filter o.k.?	-	NO	0.8=8.0*0.1

Figure 8.5: For the current case the most similar cases in the available case base can be retrieved, and a detailed comparison of the cases can be received on demand.

8.2.2. Concluding a Case

A running case is finished, either if the user manually clicks the *Result* button, or if the system has presented all indicated questions. Then, in the center pane an overview is given showing the derived diagnoses and a summary of the collected findings. The case can be



Result

processing time: 0:26:41

The system has determined the following solution:

Probable diagnoses
Clogged air filter

Observations	
Exhaust fumes	black
Exhaust pipe color	sooty black
Fuel	unleaded
Average mileage /100km	10
Real mileage /100km	12
Engine noises	no /else
Engine start	engine starts
Driving	no /else
Air filter check	
Air filter o.k.?	NO
Battery check	
Battery o.k.?	YES

Save current case print page

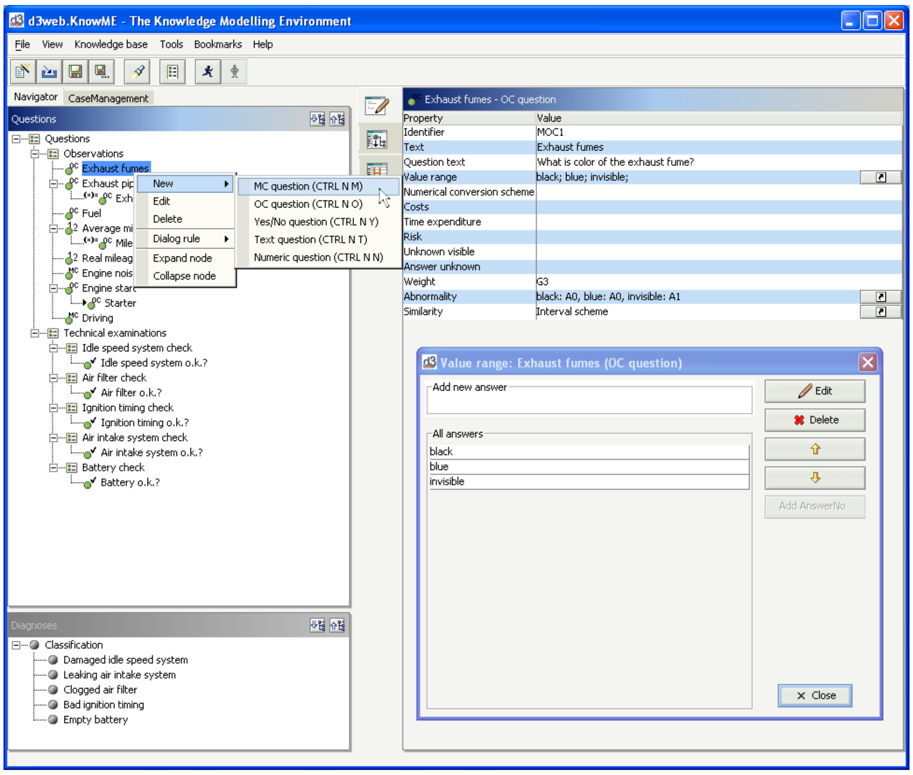
Figure 8.6: The result pane of a finished case: Solutions and answered questions (*Observations*) are shown in the center pane; the case can be stored and printed.

stored and annotated with additional information, e.g., the author of the case, a comment on the case, and comments on the derived diagnoses.

8.3. Developing Knowledge Systems: d3web.KnowME

The system d3web.KnowME is a rich client application written in Java. A Java JRE 1.4 or better is expected to be installed on the computer. Alternatively, a distribution of d3web.KnowME containing an appropriate JRE is available for download. d3web.KnowME is an integrated workbench for the agile development of diagnostic knowledge systems. It provides editors for the construction, the validation, and the maintenance of diagnostic knowledge. Technically, d3web.KnowME is a framework which initially consists of a small application with the basic editors for the construction of the question hierarchy and the diagnosis hierarchy. Furthermore, the system is extended by plug-ins providing additional functionality. For example, editors for the development of structural and strategic knowledge are embedded as plug-ins as well as editors for the corresponding test knowledge.

In Figure 8.7, the system d3web.KnowME is shown. In the left pane the question hierarchy and the diagnosis hierarchy can be edited. These fixed editors are extended by plug-ins, mostly embedded in the right pane of the application. Specialized plug-ins



Questions
- Questions
  - Observations
    - Exhaust fumes
    - Exhaust pip
      - Exh
    - Fuel
    - Average mi
      - Mile
    - Real mileag
    - Engine nois
    - Engine star
      - Starter
    - Driving
  - Technical examinations
    - Idle speed system check
      - Idle speed system o.k.?
    - Air filter check
      - Air filter o.k.?
    - Ignition timing check
      - Ignition timing o.k.?
    - Air intake system check
      - Air intake system o.k.?
    - Battery check
      - Battery o.k.?

Property	Value
Identifier	MOC1
Text	Exhaust fumes
Question text	What is color of the exhaust fume?
Value range	black; blue; invisible;
Numerical conversion scheme	
Costs	
Time expenditure	
Risk	
Unknown visible	
Answer unknown	
Weight	G3
Abnormality	black: A0, blue: A0, invisible: A1
Similarity	Interval scheme

All answers: black, blue, invisible

Diagnoses
- Classification
  - Damaged idle speed system
  - Leaking air intake system
  - Clogged air filter
  - Bad ignition timing
  - Empty battery

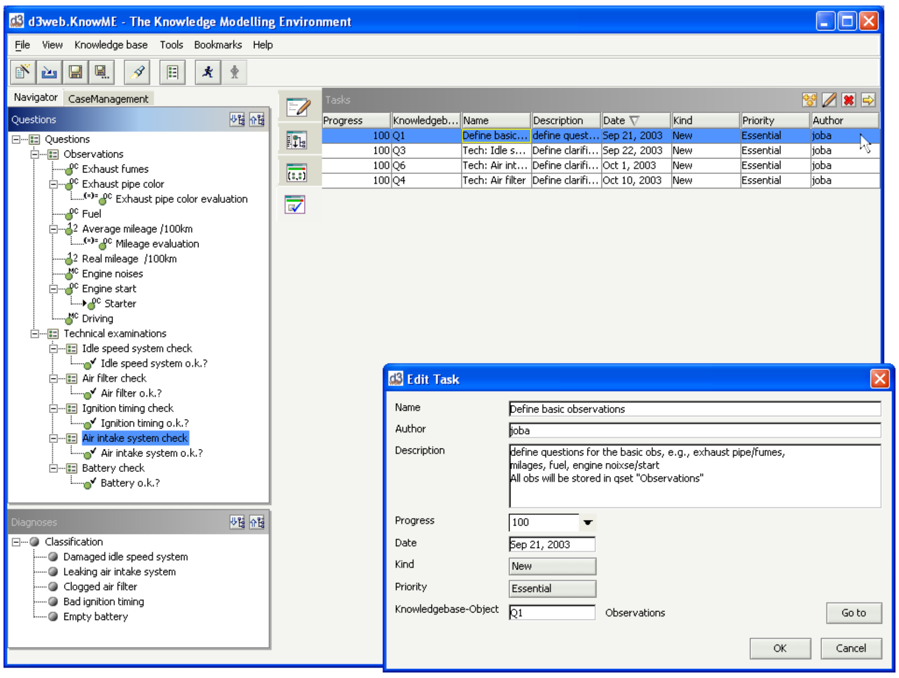
Figure 8.7: The standard view of d3web.KnowME, showing the question and diagnosis hierarchy at the left pane and the property editor at the center pane.

are the automated test tool d3web.QuaSiModu and a tool for automated restructurings. The architecture allows for a flexible extension of the d3web.KnowME framework, and has been proven to be suitable during the last years of development. For example, the d3web.KnowME application also has been configured with specialized plug-ins in order to facilitate the development of intelligent tutoring systems, i.e., d3web.Train. The implemented knowledge base is stored in a zipped jar file, containing the knowledge of all defined knowledge containers as well as corresponding test knowledge and additional development documentation, e.g., stories collected during the planning game.

In the following sections, we will introduce d3web.KnowME and its editors used for the agile development of diagnostic knowledge systems in more detail. The functionality of d3web.KnowME will be explained using an exemplary knowledge base considering the fault diagnosis of cars. This knowledge base is only used for demonstrating issues, and is not intended to model real world car diagnosis.

8.3.1. The Planning Game

The *task editor* of d3web.KnowME is a plug-in for defining, steering, and finishing stories, gathered during the planning game. In Figure 8.8 the task editor is shown. A new story



Questions
- Questions
  - Observations
    - Exhaust fumes
    - Exhaust pipe color
      - Exhaust pipe color evaluation
    - Fuel
    - Average mileage /100km
      - Mileage evaluation
    - Real mileage /100km
    - Engine noises
    - Engine start
      - Starter
    - Driving
  - Technical examinations
    - Idle speed system check
      - Idle speed system o.k.?
    - Air filter check
      - Air filter o.k.?
    - Ignition timing check
      - Ignition timing o.k.?
    - Air intake system check
      - Air intake system o.k.?
    - Battery check
      - Battery o.k.?

Diagnoses
- Classification
  - Damaged idle speed system
  - Leaking air intake system
  - Clogged air filter
  - Bad ignition timing
  - Empty battery

Tasks

Progress	Knowledgeb...	Name	Description	Date	Kind	Priority	Author
100	Q1	Define basic...	define quest...	Sep 21, 2003	New	Essential	joba
100	Q3	Tech: Idle s...	Define clarifi...	Sep 22, 2003	New	Essential	joba
100	Q6	Tech: Air int...	Define clarifi...	Oct 1, 2003	New	Essential	joba
100	Q4	Tech: Air filter	Define clarifi...	Oct 10, 2003	New	Essential	joba

Edit Task

Name	Define basic observations
Author	joba
Description	define questions for the basic obs, e.g., exhaust pipe/fumes, milages, fuel, engine noixse/start All obs will be stored in qset "Observations"
Progress	100
Date	Sep 21, 2003
Kind	New
Priority	Essential
Knowledgebase-Object	Q1 Observations

Figure 8.8: The documentation of stories using the task editor of d3web.KnowME.

is defined by starting a new task. The name of the task, the author of the task, and a description of the task needs to be specified. The progress of the task can also be stated, and it ranges between 0 and 100; the value 0 stands for a currently not started task, and the value 100 represents an already finished task. According to the planning game defined in Section 2.2.2 (p. 18), a task can be attached with information about the type (new, extension, correction) of the story and its priority (nice to have, significant, essential). An already existing knowledge base object (e.g., diagnosis, question) can be linked with the task, if available. This feature allows for a convenient integration with the other editors. All defined tasks are documented in the task editor in order to supply a detailed overview of the already finished or still pending tasks. For convenience, finished tasks (i.e., with progress 100) can be hidden in the task view.

8.3.2. Automated Tests with d3web.QuaSiModu

For the agile development the integration of appropriate test tools is a significant requirement. The knowledge modeling environment d3web.KnowME offers the plug-in d3web.QuaSiModu (*Qua*litäts-*Si*cherungs-*Modu*l – quality assurance module) for testing implemented knowledge. The initial implementation and design goals of d3web.QuaSiModu are described in [3] in more detail. The plug-in d3web.QuaSiModu is shown in Figure 8.9.

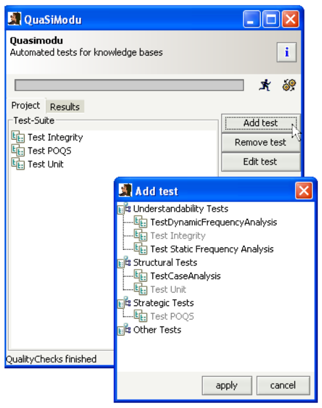
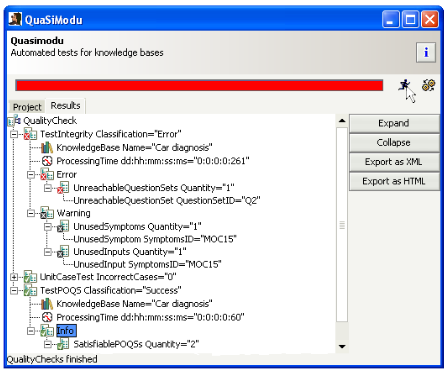


Figure 8.9: Definition of a test suite with d3web.QuaSiModu.

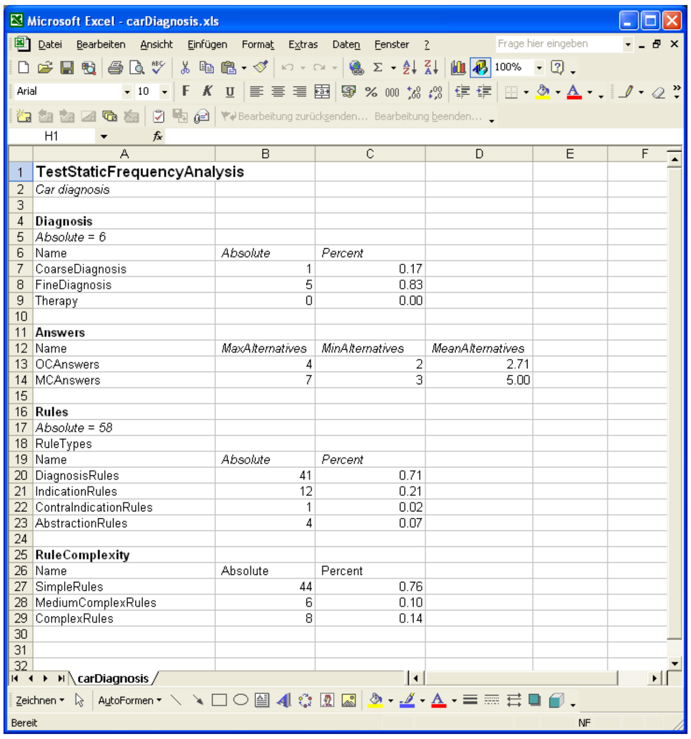


```
QualityCheck
  TestIntegrity Classification="Error"
    KnowledgeBase Name="Car diagnosis"
    ProcessingTime dd:hh:mm:ss:ms="0:0:0:0:261"
    Error
      UnreachableQuestionSets Quantity="1"
        UnreachableQuestionSet QuestionSetID="Q2"
    Warning
      UnusedSymptoms Quantity="1"
        UnusedSymptom SymptomsID="MOC15"
      UnusedInputs Quantity="1"
        UnusedInput SymptomsID="MOC15"
  UnitCaseTest IncorrectCases="0"
  TestPOQS Classification="Success"
    KnowledgeBase Name="Car diagnosis"
    ProcessingTime dd:hh:mm:ss:ms="0:0:0:0:60"
    Info
      SatisfiablePOQSs Quantity="2"
```

Figure 8.10: The verbose result view of d3web.QuaSiModu.

For example, d3web.QuaSiModu offers methods for empirical testing, for static and case-based ontological testing, and for static analysis of structural and strategic knowledge. The test suite is defined by selecting tests for the current knowledge system project (e.g., see Figure 8.9).

All tests are processed automatically and a visual feedback is given by a colored status bar, which turns green, if all tests have been finished successfully. If any of the tests reports an error, then the color of the status bar changes to red. This metaphor is known from JUnit [44], a well-known test framework for the programming language Java. In addition to this boolean visual feedback we added the yellow status bar to d3web.QuaSiModu; the bar turns yellow, if warnings but no errors are reported. Furthermore, a more detailed report of each test result is presented (e.g., see Figure 8.10). For a convenient analysis by the domain experts an export to Microsoft Excel© is provided , as shown in Figure 8.11 . For the practical application of automatic tests it is necessary to facilitate an *ignore* feature for detected warnings and errors. Then, the developer of the knowledge system has been notified about this irregularity, but he cannot or does not want to perform any


Microsoft Excel - carDiagnosis.xls
Datei Bearbeiten Ansicht Einfügen Format Extras Daten Fenster ?
Frage hier eingeben
100%
Arial
10
Bearbeitung zurücksenden... Bearbeitung beenden...
H1
carDiagnosis
Zeichnen
AutoFormen
Bereit
NF

	A	B	C	D	E	F
1	**TestStaticFrequencyAnalysis**					
2	*Car diagnosis*					
3						
4	**Diagnosis**					
5	*Absolute = 6*					
6	Name	*Absolute*	*Percent*			
7	CoarseDiagnosis	1	0.17			
8	FineDiagnosis	5	0.83			
9	Therapy	0	0.00			
10						
11	**Answers**					
12	Name	*MaxAlternatives*	*MinAlternatives*	*MeanAlternatives*		
13	OCAnswers	4	2	2.71		
14	MCAnswers	7	3	5.00		
15						
16	**Rules**					
17	*Absolute = 58*					
18	RuleTypes					
19	Name	*Absolute*	*Percent*			
20	DiagnosisRules	41	0.71			
21	IndicationRules	12	0.21			
22	ContraIndicationRules	1	0.02			
23	AbstractionRules	4	0.07			
24						
25	**RuleComplexity**					
26	Name	Absolute	Percent			
27	SimpleRules	44	0.76			
28	MediumComplexRules	6	0.10			
29	ComplexRules	8	0.14			
30						
31						
32						

Figure 8.11: Verbose result trace of the static frequency analysis exported to MS-Excel©.

action to remove this issue. Nevertheless, the warnings and errors on the ignore list need to be accessible for the developer any time. In the following sections we will explain the application of single test methods of d3web.QuaSiModu in more detail.

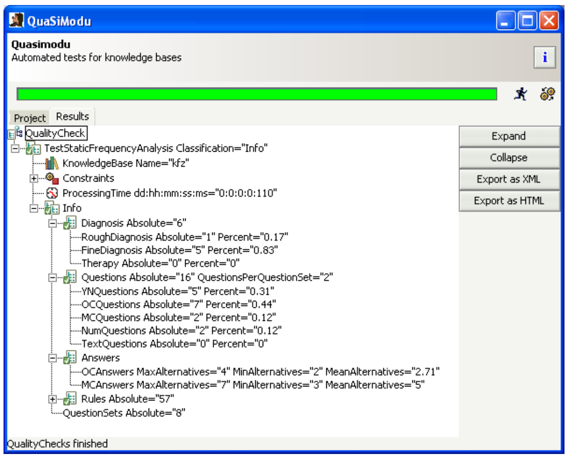
8.3.3. Handling Ontological Knowledge

Defining Ontological Knowledge

The ontological knowledge container is filled by defining the hierarchies of questions and diagnoses. Questions are grouped by question sets, and diagnoses can be structured by other diagnoses representing an even more coarse diagnostic concept. In Figure 8.7 (p. 160) the standard view of the d3web.KnowME system is shown. The left pane displays the hierarchies of questions and diagnoses in tree views. Question sets, questions, and diagnoses can be easily inserted, modified or deleted by using the context menu of the corresponding tree pane. The current location of a single question set, question or diagnosis can be simply changed by using drag and drop. However, using drag and drop the modification of the hierarchical relationship is not propagated to possibly attached knowledge (e.g., strategic knowledge) as it is done when using the corresponding restructuring method. The application of restructuring methods is explained in Section 8.3.9 (p. 177) in more detail. The property editor shown in the main pane of Figure 8.7 displays and modifies additional ontological properties of the currently selected knowledge base object, i.e., question set, question, and diagnosis. For example, the name of the object, its corresponding question text, the value range, and further properties can be viewed and edited.

Testing Ontological Knowledge

d3web.QuaSiModu offers a specialized test for inspecting ontological knowledge, i.e., static ontology testing (p. 53). Actually, the corresponding test provided by d3web.QuaSiModu is named *Test Static Frequency Analysis*, since it is also able to inspect static properties of structural and strategic knowledge.



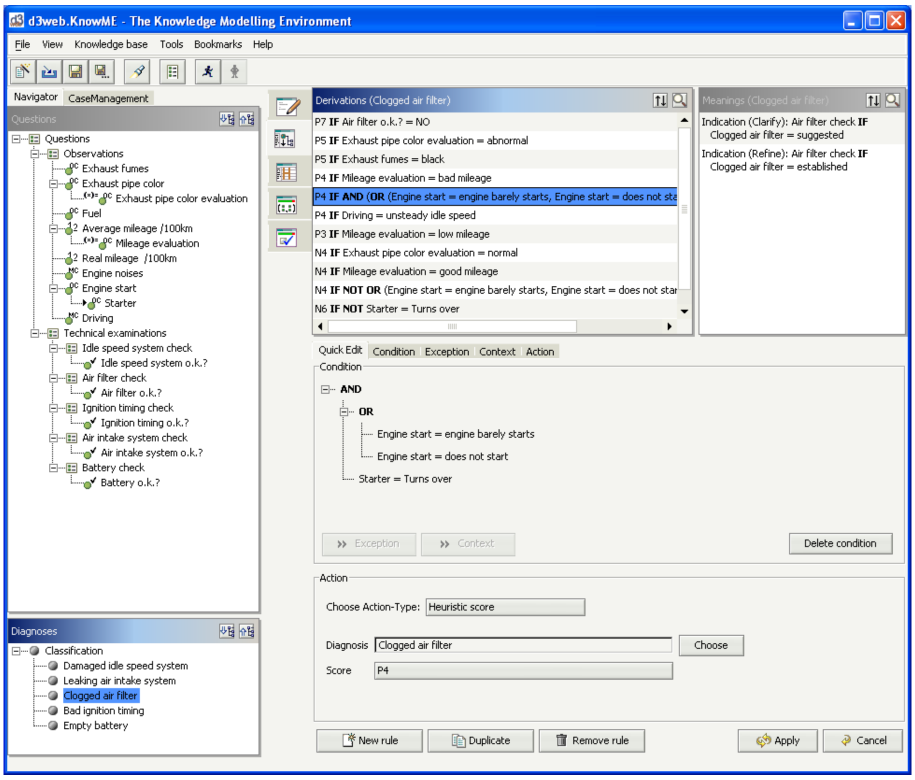
```
QualityCheck
  TestStaticFrequencyAnalysis Classification="Info"
    KnowledgeBase Name="kfz"
    Constraints
    ProcessingTime dd:hh:mm:ss:ms="0:0:0:0:110"
    Info
      Diagnosis Absolute="6"
        RoughDiagnosis Absolute="1" Percent="0.17"
        FineDiagnosis Absolute="5" Percent="0.83"
        Therapy Absolute="0" Percent="0"
      Questions Absolute="16" QuestionsPerQuestionSet="2"
        YNQuestions Absolute="5" Percent="0.31"
        OCQuestions Absolute="7" Percent="0.44"
        MCQuestions Absolute="2" Percent="0.12"
        NumQuestions Absolute="2" Percent="0.12"
        TextQuestions Absolute="0" Percent="0"
      Answers
        OCAnswers MaxAlternatives="4" MinAlternatives="2" MeanAlternatives="2.71"
        MCAnswers MaxAlternatives="7" MinAlternatives="3" MeanAlternatives="5"
      Rules Absolute="57"
      QuestionSets Absolute="8"
```

Figure 8.12: The results of the ontological tests are shown in a tree view.

In Figure 8.12 the results of the test method are shown. An overview of the static properties of the knowledge base is given, i.e., the occurrence of the different types of diagnoses and questions is reported in absolute numbers and percentages. The results can be exported to an XML file, which commonly is converted via XSL into a TAB-separated text file. Thus, the data can be analyzed with a spread sheet like Microsoft Excel© in a convenient way.

8.3.4. The General Rule Editor

The general rule editor, as shown in Figure 8.13, presents the rules related with the knowledge base object currently selected in the question or the diagnosis hierarchy. Thus, strategic knowledge (indication rules), structural abstraction knowledge (abstraction rules), and structural score-based knowledge (scoring rules) are jointly displayed and edited. At the top left pane of the rule editor the derivation rules of the selected object are shown, i.e., rules deriving a value for the object; in our example the diagnosis ”Clogged air filter”. The top right pane displays all rules, for which the selected object has a meaning, i.e., all rules that contain the selected object (”Clogged air filter”) on their condition part. If a rule in one of the top panes is selected, then the rule is presented in more detail in the center pane. In this pane, the rule can be modified, deleted or duplicated (cloned). The first tab of the pane offers the view and modification of the rule condition and the rule action. Due to their rare usage, the rule exception and the rule context are edited in separate tabs.


d3web.KnowME - The Knowledge Modelling Environment
File View Knowledge base Tools Bookmarks Help
Navigator CaseManagement
Questions
Diagnoses
Quick Edit Condition Exception Context Action
Exception Context Delete condition
Choose
New rule Duplicate Remove rule Apply Cancel

Questions
- Observations
  - Exhaust fumes
  - Exhaust pipe color
    - Exhaust pipe color evaluation
  - Fuel
  - Average mileage /100km
    - Mileage evaluation
  - Real mileage /100km
  - Engine noises
  - Engine start
    - Starter
  - Driving
- Technical examinations
  - Idle speed system check
    - Idle speed system o.k.?
  - Air filter check
    - Air filter o.k.?
  - Ignition timing check
    - Ignition timing o.k.?
  - Air intake system check
    - Air intake system o.k.?
  - Battery check
    - Battery o.k.?

Derivations (Clogged air filter)

P7 **IF** Air filter o.k.? = NO
P5 **IF** Exhaust pipe color evaluation = abnormal
P5 **IF** Exhaust fumes = black
P4 **IF** Mileage evaluation = bad mileage
P4 **IF AND** (**OR** (Engine start = engine barely starts, Engine start = does not sta
P4 **IF** Driving = unsteady idle speed
P3 **IF** Mileage evaluation = low mileage
N4 **IF** Exhaust pipe color evaluation = normal
N4 **IF** Mileage evaluation = good mileage
N4 **IF NOT OR** (Engine start = engine barely starts, Engine start = does not star
N6 **IF NOT** Starter = Turns over

Meanings (Clogged air filter)

Indication (Clarify): Air filter check **IF**
Clogged air filter = suggested
Indication (Refine): Air filter check **IF**
Clogged air filter = established

Condition
- **AND**
  - **OR**
    - Engine start = engine barely starts
    - Engine start = does not start
  - Starter = Turns over

Action

Choose Action-Type: Heuristic score
Diagnosis: Clogged air filter
Score: P4

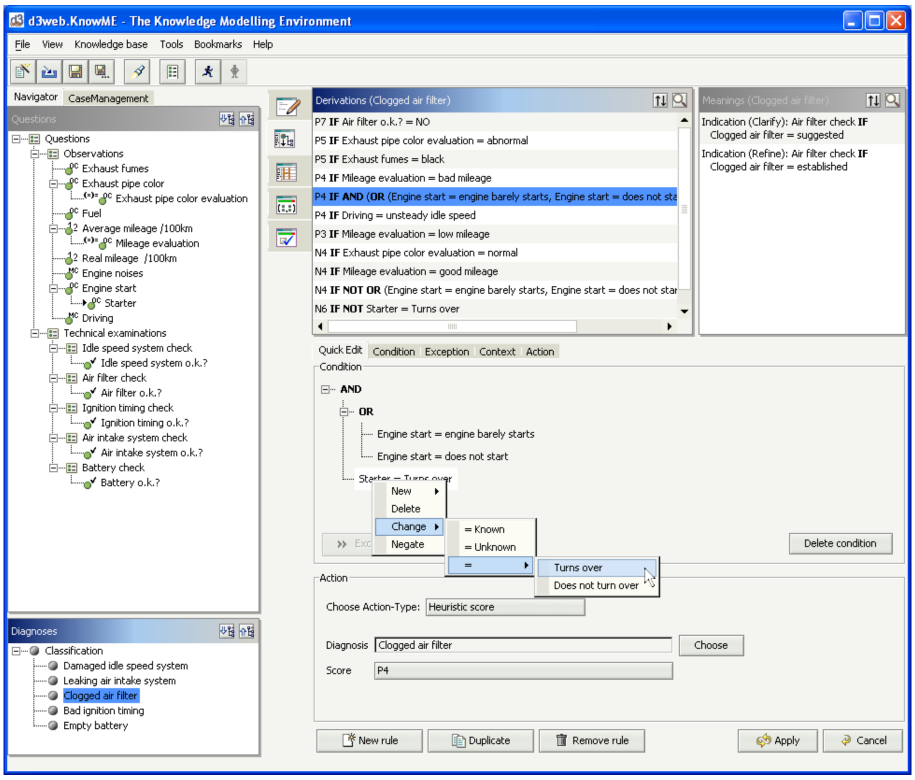
Diagnoses
- Classification
  - Damaged idle speed system
  - Leaking air intake system
  - Clogged air filter
  - Bad ignition timing
  - Empty battery

Figure 8.13: The general rule editor; the rules of the selected question or diagnosis are shown (e.g., diagnosis ”Clogged air filter”).

Rule conditions and rule actions can be visually modified as shown in Figure 8.14. Conditions are simply defined or modified using the context menu. Thus, new conditions can be created, the existing condition can be extended by a new sub-condition, or parts of the existing conditions can be changed. Rule actions are defined in the bottom pane. It is worth noticing, that the type of rule action specifies the kind of applied knowledge container. The editor provides action types for abstraction knowledge (*Set/Add question value*), for structural knowledge (*Heuristic score*), and for strategic knowledge (*Indicate/Contra-indicate question (set)*, *Clarify/Refine diagnosis*).

8.3.5. Handling Structural Knowledge

In the previous section we introduced the general rule editor as an all purpose editor for creating, modifying, and removing various kinds of rule-based knowledge. Besides strategic indication rules, rule-based abstraction knowledge, and categorical rules, the rule editor allows for the modification of score-based rules. However, for a structured acquisition of many score-based rules even more specialized and convenient editors are offered by the d3web.KnowME system. The *heuristic table* provides a specialized editor for simple scoring rules, and the *heuristic detail table* allows for the acquisition of one-level scoring rules.



P7 **IF** Air filter o.k.? = NO
P5 **IF** Exhaust pipe color evaluation = abnormal
P5 **IF** Exhaust fumes = black
P4 **IF** Mileage evaluation = bad mileage
P4 **IF AND** (**OR** (Engine start = engine barely starts, Engine start = does not sta
P4 **IF** Driving = unsteady idle speed
P3 **IF** Mileage evaluation = low mileage
N4 **IF** Exhaust pipe color evaluation = normal
N4 **IF** Mileage evaluation = good mileage
N4 **IF NOT OR** (Engine start = engine barely starts, Engine start = does not star
N6 **IF NOT** Starter = Turns over

Indication (Clarify): Air filter check **IF** Clogged air filter = suggested
Indication (Refine): Air filter check **IF** Clogged air filter = established

Condition
AND
OR
Engine start = engine barely starts
Engine start = does not start
Starter = Turns over

Action
Choose Action-Type: Heuristic score
Diagnosis: Clogged air filter
Score: P4

Figure 8.14: Rule conditions can be easily defined and modified by using the context menu.

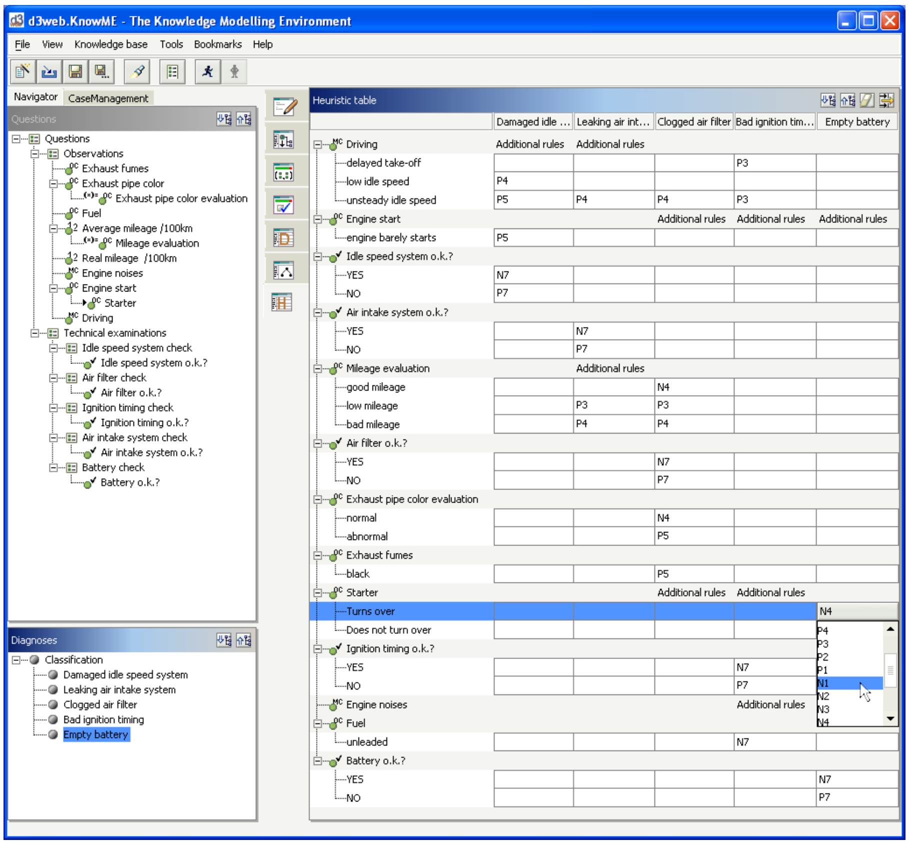
Defining Structural Knowledge

In addition to the heuristic table and the heuristic detail table we also introduce the set-covering table for defining symbolic set-covering knowledge.

The Heuristic Table In Figure 8.15 the heuristic table editor is shown, which is used for an efficient capture of simple scoring rules. For example, simple scoring rules are applied in the context of the DIAGNOSTIC SCORE pattern (p. 94).

The confirmation categories of the simple scoring rules are presented in the table cells, for which the column of a cell specifies the scored diagnosis of the rule, and the corresponding line specifies the simple condition of the rule. For the definition of a new rule the question and diagnosis simply need to be dragged from the hierarchies into the table. The rule is created by choosing an appropriate confirmation category in the corresponding table cell. For a convenient presentation, lines of selected questions can be hidden on demand. Additionally, lines containing no confirmation categories can be hidden as well.

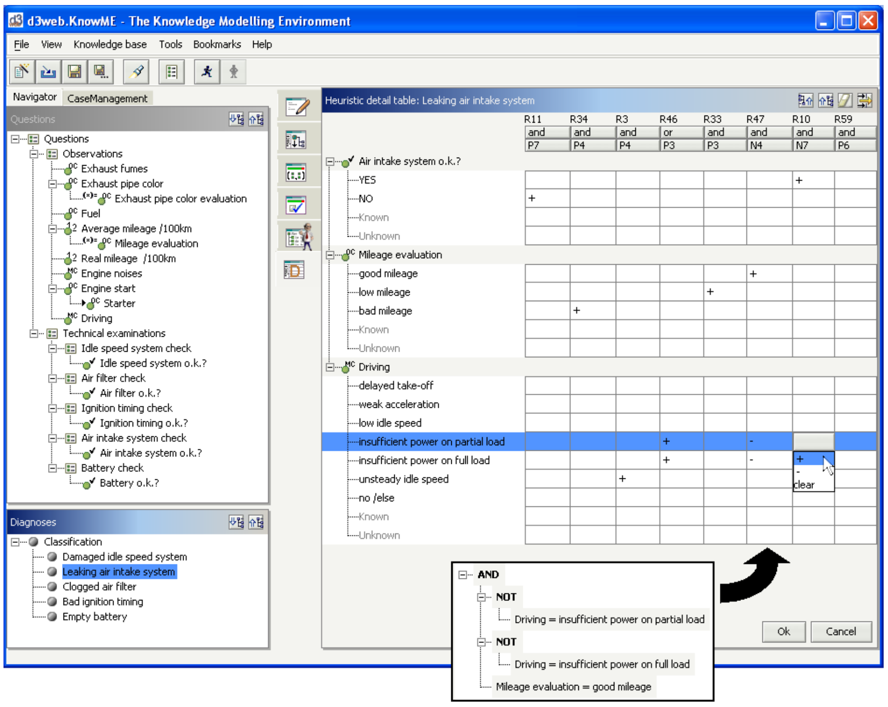
Heuristic Detail Table The heuristic detail table editor is used for the definition of one-level rules, e.g., applied by the HEURISTIC DECISION TABLE pattern [106]. As shown in Figure 8.16 the heuristic detail table presents all one-level rules for a selected diagnosis, which needs to be simply dragged from the diagnosis hierarchy into the table.



	Damaged idle …	Leaking air int…	Clogged air filter	Bad ignition tim…	Empty battery
Driving	Additional rules	Additional rules			
delayed take-off				P3	
low idle speed	P4				
unsteady idle speed	P5	P4	P4	P3	
Engine start			Additional rules	Additional rules	Additional rules
engine barely starts	P5				
Idle speed system o.k.?					
YES	N7				
NO	P7				
Air intake system o.k.?					
YES		N7			
NO		P7			
Mileage evaluation		Additional rules			
good mileage			N4		
low mileage		P3	P3		
bad mileage		P4	P4		
Air filter o.k.?					
YES			N7		
NO			P7		
Exhaust pipe color evaluation					
normal			N4		
abnormal			P5		
Exhaust fumes					
black			P5		
Starter			Additional rules	Additional rules	
Turns over					N4
Does not turn over					
Ignition timing o.k.?					
YES				N7	
NO				P7	
Engine noises				Additional rules	
Fuel					
unleaded				N7	
Battery o.k.?					
YES					N7
NO					P7

Figure 8.15: Simple scoring rules are easily defined and viewed using the editor heuristic table.

Each rule is represented by a column of the table. In the header of a column the rule identifier, the logical connector of the single conditions, and the specified confirmation category is displayed. Filled cells in the column indicate, whether the corresponding question/value pair is contained as a single sub-condition; a + indicates a positive inclusion of the sub-condition, and a − indicates a negated inclusion of the sub-condition. For example, the sixth rule column (id: R47) of the heuristic detail table shown in Figure 8.16 represents the one-level rule shown in the black box of the figure. New rules are created by simply filling the last (and empty) column in the table editor. New questions for the rule can be introduced by dragging the question from the question hierarchy into the table. For a convenient handling empty lines can be hidden, i.e., lines in the table that neither contain a + or a − sign.


d3web.KnowME - The Knowledge Modelling Environment
File View Knowledge base Tools Bookmarks Help
Navigator CaseManagement
Questions
Diagnoses
Ok
Cancel

Questions
- Observations
  - Exhaust fumes
  - Exhaust pipe color
    - Exhaust pipe color evaluation
  - Fuel
  - Average mileage /100km
    - Mileage evaluation
  - Real mileage /100km
  - Engine noises
  - Engine start
    - Starter
  - Driving
- Technical examinations
  - Idle speed system check
    - Idle speed system o.k.?
  - Air filter check
    - Air filter o.k.?
  - Ignition timing check
    - Ignition timing o.k.?
  - Air intake system check
    - Air intake system o.k.?
  - Battery check
    - Battery o.k.?

Classification
- Damaged idle speed system
- Leaking air intake system
- Clogged air filter
- Bad ignition timing
- Empty battery

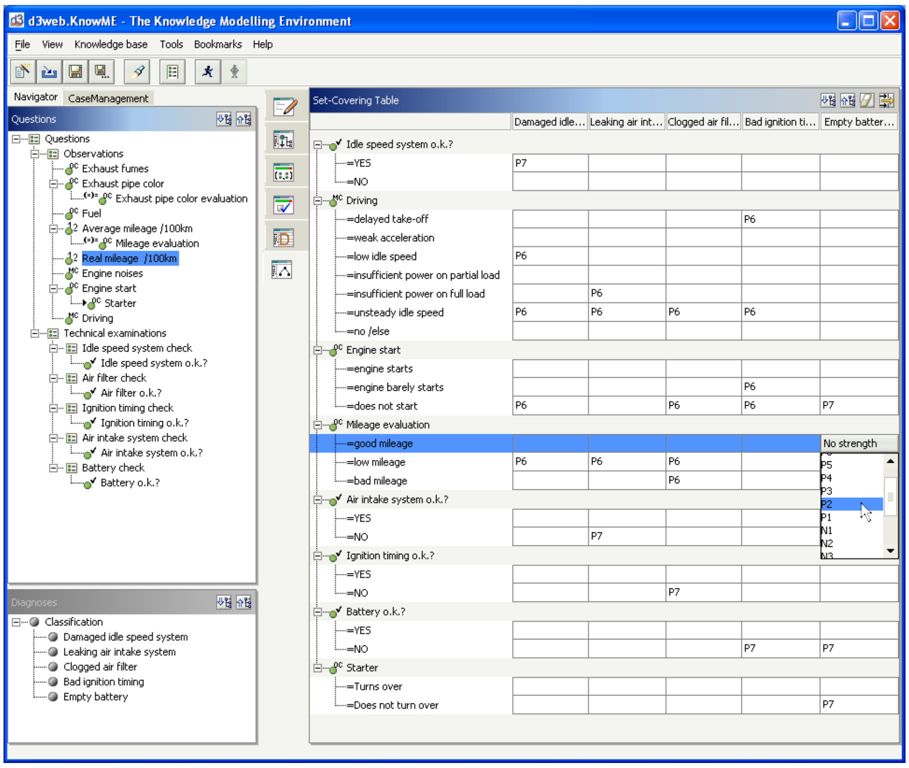
Heuristic detail table: Leaking air intake system

	R11	R34	R3	R46	R33	R47	R10	R59
	and	and	and	or	and	and	and	and
	P7	P4	P4	P3	P3	N4	N7	P6
Air intake system o.k.?								
YES							+	
NO	+							
Known								
Unknown								
Mileage evaluation								
good mileage						+		
low mileage					+			
bad mileage		+						
Known								
Unknown								
Driving								
delayed take-off								
weak acceleration								
low idle speed								
insufficient power on partial load				+		-		
insufficient power on full load				+		-	+ - clear	
unsteady idle speed			+					
no /else								
Known								
Unknown								

- AND
  - NOT
    - Driving = insufficient power on partial load
  - NOT
    - Driving = insufficient power on full load
  - Mileage evaluation = good mileage

Figure 8.16: The editor heuristic detail table is appropriate for defining and viewing one-level rules.

Set-Covering Table The set-covering table is applied for the definition of set-covering knowledge using symbolic confirmation strengths. In Figure 8.17 the set-covering table is shown. Then, each cell of the table specifies a set-covering relation with the diagnosis


d3web.KnowME - The Knowledge Modelling Environment
File View Knowledge base Tools Bookmarks Help
Navigator CaseManagement
Questions
Diagnoses
Set-Covering Table

Questions
- Questions
  - Observations
    - Exhaust fumes
    - Exhaust pipe color
      - Exhaust pipe color evaluation
    - Fuel
    - Average mileage /100km
      - Mileage evaluation
    - Real mileage /100km
    - Engine noises
    - Engine start
      - Starter
    - Driving
  - Technical examinations
    - Idle speed system check
      - Idle speed system o.k.?
    - Air filter check
      - Air filter o.k.?
    - Ignition timing check
      - Ignition timing o.k.?
    - Air intake system check
      - Air intake system o.k.?
    - Battery check
      - Battery o.k.?

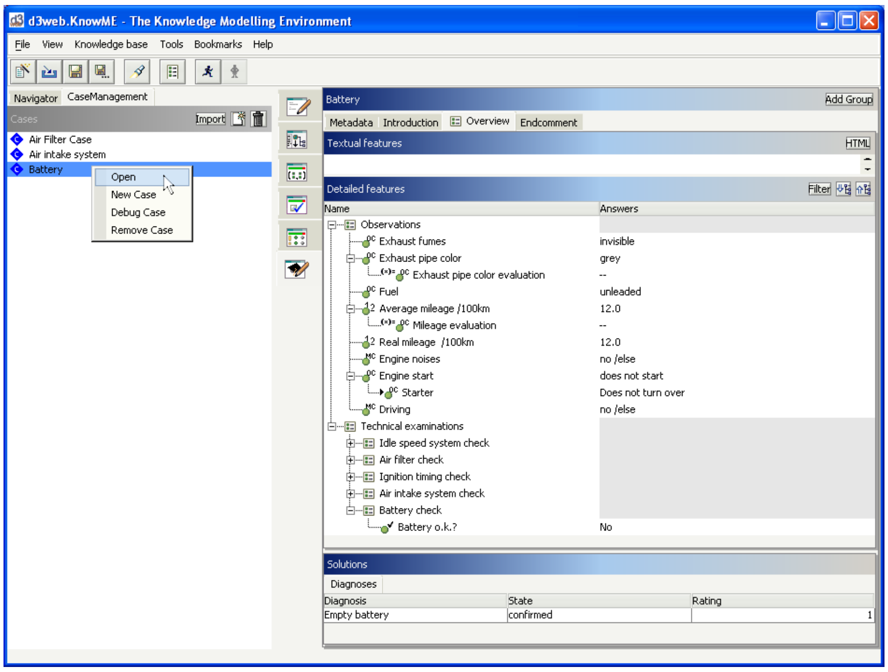
Diagnoses
- Classification
  - Damaged idle speed system
  - Leaking air intake system
  - Clogged air filter
  - Bad ignition timing
  - Empty battery

	Damaged idle…	Leaking air int…	Clogged air fil…	Bad ignition ti…	Empty batter…
Idle speed system o.k.?					
=YES	P7				
=NO					
Driving					
=delayed take-off				P6	
=weak acceleration					
=low idle speed	P6				
=insufficient power on partial load					
=insufficient power on full load		P6			
=unsteady idle speed	P6	P6	P6	P6	
=no /else					
Engine start					
=engine starts					
=engine barely starts				P6	
=does not start	P6		P6	P6	P7
Mileage evaluation					
=good mileage					No strength
=low mileage	P6	P6	P6		
=bad mileage			P6		
Air intake system o.k.?					
=YES					
=NO		P7			
Ignition timing o.k.?					
=YES					
=NO			P7		
Battery o.k.?					
=YES					
=NO				P7	P7
Starter					
=Turns over					
=Does not turn over					P7

Figure 8.17: The editor set-covering table is appropriate for defining and viewing set-covering relations.

in the corresponding cell column as the cause of the set-covering relation, and the question/value pair in the corresponding cell line as the effect of the set-covering relation. The cell itself specifies the symbolic confirmation strength of the set-covering relation. If a cell is empty, then no set-covering relation is defined for the corresponding diagnosis-finding relation. Set-covering relations of a new diagnosis are created by dragging the diagnosis from the diagnosis hierarchy into the table, and filling related cells with symbolic confirmation strengths. Additional questions are introduced by dragging the questions from the question hierarchy into the table editor.

Case Management Editor The case management editor shown in Figure 8.18 facilitates the definition and modification of cases. Although, the editor originally was implemented for the development of case-based tutoring systems (i.e., d3web.Train), it is also appropriate for cases representing structural knowledge or test knowledge (e.g., for empirical testing).



Name	Answers
Observations	
Exhaust fumes	invisible
Exhaust pipe color	grey
Exhaust pipe color evaluation	--
Fuel	unleaded
Average mileage /100km	12.0
Mileage evaluation	--
Real mileage /100km	12.0
Engine noises	no /else
Engine start	does not start
Starter	Does not turn over
Driving	no /else
Technical examinations	
Idle speed system check	
Air filter check	
Ignition timing check	
Air intake system check	
Battery check	
Battery o.k.?	No

Diagnosis	State	Rating
Empty battery	confirmed	1

Figure 8.18: The case management editor allows for the simple definition of cases, either used as structural or test knowledge.

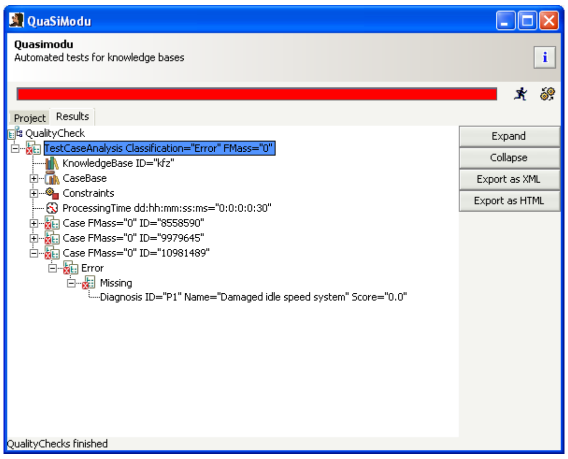
In the left pane of Figure 8.18 the available cases are listed. The context menu of the list offers the creation, deletion, and the import of cases. A double-click on a case opens the properties editor of the corresponding case in the main pane, shown at the right of Figure 8.18. For a detailed description of the case, the editor offers the *Metadata* tab, that, e.g., contains fields for defining the name, the author, the creation date, and a verbose comment on the case. The *Overview* tab, shown in the figure, actually enables the user to define the observed findings and derived solutions of the case. We do not consider the other tabs *Introduction* and *Endcomment*, since they are only used in the context of developing tutoring systems. Besides the definition of cases the developer is also able to define weights, similarities, and abnormality knowledge. This type of knowledge can be specified for each question in the corresponding ontological property editor, as shown in the main pane of Figure 8.7.

Testing Structural Knowledge

d3web.QuaSiModu offers various methods for testing structural knowledge (names of the corresponding d3web.QuaSiModu test methods are given in parentheses):

- Empirical testing (test case analysis)
- Inferential constraints (test unit)
- Static verification (test integrity)
- Dynamic rule base testing (test dynamic frequency analysis)
- Static rule base testing (test static frequency analysis)

Empirical testing The most prominent testing method for structural knowledge is empirical testing (p. 125). In Figure 8.19 the result of an exemplary test run is shown. The accuracy of the solved cases is calculated using the F-measure (p. 127). The test method



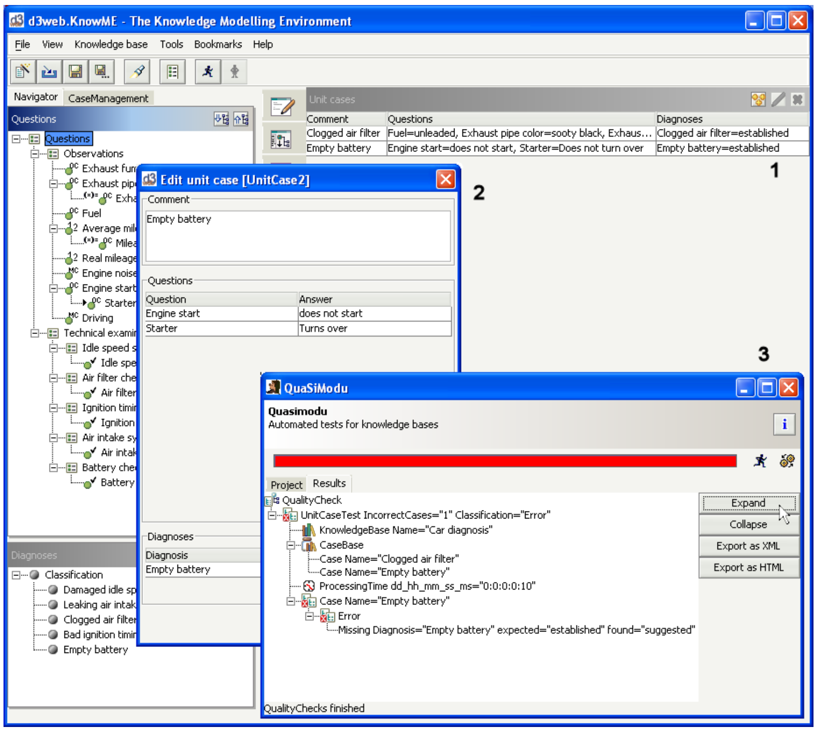
```
QualityCheck
  TestCaseAnalysis Classification="Error" FMass="0"
    KnowledgeBase ID="kfz"
    CaseBase
    Constraints
    ProcessingTime dd:hh:mm:ss:ms="0:0:0:0:30"
    Case FMass="0" ID="8558590"
    Case FMass="0" ID="9979645"
    Case FMass="0" ID="10981489"
      Error
        Missing
          Diagnosis ID="P1" Name="Damaged idle speed system" Score="0.0"
QualityChecks finished
```

Figure 8.19: d3web.QuaSiModu executing empirical testing with exemplary test cases.

can be attached with constraints specifying the minimum F-measure. If the computed F-measure falls below the given minimum measure, then an error is reported. The threshold for the expected minimum F-measure is appropriate for the application of many real world cases, which are assumed to be solved not with an 100% accuracy. However, the default minimum F-measure is set to the threshold value 1.

Inferential constraints Inferential constraints (p. 129) are checked using the unit cases method. In Figure 8.20 (1) the editor for defining unit cases is shown. Unit cases are listed in a tabular manner, containing a comment of the case, the assigned questions, and the set of diagnoses with corresponding states. A case is edited by double-clicking on the corresponding line in the table. Then, an editor window (2) is opened in order to modify the comment, the assigned questions, or diagnoses. New questions or diagnoses can be simply added by dragging them from the hierarchies. Figure 8.20 (3) shows the results of the test application. In the example, the unit case *Battery test* was not able to derive the specified diagnosis ”Empty battery”.

Static verification Static verification (p. 96) of rule-based knowledge is applied using the integrity test method, shown in Figure 8.21. For demonstration issues, a new diag-



Comment	Questions	Diagnoses
Clogged air filter	Fuel=unleaded, Exhaust pipe color=sooty black, Exhaus...	Clogged air filter=established
Empty battery	Engine start=does not start, Starter=Does not turn over	Empty battery=established

Comment: Empty battery

Question	Answer
Engine start	does not start
Starter	Turns over

Diagnosis
Empty battery

Quasimodu
Automated tests for knowledge bases

```
QualityCheck
  UnitCaseTest IncorrectCases="1" Classification="Error"
    KnowledgeBase Name="Car diagnosis"
    CaseBase
      Case Name="Clogged air filter"
      Case Name="Empty battery"
    ProcessingTime dd_hh_mm_ss_ms="0:0:0:0:10"
    Case Name="Empty battery"
      Error
        Missing Diagnosis="Empty battery" expected="established" found="suggested"
```

QualityChecks finished

Figure 8.20: Unit cases can be visually defined and used the inferential constraints test method.

nosis P6=TestDiagnosis has been inserted, which is not inferred by any rule (unreachable diagnosis). Furthermore, a derivation rule of the diagnosis was created, thus implying an unreachable rule.

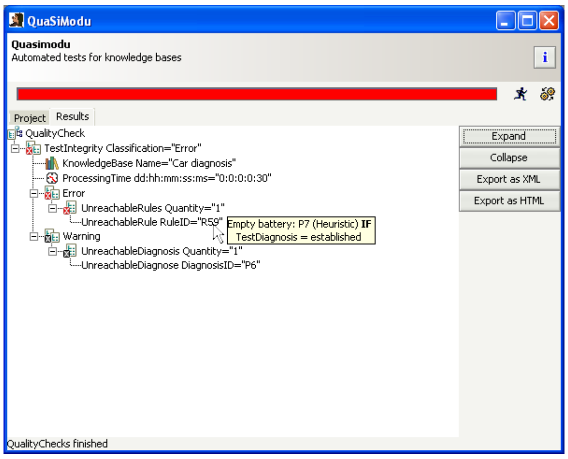
Dynamic rule base testing For dynamic rule base testing (p. 99) the dynamic frequency test is applied. For the convenient analysis of the test results in a spread sheet (e.g., Microsoft Excel©) a conversion into a TAB-separated list is provided.

Static rule base testing Static rule base testing (p. 98) is applied using the static frequency analysis, previously introduced for testing static properties of ontological knowledge. Besides ontological issues, the method also reports the number of rules categorized into rule kinds (e.g., scoring rules, indication rules) and rule complexities, i.e., simple, medium, and complex rules.

8.3.6. Handling Strategic Knowledge

Defining Strategic Knowledge

The system d3web.KnowME offers the implementation of the standardized indication using indication rules (p. 135). For the definition of indication rules the general rule editor is applied, which provides the construction of the following indication actions:


QuaSiModu

Quasimodu
Automated tests for knowledge bases

```
QualityCheck
  TestIntegrity Classification="Error"
    KnowledgeBase Name="Car diagnosis"
    ProcessingTime dd:hh:mm:ss:ms="0:0:0:0:30"
    Error
      UnreachableRules Quantity="1"
        UnreachableRule RuleID="R59"
    Warning
      UnreachableDiagnosis Quantity="1"
        UnreachableDiagnose DiagnosisID="P6"
```

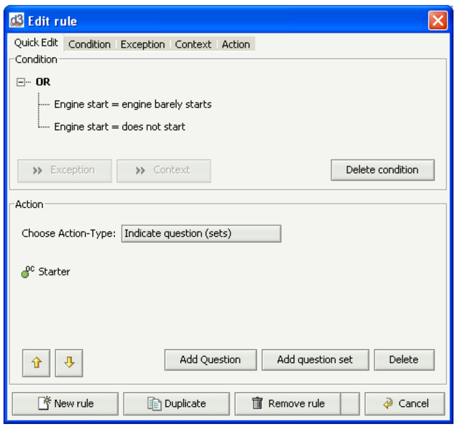
Empty battery: P7 (Heuristic) **IF**
TestDiagnosis = established

QualityChecks finished

Figure 8.21: d3web.QuaSiModu executing the integrity tests (static verification). The unreachable diagnosis P6=TestDiagnosis has been detected, which itself implies an unreachable rule with id R59.

- indicate questions or question sets
- contra-indicate questions or question sets
- clarify a diagnosis
- refine a diagnosis

The specified rule action distinguishes the different types of indication rules, that were introduced in Section 6.1 (p. 135). In Figure 8.22 an exemplary indication rule is shown. Questions or question sets are dragged from the question hierarchy into the action pane of


Edit rule

Condition

```
OR
  Engine start = engine barely starts
  Engine start = does not start
```

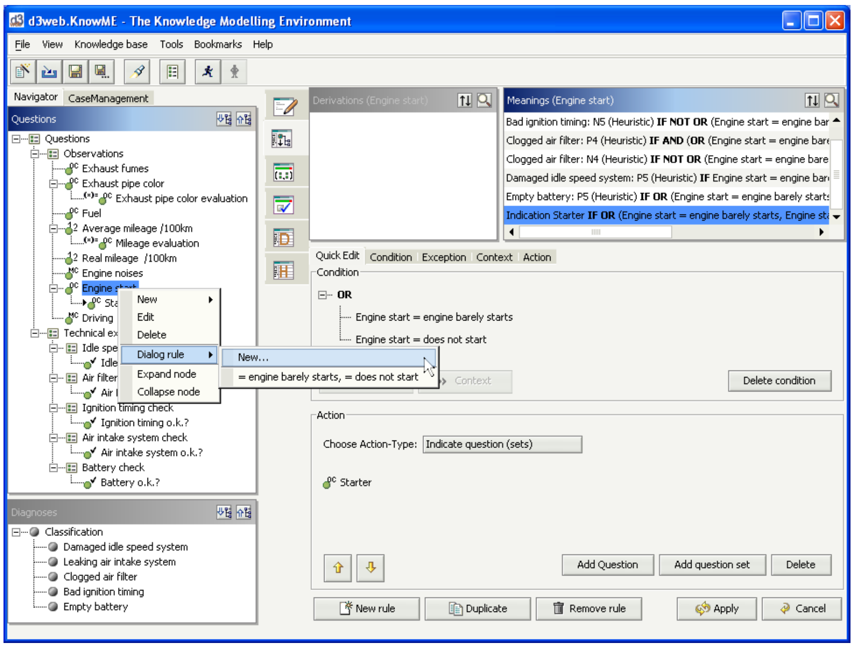
Action

Choose Action-Type: Indicate question (sets)

Starter

Figure 8.22: A rule editor window with an indication rule defining a follow-up question.

the rule editor in order to define them as the indication target. Indicated questions/question sets can be also selected by using the corresponding buttons in the rule action pane. The order of the selected questions/question sets can be manually modified by using the specified buttons ↑ and ↓. Furthermore, single questions/question sets can be deleted by using the context menu or the delete button. Follow-up questions usually are appended below the original question, and are frequently applied as strategic knowledge. In order to facilitate the simple creation of a follow-up indication, the context menu of the question hierarchy provides a menu entry for dialog rules. As shown in Figure 8.23, follow-up indications of



Bad ignition timing: N5 (Heuristic) **IF NOT OR** (Engine start = engine bar
Clogged air filter: P4 (Heuristic) **IF AND** (**OR** (Engine start = engine bare
Clogged air filter: N4 (Heuristic) **IF NOT OR** (Engine start = engine bare
Damaged idle speed system: P5 (Heuristic) **IF** Engine start = engine bare
Empty battery: P5 (Heuristic) **IF OR** (Engine start = engine barely starts
Indication Starter **IF OR** (Engine start = engine barely starts, Engine sta

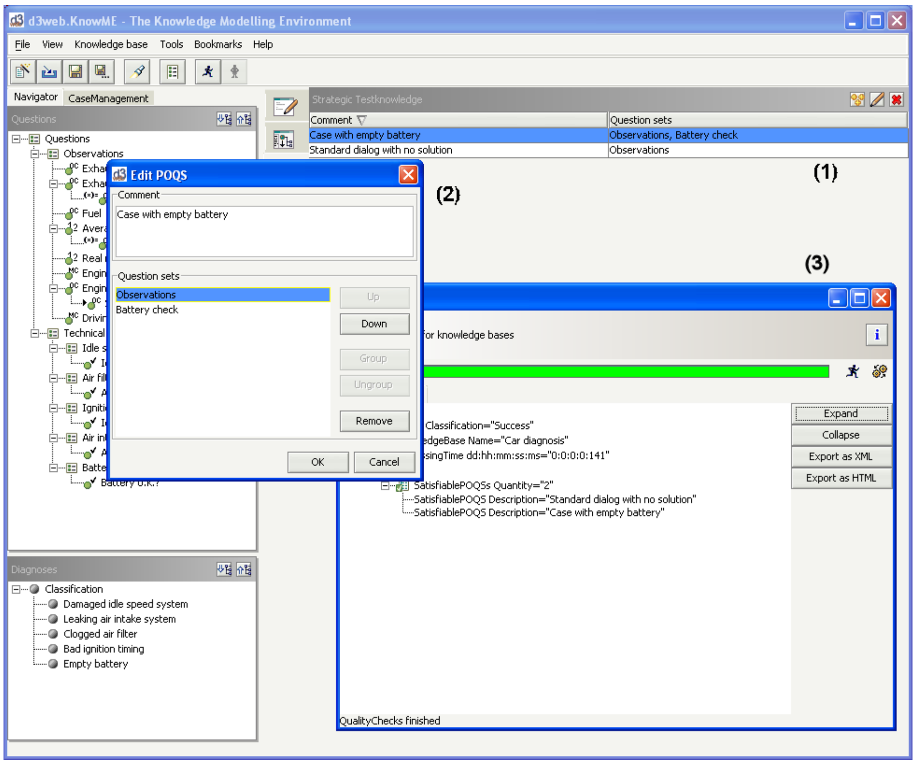
Condition
OR
Engine start = engine barely starts
Engine start = does not start

Figure 8.23: Indication rules for follow-up questions can be easily defined by using the context menu.

the selected questions can be quickly created or modified.

Testing Strategic Knowledge

Currently, d3web.QuaSiModu offers the test of partially ordered question sets, POQS, (p. 140). In Figure 8.24 the editor for defining POQS is shown: In (1) the basic editor gives an overview of all defined POQS, and new POQS can be defined or existing ones can be edited. The definition of a POQS is shown in (2): Question sets can be dragged from the question hierarchy into the *Question sets* pane. The order can be manually specified by using the *up* and *down* buttons. Partial orders can be described by grouping multiple question sets. Furthermore, each POQS is annotated with a comment, describing the specified dialog behavior. The application of the test knowledge is shown in (3): d3web.QuaSiModu



Comment	Question sets
Case with empty battery	Observations, Battery check
Standard dialog with no solution	Observations

Comment: Case with empty battery

Question sets:
Observations
Battery check

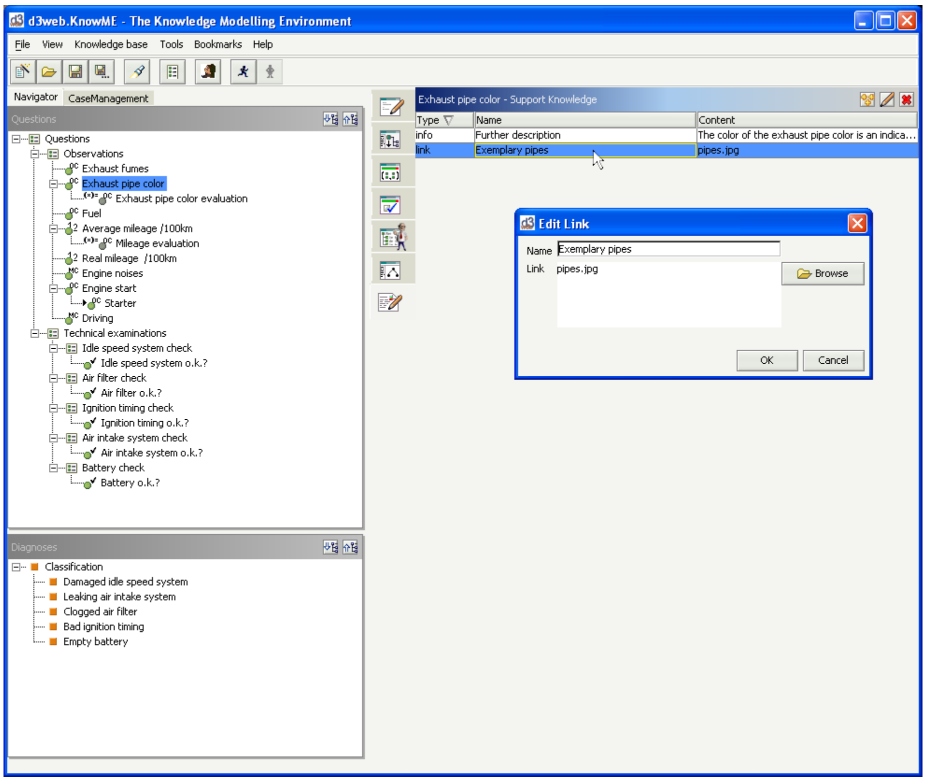
```
Classification="Success"
edgeBase Name="Car diagnosis"
singTime dd:hh:mm:ss:ms="0:0:0:0:141"
SatisfiablePOQSs Quantity="2"
    SatisfiablePOQS Description="Standard dialog with no solution"
    SatisfiablePOQS Description="Case with empty battery"
QualityChecks finished
```

Figure 8.24: The POQS editor for defining strategic test knowledge.

reports no errors, and the satisfiable POQS as info.

8.3.7. Handling Support Knowledge

Currently, the support knowledge editor of d3web.KnowME offers limited capabilities for the definition of support knowledge. The support knowledge editor is able to define explanatory texts, links to external resources (e.g., web links, PDF files), and extended texts for questions, question sets, and diagnoses. In Figure 8.25 the support knowledge



Type	Name	Content
info	Further description	The color of the exhaust pipe color is an indica...
link	Exemplary pipes	pipes.jpg

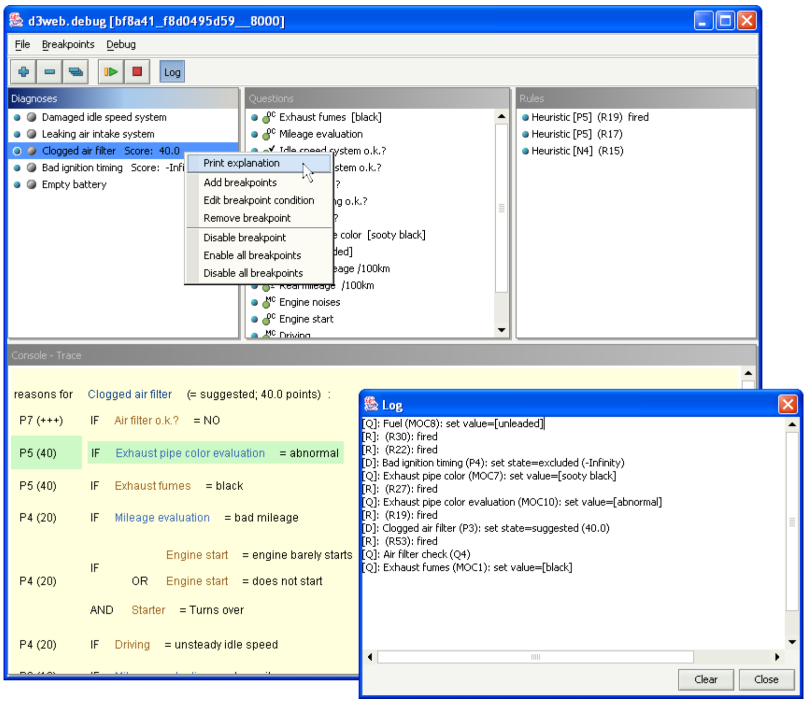
Name	Exemplary pipes
Link	pipes.jpg

Figure 8.25: The support knowledge editor of d3web.KnowME providing fields for defining an extended prompt, external links, and explanatory texts.

editor for the selected question ”Exhaust pipe color” is shown. No automated test method has been implemented for support knowledge.

8.3.8. Debugging Cases with d3web.KnowME

Another useful feature for the development of knowledge systems is the integration of a visual debugger for (test) cases. Thus, the behavior of the knowledge system can be manually inspected, e.g., in order to identify the reason for a detected error. The debugger can be invoked either via the empirical testing method of d3web.QuaSiModu (by clicking on a falsely solved test case), or by using the context menu of the case management editor as shown in Figure 8.18. Then, the case can be processed step-wise, and breakpoints for diagnoses, questions (sets), and knowledge elements can be defined. Currently, the debugger is able to define breakpoints on the following knowledge elements: Abstraction rules, categorical rules, score-based rules, and (contra-)indication rules. If an object is used while



Diagnoses:
- Damaged idle speed system
- Leaking air intake system
- Clogged air filter Score: 40.0
- Bad ignition timing Score: -Infi
- Empty battery

Questions:
- Exhaust fumes [black]
- Mileage evaluation
- Idle speed system o.k.?
- Engine noises
- Engine start
- Driving

Rules:
- Heuristic [P5] (R19) fired
- Heuristic [P5] (R17)
- Heuristic [N4] (R15)

```
reasons for   Clogged air filter   (= suggested; 40.0 points) :
P7 (+++)   IF   Air filter o.k.?   = NO
P5 (40)    IF   Exhaust pipe color evaluation   = abnormal
P5 (40)    IF   Exhaust fumes   = black
P4 (20)    IF   Mileage evaluation   = bad mileage
                       Engine start   = engine barely starts
           IF
P4 (20)          OR    Engine start   = does not start
           AND   Starter   = Turns over
P4 (20)    IF   Driving   = unsteady idle speed
```

```
[Q]: Fuel (MOC8): set value=[unleaded]
[R]:  (R30): fired
[R]:  (R22): fired
[D]: Bad ignition timing (P4): set state=excluded (-Infinity)
[Q]: Exhaust pipe color (MOC7): set value=[sooty black]
[R]:  (R27): fired
[Q]: Exhaust pipe color evaluation (MOC10): set value=[abnormal]
[R]:  (R19): fired
[D]: Clogged air filter (P3): set state=suggested (40.0)
[R]:  (R53): fired
[Q]: Air filter check (Q4)
[Q]: Exhaust fumes (MOC1): set value=[black]
```

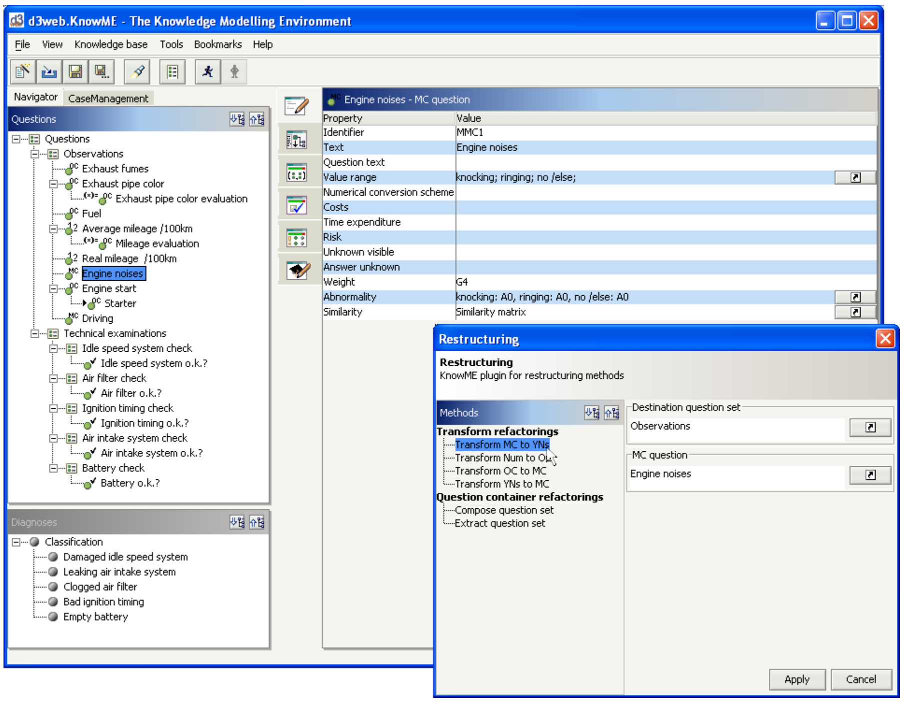
Figure 8.26: The visual debugger of d3web.KnowME. In the top panes, breakpoints for diagnoses, questions (sets), and rules can be defined. The bottom pane display a state trace of a specified object.

running the case, for which a breakpoint is defined, then the breakpoint is activated and the case is paused. At any time, the case can be introspected, i.e., currently applied knowledge (e.g., fired rules, values of questions, states of diagnoses) can be viewed for the defined objects. If the case is paused by an activated breakpoint, then a state trace can be displayed of any object defined as a breakpoint, i.e., the corresponding derivation rules are displayed attached with information about their state (e.g., fired). Figure 8.26 shows a screenshot of a debugging session. For a convenient use, the debugger allows for temporarily disabling breakpoints (i.e., watches on objects), for defining specific breakpoint conditions (additional conditions, that constrain the activation of breakpoints), and for sorting the breakpoints with respect to several factors (e.g., last use, name). Furthermore, the running case can be logged for a detailed analysis afterwards.

8.3.9. Automated Restructuring with d3web.KnowME

At the moment, d3web.KnowME provides support for the automated execution of six restructuring methods, e.g., for type transformations of questions and for the extraction/combination of question sets. In the example, shown in Figure 8.27, the multiple choice question ”Engine noises” with the value range

$$dom(Engine\ noises) = \{knocking, ringing, no/else\}$$



Engine noises - MC question

Property	Value
Identifier	MMC1
Text	Engine noises
Question text	
Value range	knocking; ringing; no /else;
Numerical conversion scheme	
Costs	
Time expenditure	
Risk	
Unknown visible	
Answer unknown	
Weight	G4
Abnormality	knocking: A0, ringing: A0, no /else: A0
Similarity	Similarity matrix

Restructuring
KnowME plugin for restructuring methods

Destination question set: Observations

MC question: Engine noises

Figure 8.27: The user interface of d3web.KnowME for the application of automated restructuring methods.

is converted into a set of three yes/no questions ”knocking”, ”ringing”, and ”no/else”. The yes/no questions are appended to the question set ”Observations”. The implementation and design goals of the restructuring methods of d3web.KnowME were described in [74] in more detail.

8.4. Comparison: D3 and d3web.KnowME

In this section, we briefly compare the knowledge modeling features of the shell-kit D3 with its successor d3web. We describe the features of both systems in a table (date of comparison was Jan. 2004).

Ontological Knowledge	D3	d3web.KnowME
Tree editors for diagnosis and question hierarchy	+	+
Structural Knowledge	D3	d3web.KnowME
Heuristic table	+	+
Heuristic decision table	+	+
Set-covering table	+	+
Case-based knowledge	+	+
Decision tree	+	−
Abstraction table	+	−
Strategic Knowledge	D3	d3web.KnowME
Rule editor	+	+
Indication table	+	−
Support Knowledge	D3	d3web.KnowME
Informal texts	+	+
Arbitrary links	+	+
Agile tools	D3	d3web.KnowME
Planning game editor	−	+
Visual debugger	limited/manual	+
Refactoring support	−	limited
Test methods	D3	d3web.KnowME
Static frequency analysis	limited	+
Dynamic frequency analysis	−	+
Test case analysis	limited/manual	+
Unit cases	−	+
Partial-Order Question Sets	−	+

8.5. Summary

In this chapter, we have given a brief overview of the systems D3 and d3web. The shell-kit D3 is the predecessor of the framework d3web, that both are used for the development and the application of diagnostic knowledge systems. The system family d3web consists of the highly integrated knowledge modeling environment d3web.KnowME and the web-based front-end for diagnostic knowledge systems d3web.Dialog. We have shown that the knowledge modeling environment d3web.KnowME offers editors for the agile development of diagnostic knowledge systems. The editors are used for the planning game, the construction of the particular knowledge containers, the definition and application of appropriate tests, the automated restructuring of knowledge, and the interactive debugging of knowledge systems. Furthermore, we presented the web-based dialog server d3web.Dialog for running diagnostic knowledge systems. The developed knowledge systems can be used over the internet via a web-browser as the user client.

9. Experiences

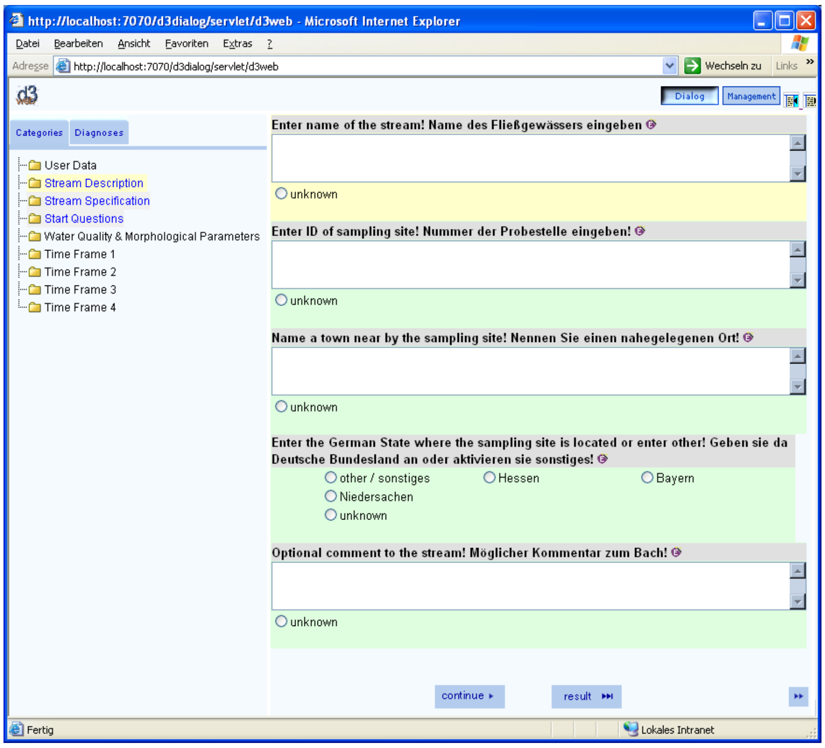
We describe two projects implemented in the context of this thesis. The LIMPACT project considers an area of the eco-toxilogical domain, and it was started before the presented process model has been introduced. However, some aspects of the methodology were also realized. The ECHODOC project delivers an application in the medical domain, and adopts many methods and practices presented in the context of the introduced agile process model.

9.1. The Eco-Toxilogical Knowledge System LIMPACT

The LIMPACT project (LIMnology and imPACT) was running between 2001 and 2003, and was joint work with Dr. Neumann at the Institute of Ecology, University of Braunschweig, Germany. The knowledge base and the development of the knowledge system LIMPACT was presented in [83, 80]. The system can be accessed by the web-site `http://www.limpact.de`. Figure 9.1 depicts a screenshot of the web-based user interface of LIMPACT. LIMPACT considers the estimation of pesticide contaminations of small lowland streams with agricultural catchment areas.

Small streams add up to an enormous length on the landscape level. Therefore, the conservation and protection of their aquatic community should be a major concern. In agriculturally used catchment areas these streams are subject to various stressors. For example, during heavy rainfall, runoff from agricultural fields may introduce soil, nutrients and pesticides, and increases the discharge [34, 82]. It has been shown that the impact of pesticides is an important parameter of influence for the aquatic fauna [67]. Unfortunately, no regular monitoring systems are established for these agricultural non-point sources of pesticides. Because of its short-term character, only rainfall event-controlled sampling methods can reflect such transient pesticide contamination, which makes its detection via chemical analysis costly.

For this reason, the use of a biological indicator system can provide a number of benefits. The main advantage is its easy and cost-efficient application. When used to monitor toxic contamination, it additionally indicates the ecotoxicological effect of the contaminant. It allows for a long-term information, whereas information of each chemical measurement applies at only one point in time. However, no biological indicator system has yet estimated the pesticide contamination of small streams via benthic macroinvertebrate indicators. See, e.g., [79] for a detailed discussion of related systems. To fill this gap, a consultation system was developed, that estimates the pesticide contamination of small streams based on biological indicators.



http://localhost:7070/d3dialog/servlet/d3web - Microsoft Internet Explorer

Datei Bearbeiten Ansicht Favoriten Extras ?

Adresse http://localhost:7070/d3dialog/servlet/d3web Wechseln zu Links

Dialog Management

Categories Diagnoses

- User Data
- Stream Description
- Stream Specification
- Start Questions
- Water Quality & Morphological Parameters
- Time Frame 1
- Time Frame 2
- Time Frame 3
- Time Frame 4

Enter name of the stream! Name des Fließgewässers eingeben

○ unknown

Enter ID of sampling site! Nummer der Probestelle eingeben!

○ unknown

Name a town near by the sampling site! Nennen Sie einen nahegelegenen Ort!

○ unknown

Enter the German State where the sampling site is located or enter other! Geben sie da Deutsche Bundesland an oder aktivieren sie sonstiges!

○ other / sonstiges ○ Hessen ○ Bayern
○ Niedersachen
○ unknown

Optional comment to the stream! Möglicher Kommentar zum Bach!

○ unknown

continue ▸ result ⏭

Fertig Lokales Intranet

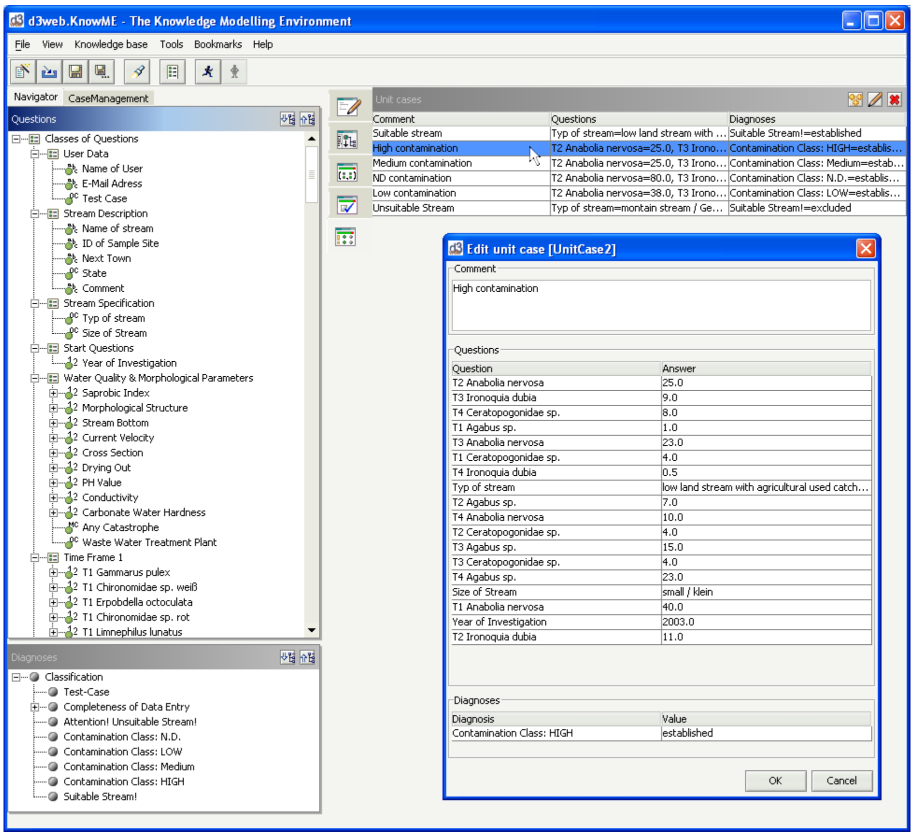
Figure 9.1: Screenshot of the web-based interface of the LIMPACT system.

9.1.1. Development Report

Using the global system metaphor the system was defined to be a *consultation system*, since LIMPACT should be able to determine the pesticide contamination of a small stream according to benthic macroinvertebrate indicators. The contamination of a stream was classified into four categories represented by the diagnoses *n.d.* (not detected), *low* (low contamination), *mod.* (moderate contamination), and *high* (high contamination). The knowledge base contains 13 questions describing the stream in more detail, e.g., structural parameters of the stream. Furthermore, the system contains questions representing the abundance of 39 different taxa. For each taxa the abundance can be recorded at 4 time-frames (spring, summer, autumn, winter). Besides the initial question set determining the suitability of the currently investigated stream, no dialog control is supporting the user. This procedure was motivated by the fact that in a typical case only a small portion of the represented taxa is actually observable. Therefore, the user has to manually select the taxa, for which abundance data should be entered.

When starting the LIMPACT project no agile process model has been defined, and therefore the development was not explicitly structured by planning games. However, the ontological knowledge and the scoring rules were incrementally implemented in the context of several planning phases, although not supported by appropriate tests and restructuring methods. In fact, (automated) tests were only performed after completing the rule-based implementation of the knowledge system. In the context of the agile process model it would have been desirable to define particular plans and tests considering the classification of the several contamination categories (represented as diagnoses). Besides the original rule-based knowledge an equivalent set-covering model was implemented. As reported in [79] the size and complexity of the implemented knowledge was significantly reduced using the set-covering representation. The set-covering model was developed according to the incremental process as described in [16]. Starting with simple set-covering relations, the model was incrementally refined by weights and exception conditions.

The structural knowledge was validated using unit cases (p. 129) and the empirical testing method (p. 125). Figure 9.2 depicts d3web.KnowME with the LIMPACT system defining unit cases. The definition of the unit cases revealed errors in the rule base. After a de-



Questions

- Classes of Questions
  - User Data
    - Name of User
    - E-Mail Adress
    - Test Case
  - Stream Description
    - Name of stream
    - ID of Sample Site
    - Next Town
    - State
    - Comment
  - Stream Specification
    - Typ of stream
    - Size of Stream
  - Start Questions
    - Year of Investigation
  - Water Quality & Morphological Parameters
    - Saprobic Index
    - Morphological Structure
    - Stream Bottom
    - Current Velocity
    - Cross Section
    - Drying Out
    - PH Value
    - Conductivity
    - Carbonate Water Hardness
    - Any Catastrophe
    - Waste Water Treatment Plant
  - Time Frame 1
    - T1 Gammarus pulex
    - T1 Chironomidae sp. weiß
    - T1 Erpobdella octoculata
    - T1 Chironomidae sp. rot
    - T1 Limnephilus lunatus

Diagnoses

- Classification
  - Test-Case
  - Completeness of Data Entry
  - Attention! Unsuitable Stream!
  - Contamination Class: N.D.
  - Contamination Class: LOW
  - Contamination Class: Medium
  - Contamination Class: HIGH
  - Suitable Stream!

Unit cases

Comment	Questions	Diagnoses
Suitable stream	Typ of stream=low land stream with ...	Suitable Stream!=established
High contamination	T2 Anabolia nervosa=25.0, T3 Irono...	Contamination Class: HIGH=establis...
Medium contamination	T2 Anabolia nervosa=25.0, T3 Irono...	Contamination Class: Medium=estab...
ND contamination	T2 Anabolia nervosa=80.0, T3 Irono...	Contamination Class: N.D.=establis...
Low contamination	T2 Anabolia nervosa=38.0, T3 Irono...	Contamination Class: LOW=establis...
Unsuitable Stream	Typ of stream=montain stream / Ge...	Suitable Stream!=excluded

Edit unit case [UnitCase2]

Comment

High contamination

Questions

Question	Answer
T2 Anabolia nervosa	25.0
T3 Ironoquia dubia	9.0
T4 Ceratopogonidae sp.	8.0
T1 Agabus sp.	1.0
T3 Anabolia nervosa	23.0
T1 Ceratopogonidae sp.	4.0
T4 Ironoquia dubia	0.5
Typ of stream	low land stream with agricultural used catch...
T2 Agabus sp.	7.0
T4 Anabolia nervosa	10.0
T2 Ceratopogonidae sp.	4.0
T3 Agabus sp.	15.0
T3 Ceratopogonidae sp.	4.0
T4 Agabus sp.	23.0
Size of Stream	small / klein
T1 Anabolia nervosa	40.0
Year of Investigation	2003.0
T2 Ironoquia dubia	11.0

Diagnoses

Diagnosis	Value
Contamination Class: HIGH	established

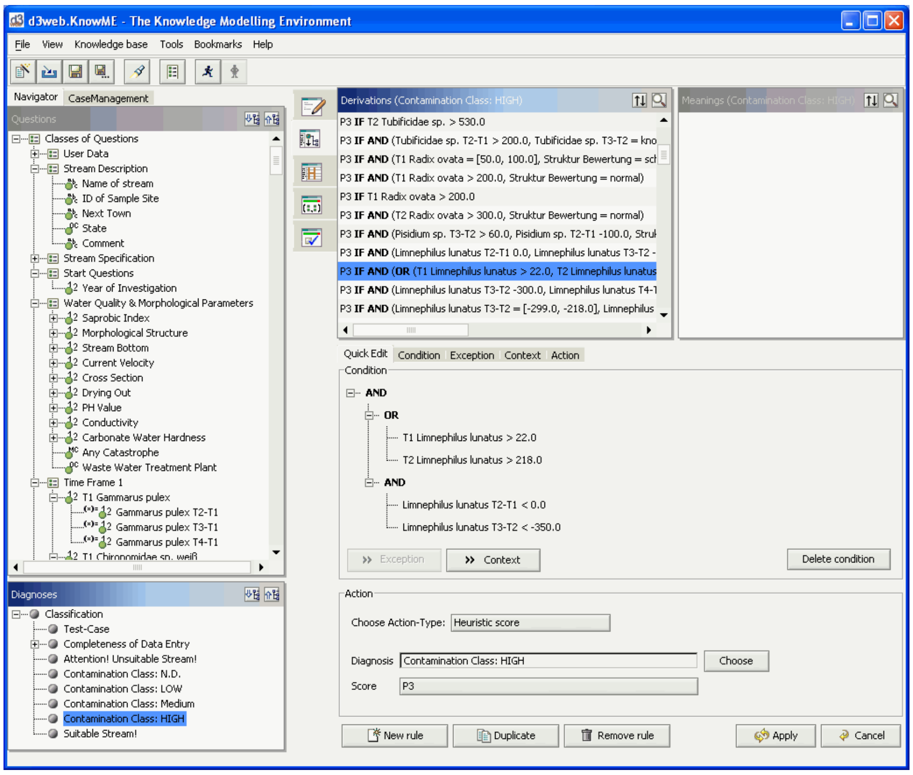
Figure 9.2: Screenshot of defining unit cases for the LIMPACT system.

bug session three falsely formalized rule conditions were identified containing a wrong numerical condition, i.e., *less* instead of *greater*. The case base used for empirical testing contained 146 test cases, gathered from investigations of 104 real streams during the years 1992 and 2000. Due to the chemical analysis of the streams the cases are attached with the correct contamination category. Figure 9.3 depicts the percentages of correct classifications for the rule-based and the set-covering approach. The first columns display the classification accuracy according to the specified contamination category, and the last column depicts the overall accuracy of both approaches. The number of cases contained in the case base with the specified contamination class is given in parentheses. We can see that

	n.d. (52)	low (30)	mod. (40)	high (24)	overall (146)
rule-based	90.4%	90.0%	72.5%	87.5%	84.9%
set-covering	96.2%	93.3%	87.5%	79.1%	90.4%

Figure 9.3: Accuracies measured for the rule-based and set-covering implementation of the LIMPACT system. The numbers in the parentheses are the number of cases available for each contamination class.

the rule-based version of LIMPACT has no high confidence level regarding streams with contamination the class *mod.*, whereas LIMPACT utilizing set-covering knowledge has no high confidence level for the diagnosis of highly contaminated streams. A screenshot



Classes of Questions
- User Data
- Stream Description
  - Name of stream
  - ID of Sample Site
  - Next Town
  - State
  - Comment
- Stream Specification
- Start Questions
  - Year of Investigation
- Water Quality & Morphological Parameters
  - Saprobic Index
  - Morphological Structure
  - Stream Bottom
  - Current Velocity
  - Cross Section
  - Drying Out
  - PH Value
  - Conductivity
  - Carbonate Water Hardness
  - Any Catastrophe
  - Waste Water Treatment Plant
- Time Frame 1
  - T1 Gammarus pulex
    - Gammarus pulex T2-T1
    - Gammarus pulex T3-T1
    - Gammarus pulex T4-T1
  - T1 Chironomidae sp. weiß

Classification
- Test-Case
- Completeness of Data Entry
- Attention! Unsuitable Stream!
- Contamination Class: N.D.
- Contamination Class: LOW
- Contamination Class: Medium
- Contamination Class: HIGH
- Suitable Stream!

P3 **IF** T2 Tubificidae sp. > 530.0
P3 **IF AND** (Tubificidae sp. T2-T1 > 200.0, Tubificidae sp. T3-T2 = kno
P3 **IF AND** (T1 Radix ovata = [50.0, 100.0], Struktur Bewertung = sch
P3 **IF AND** (T1 Radix ovata > 200.0, Struktur Bewertung = normal)
P3 **IF** T1 Radix ovata > 200.0
P3 **IF AND** (T2 Radix ovata > 300.0, Struktur Bewertung = normal)
P3 **IF AND** (Pisidium sp. T3-T2 > 60.0, Pisidium sp. T2-T1 -100.0, Strul
P3 **IF AND** (Limnephilus lunatus T2-T1 0.0, Limnephilus lunatus T3-T2 -
P3 **IF AND** (**OR** (T1 Limnephilus lunatus > 22.0, T2 Limnephilus lunatus
P3 **IF AND** (Limnephilus lunatus T3-T2 -300.0, Limnephilus lunatus T4-T
P3 **IF AND** (Limnephilus lunatus T3-T2 = [-299.0, -218.0], Limnephilus

Condition
- **AND**
  - **OR**
    - T1 Limnephilus lunatus > 22.0
    - T2 Limnephilus lunatus > 218.0
  - **AND**
    - Limnephilus lunatus T2-T1 < 0.0
    - Limnephilus lunatus T3-T2 < -350.0

Action

Choose Action-Type: Heuristic score

Diagnosis: Contamination Class: HIGH

Score: P3

Figure 9.4: Screenshot of the LIMPACT knowledge base in the modeling environment d3web.KnowME.

of the knowledge modeling environment d3web.KnowME with the LIMPACT knowledge base is depicted in Figure 9.4.

The rule-based implementation contained 959 diagnostic rules with 450 simple rules, 303 one-level rules, and 206 multiple-level rules. The set-covering approach was implemented by defining 816 simple set-covering relations. A comparison of the rule-based and the set-covering knowledge bases shows that the number of set-covering relations is only slightly smaller than the number of implemented rules in the rule-based system. Nevertheless, the modeled set-covering knowledge is less complex than the implemented rules by an order-of-magnitude. When adding rules for taxa to the rule-base, we also have to consider the associated confirmation categories. These categories interact with other rules when aggregating to the same diagnosis score, and therefore have to be obtained by a thorough analysis. Thus, adding a new rule to the rule base can demand the reconsideration of all rules (or of the associated scores) deriving the same diagnosis. In contrast to these interwoven rules, set-covering relations can be viewed as self-contained knowledge elements without mutual interdependencies. For a new taxon we only have to define relevant set-covering relations between the four diagnoses, i.e., contamination classes, and the new taxon. In general, this means that we have to define the abundance of the new taxon for each diagnosis, if we expect the taxon to occur with the given diagnosis.

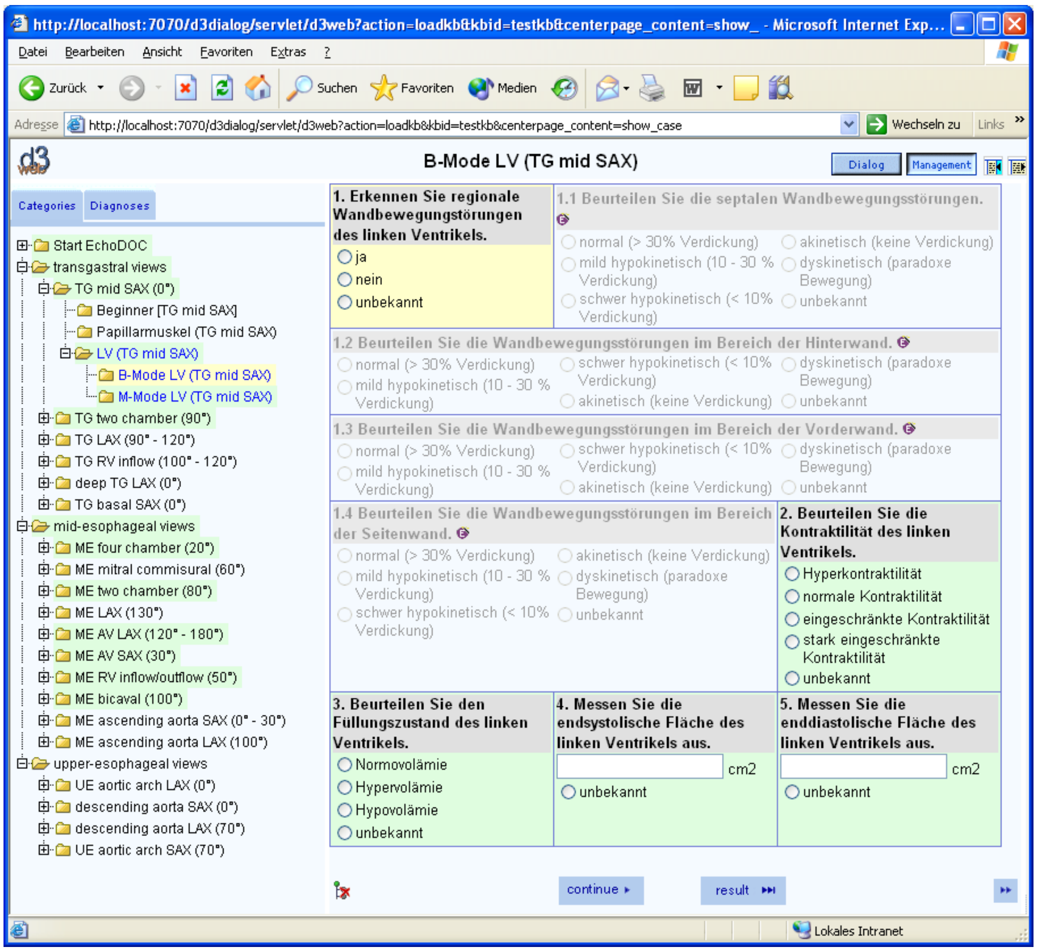
9.1.2. Conclusion

With the LIMPACT project the research on agile process models for developing knowledge systems was inspired. The agile process model was not fully applied during the development; some techniques like the incremental development of set-covering models and the use of automated tests have been applied during the construction of LIMPACT.

After the project was finished a concluding analysis showed that an adapted scoring approach would have probably yield a more compact rule base. The LIMPACT system basically tries to determine the pollution of a small stream, which itself is implemented by four diagnoses representing symbolic pollution categories in ascending order. For each diagnosis often very similar scoring rules were implemented simply differing in their confirmation category. A more compact approach would have considered the definition of a common score for all diagnoses, which is used by all scoring rules. The derivation of the actual diagnosis then depends on the aggregated value of the common score, e.g., low aggregates infer a low pollution level.

9.2. Medical TEE-Examinations with EchoDOC

The EchoDOC project (formerly QualiTEE) was started to support physicians during transesophageal echocardiography (TEE) examinations [68]. The intraoperative monitoring of critical ill patients is a permanent challenge in modern anaesthesia and the use of TEE is a well established method in monitoring these patients. But besides the understanding of the technical aspects the results of a TEE examination can be improved, if the examination is done in a standardized and efficient way. Therefore, the major goal of the project was the implementation of a standardized documentation system. In Figure 9.5 a sample dialog of the EchoDOC system is depicted.


http://localhost:7070/d3dialog/servlet/d3web?action=loadkb&kbid=testkb&centerpage_content=show_ - Microsoft Internet Exp...

Datei Bearbeiten Ansicht Favoriten Extras ?

Zurück Suchen Favoriten Medien

Adresse http://localhost:7070/d3dialog/servlet/d3web?action=loadkb&kbid=testkb&centerpage_content=show_case Wechseln zu Links

B-Mode LV (TG mid SAX)

Dialog Management

Categories Diagnoses

- Start EchoDOC
- transgastral views
  - TG mid SAX (0°)
    - Beginner [TG mid SAX]
    - Papillarmuskel (TG mid SAX)
    - LV (TG mid SAX)
      - B-Mode LV (TG mid SAX)
      - M-Mode LV (TG mid SAX)
  - TG two chamber (90°)
  - TG LAX (90° - 120°)
  - TG RV inflow (100° - 120°)
  - deep TG LAX (0°)
  - TG basal SAX (0°)
- mid-esophageal views
  - ME four chamber (20°)
  - ME mitral commisural (60°)
  - ME two chamber (80°)
  - ME LAX (130°)
  - ME AV LAX (120° - 180°)
  - ME AV SAX (30°)
  - ME RV inflow/outflow (50°)
  - ME bicaval (100°)
  - ME ascending aorta SAX (0° - 30°)
  - ME ascending aorta LAX (100°)
- upper-esophageal views
  - UE aortic arch LAX (0°)
  - descending aorta SAX (0°)
  - descending aorta LAX (70°)
  - UE aortic arch SAX (70°)

1. Erkennen Sie regionale Wandbewegungstörungen des linken Ventrikels.
○ ja
○ nein
○ unbekannt

1.1 Beurteilen Sie die septalen Wandbewegungsstörungen.
○ normal (> 30% Verdickung) ○ akinetisch (keine Verdickung)
○ mild hypokinetisch (10 - 30 % Verdickung) ○ dyskinetisch (paradoxe Bewegung)
○ schwer hypokinetisch (< 10% Verdickung) ○ unbekannt

1.2 Beurteilen Sie die Wandbewegungsstörungen im Bereich der Hinterwand.
○ normal (> 30% Verdickung) ○ schwer hypokinetisch (< 10% Verdickung) ○ dyskinetisch (paradoxe Bewegung)
○ mild hypokinetisch (10 - 30 % Verdickung) ○ akinetisch (keine Verdickung) ○ unbekannt

1.3 Beurteilen Sie die Wandbewegungsstörungen im Bereich der Vorderwand.
○ normal (> 30% Verdickung) ○ schwer hypokinetisch (< 10% Verdickung) ○ dyskinetisch (paradoxe Bewegung)
○ mild hypokinetisch (10 - 30 % Verdickung) ○ akinetisch (keine Verdickung) ○ unbekannt

1.4 Beurteilen Sie die Wandbewegungsstörungen im Bereich der Seitenwand.
○ normal (> 30% Verdickung) ○ akinetisch (keine Verdickung)
○ mild hypokinetisch (10 - 30 % Verdickung) ○ dyskinetisch (paradoxe Bewegung)
○ schwer hypokinetisch (< 10% Verdickung) ○ unbekannt

2. Beurteilen Sie die Kontraktilität des linken Ventrikels.
○ Hyperkontraktilität
○ normale Kontraktilität
○ eingeschränkte Kontraktilität
○ stark eingeschränkte Kontraktilität
○ unbekannt

3. Beurteilen Sie den Füllungszustand des linken Ventrikels.
○ Normovolämie
○ Hypervolämie
○ Hypovolämie
○ unbekannt

4. Messen Sie die endsystolische Fläche des linken Ventrikels aus.
cm2
○ unbekannt

5. Messen Sie die enddiastolische Fläche des linken Ventrikels aus.
cm2
○ unbekannt

continue ▸ result ⏭

Lokales Intranet

Figure 9.5: Screenshot of the web-based interface of the EchoDOC system.

The project is a cooperation with the Department of Anaesthesiology of the University of Würzburg Hospitals. The knowledge base was mainly implemented by the physician Dr. Lorenz, and reviewed by the recognized expert Prof. Greim.

The development of the system was preceded by a global analysis of the TEE examination process. Since no global standard for the overall examination has been established the dialog was designed according to the local examination process defined by Prof. Greim. This examination process is applied by the physicians working at the University of Würzburg Hospitals, and is regularly taught in specialized TEE courses. In general, the overall examination is partitioned into an ordered sequence of examination blocks. Each block describes a special aspect of the examination, i.e., a partial stage. The documentation of these partial stages is catched up by specified questions grouped by question sets. The defined order and chosen abstraction level of the stages follow standardized suggestions given in medical textbooks, e.g., Sidebotham et al. [122]. Given the fully defined examination process quick sets were specified in a next phase. Quick sets are a subset of the full examination and consider specialized problems faced during a TEE examination. Different quick sets are applied for different medical departments. While the full examination is appropriate for the physicians with no or low experience, quick sets can be used by more experienced physicians. However, ECHODOC also offers a completely free documentation of a TEE examination for expert users.

The ECHODOC project is the first adopter of the presented agile process model. Thus, we briefly describe the application of the process model in the following.

9.2.1. The Development Cycle

The cyclic development process consists of the definition of the system metaphor, the planning game, the implementation phase, and the integration phase.

The System Metaphor During the initial requirements analysis, the system metaphor was identified. According to a white paper the most important aspect of the intended system was a standardized documentation of the TEE examination. For this reason, the global metaphor *documentation system* was chosen. In the future, an extension to a consultation system is planned but not yet realized. Since the experts had previous experiences with D3, they were already familiar with the local system metaphor, i.e., the common naming conventions.

Planning Game Actually, the planning game was divided into two main phases: In the first phase the overall examination was implemented according to a structured and standardized process. Since a full examination according to the standardized process is very time-consuming, and in most cases not necessary, quicksets for specialized problems were defined in the second phase.

The first phase was initiated by an informal analysis of the TEE examination. The experts firstly discussed the several phases of the examination, which were then partitioned into *examination blocks*. In Figure 9.6 the original draft of the examination is depicted.

Each block describes an aspect of the examination and was easily transferred to a separate story. Therefore, the experts used the exam draft as a *story board* during the implementation, and successfully defined stories for the particular blocks. For each story, i.e., partial examination, the current (manual) acquisition of the corresponding findings was investigated, e.g., the actual set of gathered findings was defined. Then, acquisition standards given by medical textbooks were considered, the expected experience of the users for this

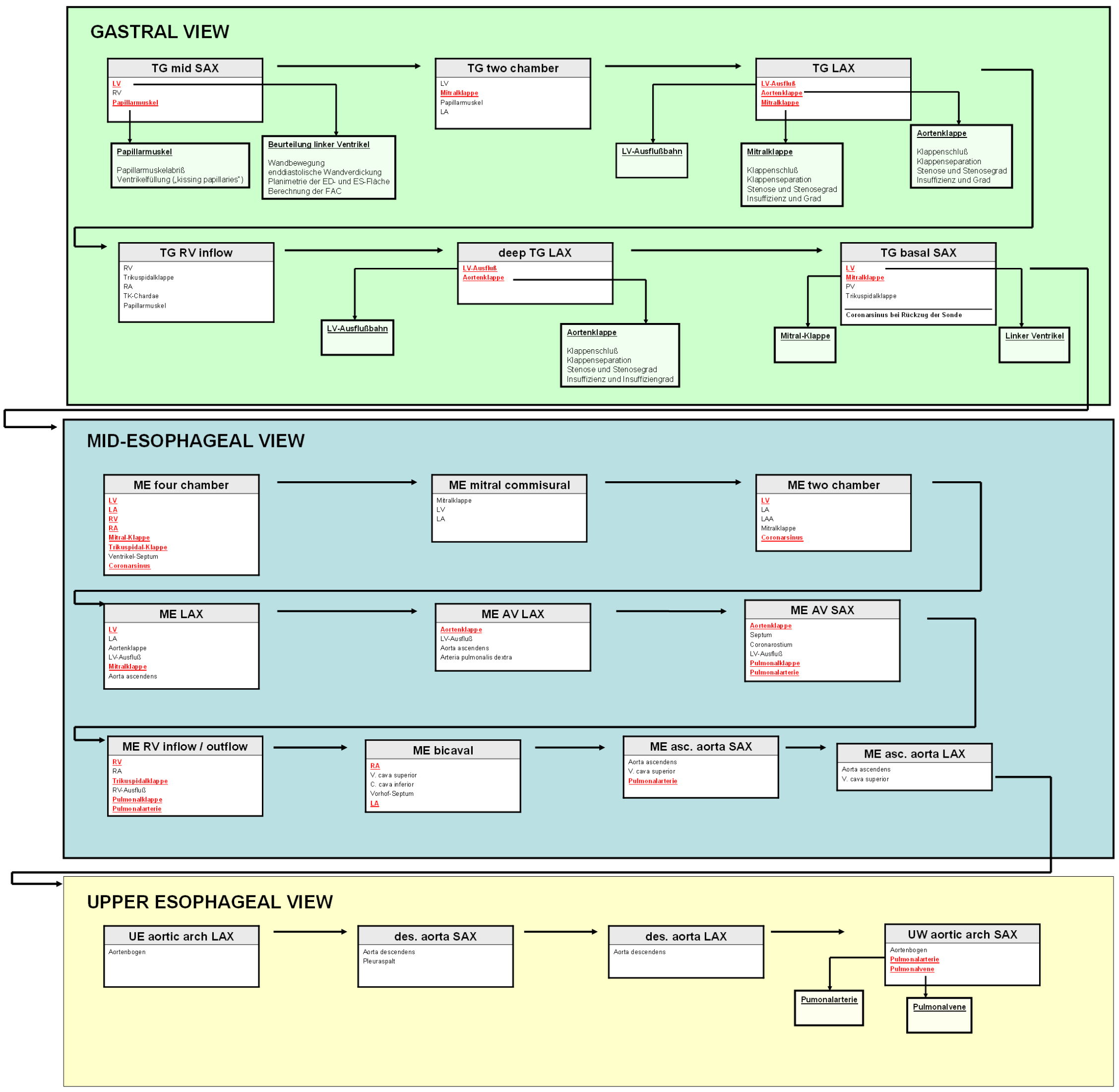


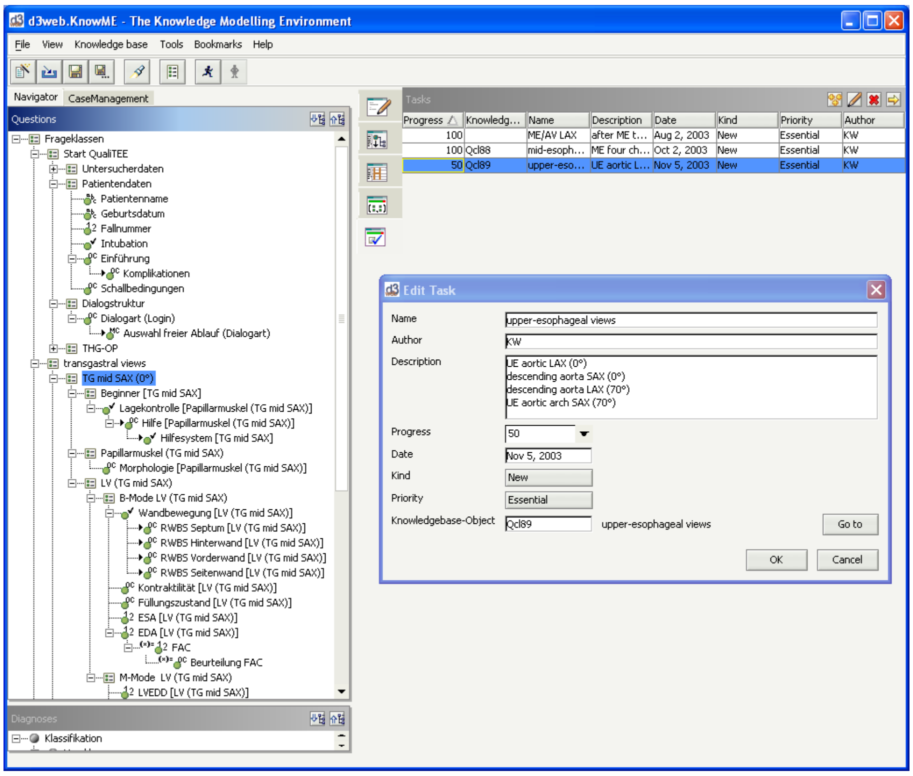
Figure 9.6: The draft of a TEE examination.

examination, as well as the required time needed for a manual acquisition of the examination. Based on these investigations the story was refined by a precise definition of applied questions and question sets.

As noted in Section 4.3 (p. 49), an important issue is the level of detail of the gathered questions. Then, the level of precision and the expected previous knowledge need to be considered. Users of the ECHODOC system are assumed to be experienced with sonography examinations, i.e., to understand and to interpret questioned findings in a technical terminology (e.g., *kissing papillaries*). However, in a later milestone of the project, it is planned to attach additional support knowledge in order to explain the used terminology. A user of the system is categorized as a *beginner*, if not familiar with the exact process of the standardized TEE examination.

Not all stories could be directly transferred from the story board. Some stories were removed because they appeared to be useless (e.g., the acquisition of unused exams), and some stories were included afterwards (e.g., beginner questions). Since the domain expert is only working part-time on the project, the use of stories in conjunction with the planning game was very advantageous. Then, the collection of stories attached with progress information was helpful when continuing the project after development breaks.

Additionally, the integration of the task editor into the modeling workbench d3web.KnowME was appreciated by the experts. Figure 9.7 depicts the task editor of the workbench d3web.KnowME. The task editor is used for inserting and editing stories gathered by the planning game. Before the integration of the task editor, the stories were



```
Frageklassen
  Start QualiTEE
    Untersucherdaten
    Patientendaten
      Patientenname
      Geburtsdatum
      Fallnummer
      Intubation
      Einführung
        Komplikationen
      Schallbedingungen
    Dialogstruktur
      Dialogart (Login)
        Auswahl freier Ablauf (Dialogart)
    THG-OP
  transgastral views
    TG mid SAX (0°)
      Beginner [TG mid SAX]
        Lagekontrolle [Papillarmuskel (TG mid SAX)]
          Hilfe [Papillarmuskel (TG mid SAX)]
            Hilfesystem [TG mid SAX]
      Papillarmuskel (TG mid SAX)
        Morphologie [Papillarmuskel (TG mid SAX)]
      LV (TG mid SAX)
        B-Mode LV (TG mid SAX)
          Wandbewegung [LV (TG mid SAX)]
            RWBS Septum [LV (TG mid SAX)]
            RWBS Hinterwand [LV (TG mid SAX)]
            RWBS Vorderwand [LV (TG mid SAX)]
            RWBS Seitenwand [LV (TG mid SAX)]
          Kontraktilität [LV (TG mid SAX)]
          Füllungszustand [LV (TG mid SAX)]
          ESA [LV (TG mid SAX)]
          EDA [LV (TG mid SAX)]
            FAC
              Beurteilung FAC
        M-Mode  LV (TG mid SAX)
          LVEDD [LV (TG mid SAX)]

Klassifikation
```

Progress	Knowledg...	Name	Description	Date	Kind	Priority	Author
100		ME/AV LAX	after ME t...	Aug 2, 2003	New	Essential	KW
100	Qcl88	mid-esoph...	ME four ch...	Oct 2, 2003	New	Essential	KW
50	Qcl89	upper-eso...	UE aortic L...	Nov 5, 2003	New	Essential	KW

Name	upper-esophageal views
Author	KW
Description	UE aortic LAX (0°) descending aorta SAX (0°) descending aorta LAX (70°) UE aortic arch SAX (70°)
Progress	50
Date	Nov 5, 2003
Kind	New
Priority	Essential
Knowledgebase-Object	Qcl89 upper-esophageal views

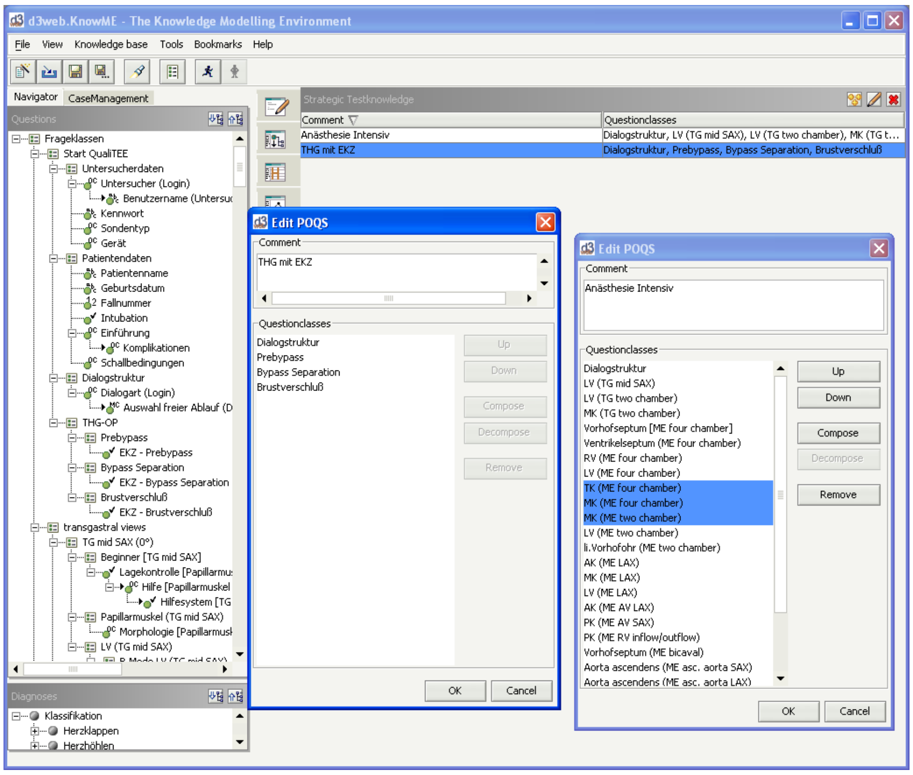
Figure 9.7: The task editor of the workbench d3web.KnowME.

separately recorded using a text processing program. This of course yields the undesirable property of a separate storage of knowledge base and corresponding story documentation.

In the second phase, the planning game mainly considered the improvement of the system by the introduction of quick sets. Quicksets are defined according to specialized problems faced during practical TEE examinations. Then, the examination does not cover all but a set of specialized aspects of the TEE. These quicksets are defined according to local standards of the particular departments the system is applied, but also considers textbook knowledge [122, 75].

Implementation Since the project firstly focussed on the development of a documentation system, only the ontological and strategic knowledge containers were considered.

Based on a sequence of stories the ontological knowledge container was successfully extended by questions, grouped and structured by question sets describing the particular examination blocks. For example, at the left of Figure 9.7 a part of the question hierarchy is depicted. Strategic knowledge was implemented using indication rules (p. 135). In context of the ECHODOC project the POQS test method (p. 140) was introduced in order to provide an appropriate test for documentation systems. Only a small number of POQS were defined, one POQS for the guided dialog and one for each quick set. The POQS were subsequently adapted due to the implementation of new stories. Figure 9.8 depicts the POQS editor of the workbench d3web.KnowME, which facilitates the visual acquisition of POQS test knowledge.



```
Frageklassen
  Start QualiTEE
    Untersucherdaten
      Untersucher (Login)
        Benutzername (Untersuc
      Kennwort
      Sondentyp
      Gerät
    Patientendaten
      Patientenname
      Geburtsdatum
      Fallnummer
      Intubation
      Einführung
        Komplikationen
      Schallbedingungen
    Dialogstruktur
      Dialogart (Login)
        Auswahl freier Ablauf (D
    THG-OP
      Prebypass
        EKZ - Prebypass
      Bypass Separation
        EKZ - Bypass Separation
      Brustverschluß
        EKZ - Brustverschluß
  transgastral views
    TG mid SAX (0°)
      Beginner [TG mid SAX]
        Lagekontrolle [Papillarmu:
          Hilfe [Papillarmuskel
            Hilfesystem [TG
      Papillarmuskel (TG mid SAX)
        Morphologie [Papillarmusk
      LV (TG mid SAX)
```

```
Klassifikation
  Herzklappen
  Herzhöhlen
```

Comment	Questionclasses
Anästhesie Intensiv	Dialogstruktur, LV (TG mid SAX), LV (TG two chamber), MK (TG t…
THG mit EKZ	Dialogstruktur, Prebypass, Bypass Separation, Brustverschluß

Comment: THG mit EKZ

Questionclasses:
Dialogstruktur
Prebypass
Bypass Separation
Brustverschluß

Comment: Anästhesie Intensiv

Questionclasses:
Dialogstruktur
LV (TG mid SAX)
LV (TG two chamber)
MK (TG two chamber)
Vorhofseptum [ME four chamber]
Ventrikelseptum (ME four chamber)
RV (ME four chamber)
LV (ME four chamber)
TK (ME four chamber)
MK (ME four chamber)
MK (ME two chamber)
LV (ME two chamber)
li.Vorhofohr (ME two chamber)
AK (ME LAX)
MK (ME LAX)
LV (ME LAX)
AK (ME AV LAX)
PK (ME AV SAX)
PK (ME RV inflow/outflow)
Vorhofseptum (ME bicaval)
Aorta ascendens (ME asc. aorta SAX)
Aorta ascendens (ME asc. aorta LAX)

Figure 9.8: The POQS editor of the workbench d3web.KnowME.

In addition of the automatic test method, the system was undergoing a repeated manual inspection by running exemplary examination dialogs. This visual and interactive evaluation of the implementation was especially important during the reviews of the system together with the second domain specialist.

Integration In particular, the continuous integration was very convincing for the success of the development project, because the developing expert always could provide a running system for review. Since the reviewers had a view of the current system at any time, the already implemented functionality could be tested under "real-life" conditions. This in turn was very helpful when deciding about further plans.

9.2.2. Application of Agile Practices

In Chapter 2 we have identified *testing*, *restructuring*, and *learning* as the key practices of the agile process model. During the implementation of the ECHODOC project we only applied testing and restructuring. Learning was not considered since we could not provide appropriate methods for learning strategic knowledge.

Restructuring

In the finished phases of the project restructuring was mainly applied for ontological knowledge considering question sets. The automatic restructuring of related strategic knowledge appeared to be very useful, because it reduced the complexity of the single operations.

Restructuring during the ECHODOC project was motivated by the introduction of quick sets, that were defined in a later phase of the project in order to facilitate an efficient and focussed dialog. Thus, sometimes not all questions contained in a question set were required to cover the aspect of a defined quickset. Therefore, original question sets were extracted into a set of question sets in order to facilitate a modular use of the individual question sets.

In Figure 9.9 the restructuring tool of the workbench d3web.KnowME is depicted, which, e.g., offers the combination and extraction of question sets.

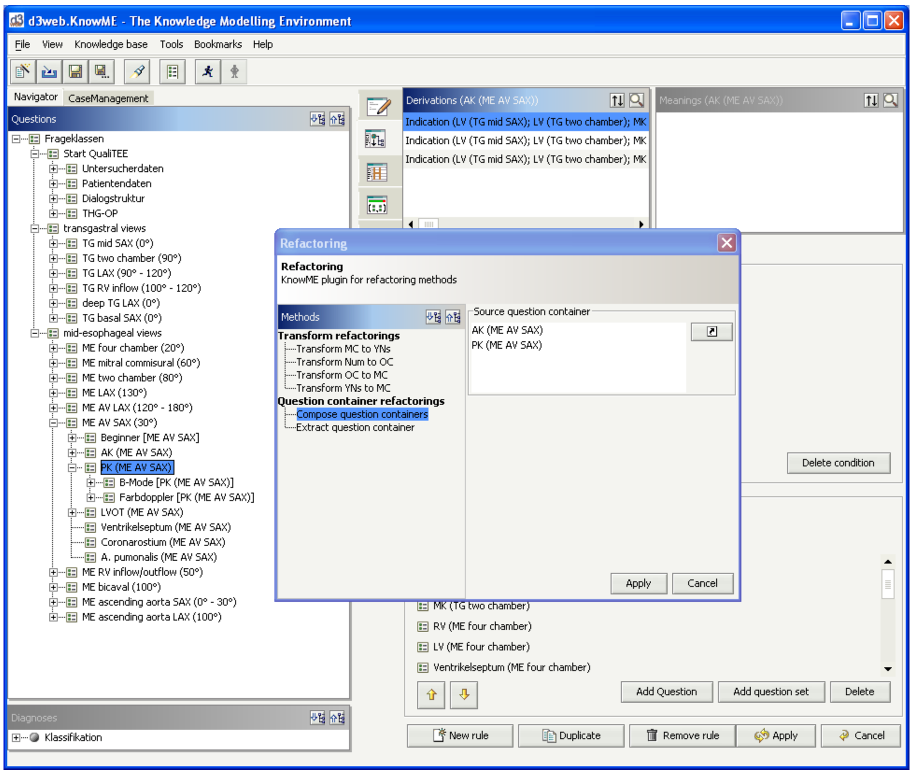


Figure 9.9: The restructuring tool of the workbench d3web.KnowME.

Testing

Due to the documentation system metaphor the developer mainly focussed the testing of strategic knowledge. For testing strategic knowledge we consider the correctness and the efficiency of the dialog. The correctness of the strategic knowledge is evaluated using the POQS method. We have already discussed the application of the POQS method on page 189. Furthermore, we plan to evaluate the efficiency of the dialog using the test case duration method. A pre-study has been undertaken to define a benchmark value, i.e., determining the typical dialog duration in the current examination setting (without EchoDOC).

Application of the test case duration method The acceptance of the system is mainly defined by the typical duration of an examination. Therefore, a pre-study was performed by the physician Dr. Lorenz in order to determine the usual duration for a manual examination without the intended system. Thus, at two departments of the medical faculty the physicians performing TEE examinations answered a questionary for about two months. During this study 37 examinations were documented, with 8 physicians involved. Basically, the questionary acquires the experience of the performing physician, the conditions of the examination, the pathological findings, the duration of the examination, and the time for documenting the examination.

Figure 9.10 depicts the results of this pre-study, presenting the required time in minutes (with standard deviation) for the examination and the documentation of the examination separately. The first block of the diagram containing four columns depicts the averaged duration of the examination and its corresponding documentation before (vEKZ) and after (nEKZ) the application of the extracorporal circulation. The following three blocks state these values more precisely with respect to the position of the physician (ASS=resident, FA=anaesthesia specialist, OA=attending physician). The last three blocks depict the duration of the examination and documentation with respect to the experience of the physician (Anf=beginner, Fortg.=advanced, Experte=expert). Separate values for ”Documentation nEKZ FA/OK/Anf” were not available in the context of this study; they are contained in the corresponding values ”Examination nEKZ FA/OK/Anf”.

In summary, the average duration in minutes of a TEE examination was $37,57 \pm 16,84$ before, and $37,12 \pm 22,07$ after extracorporal circulation. However, the average duration in minutes of the (manual) documentation is even more interesting in the context of our pre-study, which was $7,54 \pm 4,53$ before, and $2,00 \pm 1,20$ after extracorporal circulation. Due to the limited number of samples an even more detailed analysis with respect to the position and the experience of the physician is not reasonable.

The duration of documentation before extracorporal circulation is used as the benchmark for the evaluation of test case duration method.

Currently, a real-life evaluation of the EchoDOC system is planned and scheduled at one medical department of the University of Würzburg Hospitals.

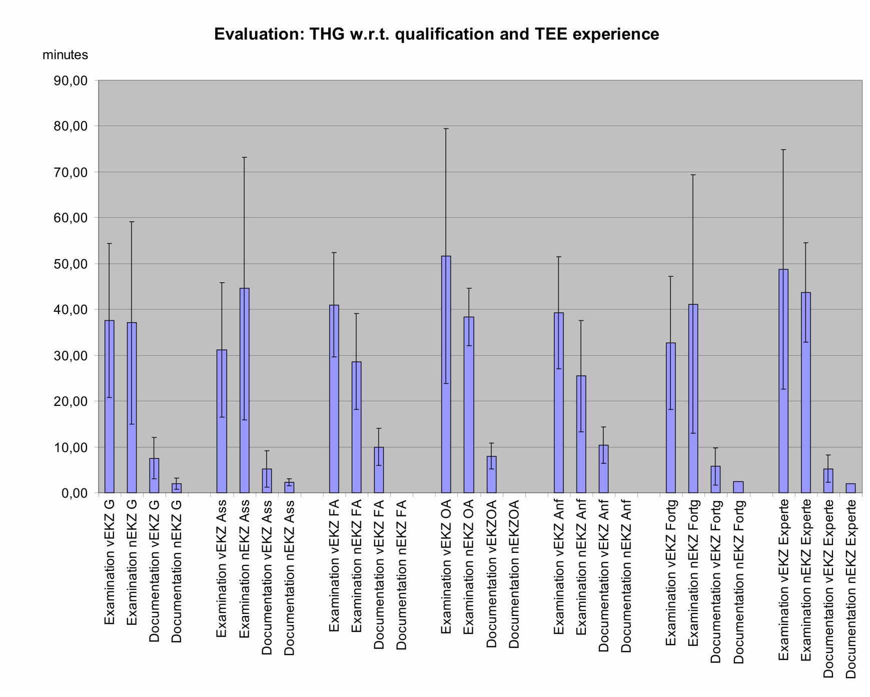


Figure 9.10: Results of the pre-study for determining the required time in minutes for a TEE examination.

9.2.3. Conclusion

For the development of the ECHODOC system the agile process model was applied for the first time. Although currently only ontological and strategic knowledge has been implemented we have made promising experiences with the application of the development process, i.e., the planning game, the implementation, and the integration. Furthermore, some agile practices showed to be very useful during implementation, especially the automated restructuring of question sets, and the application of testing methods, e.g., the POQS method.

9.3. Summary

In this chapter, we have described two diagnostic knowledge systems, that were developed in the context of this thesis.

The eco-toxilogical LIMPACT system was build before the presented agile process model was introduced, but some agile techniques like automated testing and the incremental modeling have been already applied.

The medical documentation system ECHODOC was the second project described in this chapter, and many agile practices have been used in the context of this project. For example, the planning game appeared to be very suitable for small project teams and the part-time development.

Part IV.

Conclusion

10. Summary

The development of knowledge systems is still a complex and error-prone task. Although a lot of research has been done in the last decades, the *perfect* process model for all types of knowledge system projects has not been found, so far. In this work, we introduced a novel process model that allows for an agile construction and maintenance of diagnostic knowledge systems. The presented process model narrows on the development of diagnostic systems in order to provide a suitable environment for the domain specialist self-formalizing and maintaining the required knowledge. Usually, the process model is suitable for small teams of about 1-3 people.

10.1. The Agile Process Model

The presented process model was inspired by the popular methodology eXtreme programming known in software engineering research. The development process was structured by the agile phases: Definition of the system metaphor, the planning game, the implementation phase, and the integration phase.

The system metaphor defines a common system of names for the knowledge system project. With the system metaphor, the semantics of, e.g., a diagnosis, a question, a question set, and a case are defined. The remaining phases are repeatedly traversed in a cyclic manner. The planning game typically considers the short-term scope and requirements of the running project. It is designed to provide an early and concrete feedback, a flexible schedule of the development process, and it lasts as long as the system lasts. The implementation phase consists of a test-implementation and a code-implementation: In principle, the coding of new knowledge or the restructuring of existing knowledge is always preceded by the coding of appropriate test knowledge. The integration phase guarantees an always running system by continuously integrating the new and modified knowledge into a production version of the knowledge system. The production version of the system is always validated using additional integration tests, extensive tests that are usually time-consuming, and that are covering the expected behavior of the knowledge system as a whole.

10.2. The Application of Knowledge Containers

The formalization of knowledge is simplified by the definition of knowledge containers, which are used as a design abstraction. They classify the applied knowledge into ontological knowledge, structural knowledge, strategic knowledge, and support knowledge. Ontological knowledge represents the framework of the knowledge system by defining the basic objects of the knowledge, i.e., diagnoses and questions together with hierarchical relations between them. Thus, questions are semantically grouped by question sets, and diagnoses can describe *is-a* or *is-part-of* relationships to other diagnoses. The structural knowledge container consists of the inferential knowledge, which is applied for deriving diagnoses, i.e., solutions, for a current case. There are various approaches for formalizing structural knowledge; we focussed on explicit representations based on mental models: Abstraction rules, categorical rules, scoring rules, case-based reasoning, and causal set-covering models. Strategic knowledge is applied for guiding the user dialog of the knowledge system. Using strategic knowledge the extensive acquisition of data can be reduced, which is possibly unnecessary for the current case. Ontological knowledge can be augmented with support knowledge, i.e., additional information given by multimedia content like texts, images, and movies. Support knowledge is used to help the user during data acquisition or to explain the derived solution of the knowledge system.

For all presented knowledge containers we also discussed the application of the agile practices, i.e., we introduced methods for testing, for restructuring, and for learning the particular knowledge.

10.3. Knowledge System Development in Practice: d3web

The practical significance of a process model strongly depends on the available tools supporting the application of the process model. This work presented the system family d3web which contains the knowledge modeling environment d3web.KnowME and the user interface d3web.Dialog for using the implemented knowledge system. The knowledge modeling environment d3web.KnowME is a highly integrated workbench for the agile development of diagnostic knowledge systems, including specialized editors for the described knowledge containers, as well as appropriate tools for testing and restructuring the implemented knowledge. The system d3web.Dialog is a web-based server application running the implemented knowledge system. With a web-browser serving as a client, the developed system can be accessed over the internet.

10.4. Experiences with the Process Model

In two projects the presented process model or significant parts of it were used and promising experiences were made. In the LIMPACT project a knowledge system for estimating the pesticide contamination of small water streams was developed. Since this project started before the final definition of the process model, the process was not entirely applied. However, parts of the presented work, like automated testing and the incremental development of set-covering knowledge, were used and their significance was shown. The ECHODOC system was the first adopter of the agile process model. Until now, ontological knowledge, strategic knowledge, and support knowledge are developed using the agile phases and practices. In a next milestone of ECHODOC the extension by structural knowledge is planned. In the context of this project, the implementation of the agile phases (planning game, implementation, integration) has proven to be advantageous. In particular, the application of agile practices, i.e., automated testing and restructuring, were observed to be very beneficial.

11. Discussion

A novel approach for developing diagnostic knowledge systems is presented. The introduced process model proposes an integration of key techniques known from knowledge engineering research. The main contributions to knowledge engineering research are discussed in the following sections.

11.1. Self-Contained Methodology

The agile process model represents a self-contained methodology for the development of diagnostic knowledge systems, i.e., all required aspects for developing a diagnostic knowledge system are considered. Classic development projects consists of the phases requirements analysis, the design phase, the implementation phase, and the test phase. These are covered in the given methodology as described below.

Requirements Analysis The coarse analysis of system requirements is performed during the definition of the system metaphor: Then, the overall scope of the system is specified by the global system metaphor. The detailed requirements of the currently planned aspects of the system are determined by the planning game, e.g., describing the diagnoses to be detected and the possible observations, that could be used as input data for the system. It is worth noticing, that in general the requirements analysis is not preceding all other phases, but is repeatedly performed during the development process. In contrast to classical approaches there is no description available of the overall project requirements. Therefore, during an agile development project we never can refer to a comprehensive document of the planned system. At first sight, the absence of this document seems to be the major disadvantage of the agile process model. However, the repeated application of the requirements analysis in the planning game allows for an adaptive schedule of the development project, including flexible changes during the project. Moreover, the planning game facilitates methods for a concrete feedback of the time and cost estimations made in previous plans. Consequently, an accurate benchmark of the implementation velocity can be given during the project.

Design With the design of the knowledge system we identify the design of the knowledge included in the system. We do not consider general architectural issues as task method design and controller design as, e.g., is done with respect to the CommonKADS methodology [117, Chap. 11]. In the presented process model the architecture is predefined, since focussed knowledge systems are restricted to diagnostic tasks and specialized knowledge roles. The design of the knowledge base is described by the definition of the knowledge containers. Especially the ontological knowledge container consisting of the diagnoses and questions with their hierarchical relationships contain the knowledge design, but also parts of the remaining knowledge containers. The continuous change of knowledge design is a critical issue for knowledge base development, since modifications on large knowledge bases are often very difficult to perform. We partially solved this problem by providing restructuring methods, that represent typical design modification procedures. Restructuring methods do not only change the specified design spot of the knowledge base, but also update existing knowledge corresponding with the spot. If the application of the restructuring method causes (syntactical or semantic) conflicts of the existing knowledge, then the developer is warned and the method is aborted. The problem is only partially solved, because currently we are only providing restructuring methods for the most typical design modifications. This library of methods can be extended easily as it is impossible to cover *all* potential design modifications in advance.

Implementation and Test In the context of the agile process model the test phase is included in the implementation phase. The combination of the validation and the implementation of knowledge emphasises the importance of the validation task: Any code implementation phase should be preceded by a corresponding implementation of test knowledge. To the view of the author, classic knowledge engineering methodologies did not emphasize this relationship appropriately. For example, the CommonKADS methodology [117] focusses on the definition of the particular models, but does not adequately describe the validation of the resulting system. Similarly, the MIKE methodology [6] mainly focusses on the construction of the knowledge system. A visual debugger for KARL models is provided for the validation of the implemented knowledge system. The debugger is comparable to the debugger presented in Section 8.3.8 (p. 176), and do not allow for an automated validation of the knowledge base.

As a major contribution of this work, the implementation task and the corresponding testing task are combined in a single phase. Thus, classical approaches for the validation of knowledge, e.g., Preece et al. [95] and Knauf [61] for rule-based systems, are jointly discussed with more recent approaches, e.g., testing the robustness of knowledge systems [48]. Furthermore, novel approaches are presented suitable for the presented knowledge containers, e.g., partial-ordered question sets for validating strategic knowledge. This work also includes the integration of semi-automatic learning methods for simplifying the acquisition of (large) knowledge bases. In contrast to classical learning methods, semi-automatic learning methods do not only focus on the accuracy of the learned patterns, but also consider the understandability and incrementally of the learned knowledge. For example, in Section 5.4.6 (p. 102) a method for inductively learning understandable rule bases is introduced.

11.2. The Knowledge Modeling Environment d3web.KnowME

In addition to the theoretical description of the agile process model, the system family d3web, and especially the system d3web.KnowME, is presented. Most parts of d3web were implemented in collaboration with colleagues and students. However, the key editors required for an agile development and their tight integration were inspired and contributed by the author of this work. For example, the research on structured modifications of knowledge bases using restructuring methods was started in the context of the master thesis of Timm Michael [74], and later extended and refined, e.g. with respect to the automated adaptation of test knowledge. Furthermore, the JUnit [44] metaphor for automated tests was adapted to the validation of knowledge systems initially by the master thesis of Klaus Akin [3], and later extended by test methods covering further kinds of knowledge. The research on semi-automatic learning methods was initiated with the master thesis of Martin Atzmueller [7], and later continued with Martin Atzmueller focussing on semi-automatically learning set-covering models (see Section 5.5.8) and scoring rules (see Section 5.4.6). Further components for the application of the agile process model, like various editors for test knowledge, specialized editors for (structural) knowledge, the visual debugger, and the limited editor for support knowledge were jointly implemented with students.

11.3. Experiences with Two Real-Life Projects

The presented work has been used in the context of two real world projects, i.e., the LIMPACT and the ECHODOC project, and early experiences show promising results.

In both projects, the domain specialists easily adopted the concepts of the global and local system metaphors. It has been shown, that a common system of names greatly simplified the introduction into the concept of diagnostic knowledge systems, as well as, actually developing the system. During discussions with the domain specialists the common terminology for the overall system idea and the basic entities was experienced to be very beneficial. Furthermore, in the ECHODOC project the planning game was appropriate and helpful for the domain specialist. At the one hand, the planning game was used by the developer as a *personal* outline of the complete project, but also helped when explaining the system to other domain specialists. Especially, the continuous integration demonstrated its practical significance for the ECHODOC project: During the running project the developer could always present the current system to other domain specialists, that give accurate feedback for improving the design of the system.

In both projects, the high integration of test methods has proven to be reasonable for the development of high quality knowledge systems. For example, the implementation of unit cases in the context of the LIMPACT project revealed falsely implemented scoring rules, that were not detected by the previously applied empirical testing method. In the context of the ECHODOC project the application of POQS has been demonstrated its significance. Due to restructurings of the strategic knowledge the automated testing of the several desired dialog paths was very beneficial.

12. Outlook

The agile process model is a novel and fascinating approach for the development of diagnostic knowledge systems. However, for nearly all presented topics there is still room for future work. In the following, we discuss the most promising directions, that we are planning to consider in the future.

12.1. Knowledge Design Analysis using Visualization Techniques

Large knowledge bases tend to become very complex, e.g., including extensive ontological knowledge with a well-elaborated structural knowledge container. Then, the analysis of the knowledge design is very difficult, because with simple methods like *static ontology testing* (p. 53) one cannot acquire a sufficient overview of the implemented knowledge. With knowledge design we consider the structure of the ontological knowledge container and the interweaving of the ontological entities by the remaining knowledge containers.

Appropriate *visualization techniques* are a promising approach for interactively analyzing the design. For example, the knowledge base can be visualized as a graph with ontological entities (diagnoses, questions) as nodes, the size of the nodes is proportional to the number of attached (structural) knowledge; edges between the nodes represent structural knowledge. This simple visualization enables the developer to analyze the design of the knowledge base by simply viewing the graph structure. Therefore, large nodes are emphasized as very frequently applied ontological entities in contrast to small nodes, for which only sparse knowledge is available. Thus, a sub-graph connected to the remaining graph structure only by one node is an indicator for vulnerable knowledge design, since a part of the implemented knowledge depends on a single node. A first step in this direction was undertaken by specifying and implementing a declarative language for defining visualizations [120]. Besides this simple example there are even more opportunities for visualization techniques, e.g., the same graph analysis can be performed on the case based testing method depicting the use of the ontological entities (p. 54). Card [30] gives a recent introduction into the techniques of information visualization, for which some can be easily adapted for knowledge visualization.

12.2. Agile Practices in Legacy Knowledge Systems

The presented work only considers the manual construction of knowledge systems from scratch. It is easy to see that the application of the agile practices testing, restructuring, and learning can be also useful for already existing knowledge systems. We define *legacy knowledge systems* as fielded systems, that were originally not developed using the agile process model. If such systems need to be maintained or extensions of the system become necessary, then automated restructurings with corresponding test suites are a valuable practice for performing complex modifications. Typical restructurings of legacy systems consider the modification of the ontological hierarchies or the insertion of abstractions. An even more complex restructuring method not considered in the context of this work would be the breakdown of a large knowledge base into coherent parts, thus defining smaller but partly independent knowledge bases. Then, the previously large knowledge system is replaced with a collection of smaller knowledge systems (component-based knowledge) with a semantically equivalent reasoning behavior. Theoretical considerations and appropriate techniques for knowledge systems with component-based knowledge are an interesting direction for future work. Related work about the representation and reasoning of distributed knowledge bases can be found in [12, 20].

In the context of splitting large knowledge bases, complex restructurings for integrating knowledge formalization patterns is an issue to consider. Then, the restructuring methods INTRODUCEDIAGNOSTICSCORE and INTRODUCEHEURISTICDECISIONTABLE briefly sketched in Section 5.4.5 (p. 99) need to be further investigated and tool support should be considered (see also Section 12.3.2).

12.3. Extensions of Restructuring Methods

We presented only a small portion of possible restructuring methods. Future work will concentrate on even more complex and powerful methods for restructuring knowledge.

12.3.1. Interactive Restructuring Methods

Currently, only automatic restructuring methods are implemented, i.e., methods that are applicable without possibly causing conflicts. However, often the implemented knowledge will cause conflicts, when a restructuring method is applied. For example, moving the value of an one-choice question to the value range of another one-choice question can cause a rule to become syntactically ambivalent. In order to facilitate the application of restructuring methods possibly causing conflicts, we propose to consider *interactive approaches* with wizard-like tools. Then, the developer successfully can resolve caused conflicts supported by the wizards in a user-friendly way.

12.3.2. Big Restructurings on Knowledge Bases

If the application of restructuring methods is considered for large, already-existing knowledge bases, e.g., legacy knowledge systems, then the introducing of big restructuring methods is a promising direction. Especially, during the implementation of knowledge formalization patterns [102] for already existing knowledge systems the need for big restructurings will arise. Then, interactive and intelligent wizards for conflict resolution during the application of the restructuring method can support the developer.

12.4. Transparent Integration of Learning Methods

We also identified semi-automatic learning methods as an agile practice. However, currently the implemented methods for learning scoring rules and set-covering models are only available in batch mode. A wizard-like integration of these methods into the existing knowledge modeling workbench d3web.KnowME is a promising direction for future work, since this would significantly increase the usability of the presented methods.

AI – The art and science of making computers do interesting things that are not in their nature.

from the november 2003 issue of the *AI Expert Newsletter* –
`http://www.ainewsletter.com`

Bibliography

[1] Agnar Aamodt and Enric Plaza. Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches. *AI Communications*, 7(1), 1994.

[2] Andreas Abecker, Ansgar Bernardi, Knut Hinkelmann, Otto Kuhn, and Michael Sintek. Toward a Technology for Organizational Memories. *IEEE Intelligent Systems*, 13(3):40–48, 1998.

[3] Klaus Akin. Qualitätssicherung in diagnostischen Wissensbasen [Quality Management for Diagnostic Knowledge Bases]. Master's thesis, University Wuerzburg, Department for Computer Science VI, Wuerzburg, Germany, 2003.

[4] Steen Andersen, Kristian G. Olesen, and Finn Jensen. HUGIN - a Shell for Building Bayesian Belief Universes for Expert Systems. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence (IJCAI-89)*, 1989.

[5] C3-Team: Ann Anderson, Ralph Beattie, Kent Beck, David Bryant, Marie DeArment, Martin Fowler, Margaret Fronczak, Rich Garzaniti, Dennis Gore, Brian Hacker, Chet Hendrickson, Ron Jeffries, Doug Joppie, David Kim, Paul Kowalsky, Debbie Mueller, Tom Murasky, Richard Nutter, Adrian Pantea, and Don Thomas. Chrysler goes to 'Extremes'. *Distributed Object Computing*, pages 24–28, October 1998.

[6] Jürgen Angele, Dieter Fensel, Dieter Landes, and Rudi Studer. Developing Knowledge-Based Systems with MIKE. *Automated Software Engineering: An International Journal*, 5(4):389–418, October 1998.

[7] Martin Atzmueller. Semi-automatic Data Mining Methods for Improving Case-Based Reasoning. Master's thesis, University Wuerzburg, Department for Computer Science VI, Wuerzburg, Germany, 2002.

[8] Martin Atzmueller, Joachim Baumeister, and Frank Puppe. Evaluation of two Strategies for Case-Based Diagnosis handling Multiple Faults. In *Proceedings of the 2nd Conference of Professional Knowledge Management (WM2003)*, Luzern, Switzerland, 2003.

[9] Martin Atzmueller, Joachim Baumeister, and Frank Puppe. Inductive Learning of Simple Diagnostic Scores. In *Proceedings of the 4th International Symposium on Medical Data Analysis (ISMDA 2003)*, LNCS 2868, Berlin, 2003. Springer Verlag.

[10] Martin Atzmueller, Joachim Baumeister, and Frank Puppe. Quality Measures for Semi-Automatic Learning of Simple Diagnostic Rule Bases. In *Proceedings of the 15th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2004)*, pages 203–213, 2004.

[11] Marc Ayel and Jean-Pierre Laurent. *Validation, Verification and Test of Knowledge-Based Systems*. Wiley, 1991.

[12] Stefan K. Bamberger. Cooperating Diagnostic Expert Systems to Solve Complex Diagnosis Tasks. In *Proceedings of the German Conference on Artificial Intelligence (KI 1997)*, LNCS 1303, pages 325–336, Berlin, 1997. Springer Verlag.

[13] Ralph Barletta and William Mark. Explanation-Based Indexing of Cases. In *Proceedings of the 7th Annual National Conference on Artificial Intelligence (AAAI-88)*. Morgan Kaufmann Publisher, 1988.

[14] Joachim Baumeister, Martin Atzmueller, and Frank Puppe. Inductive Learning for Case-Based Diagnosis with Multiple Faults. In *Proceedings of the 6th European Conference on Case-Based Reasoning (ECCBR 2002)*, LNAI 2416, pages 28–42, Aberdeen, Scotland, 2002. Springer Verlag.

[15] Joachim Baumeister and Dietmar Seipel. Diagnostic Reasoning with Multilevel Set–Covering Models. In *Proceedings of the 13th International Workshop on Principles of Diagnosis (DX-02)*, Semmering, Austria, 2002.

[16] Joachim Baumeister, Dietmar Seipel, and Frank Puppe. Incremental Development of Diagnostic Set-Covering Models with Therapy Effects. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 11(Suppl.):25–50, November 2003.

[17] Joachim Baumeister, Dietmar Seipel, and Frank Puppe. An Agile Process Model for Developing Diagnostic Knowledge Systems. *KI Journal (Special Issue on 'AI and Software Engineering')*, 3, 2004.

[18] 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)*, 2004.

[19] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.

[20] Mitchel Berberich and Stefan Bamberger. Building Web-based Knowledge Clusters. In *Proceedings of the Colloquium 'Web-based Knowledge Servers'*. IEE, 1998.

[21] Ralph Bergmann and Armin Stahl. Similarity Measures for Object-Oriented Case Representations. In *Advances in Case-Based Reasoning, 4th European Workshop on Case-Based Reasoning (EWCBR-98)*, LNCS 1488, pages 25–36, Dublin, Ireland, 1998.

[22] Robin Boswell and Susan Craw. *Organizing Knowledge Refinement Operators, In: Validation and Verification of Knowledge Based Systems*, pages 149–161. Kluwer, Oslo, Norway, 1999.

[23] Norman Brümmer. Überdeckungsmodelle zur Akquisition und Verarbeitung von unsicherem Wissen mit einer Überführung in Bayes'sche Netze [Set-Covering Models for the Acquisition and Inference of Uncertain Knowledge with a Transformation to Bayesian Networks]. Master's thesis, University Wuerzburg, Department for Computer Science VI, Wuerzburg, Germany, 2003.

[24] B.G. Buchanan and E.H. Shortliffe. *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*. Addison-Wesley, 1984.

[25] Bruce G Buchanan and Edward A. Feigenbaum. DENDRAL and META-DENDRAL: Their Applications Dimensions. *Artificial Intelligence*, 11:5–24, 1978.

[26] Reinhard Budde, Karlheinz Kautz, Karin Kuhlenkamp, and Heinz Züllinghoven. *Prototyping: An Approach to Evolutionary System Development*. Springer Verlag, Berlin, 1992.

[27] Reinhard Budde, Karin Kuhlenkamp, Lars Mathiassen, and Heinz Zülighoven. *Approaches to Prototyping*. Springer Verlag, Berlin, 1984.

[28] Hans-Peter Buscher, Ch. Engler, A. Führer, S. Kirschke, and F. Puppe. HepatoConsult: A Knowledge-Based Second Opinion and Documentation System. *Artificial Intelligence in Medicine*, 24(3):205–216, 2002.

[29] Tom Bylander and Chandrasekaran. Generic Tasks in Knowledge-Based Reasoning. In Brian R. Gaines and John H. Boose, editors, *Knowledge Acquisition for Knowledge-Based Systems*, pages 65–77. Academic Press, 1988.

[30] Stuart Card. Information Visualization. In *Julie A. Jacko, Andrew Sears: The Human-Computer Interaction Handbook*, pages 544–582. Lawrence Erlbaum Associates, 2003.

[31] William J. Clancey. The Epistemology of a Rule-Based Expert System: A Framework for Explanation. *Artificial Intelligence*, 20:215–251, 1983.

[32] Alistair Cockburn. *Agile Software Development*. Addison-Wesley, 2002.

[33] Frans Coenen and Trevor Bench-Capon. *Maintenance of Knowledge-Based Systems*. Academic Press, 1993.

[34] C.M. Cooper. Biological effects of agriculturally derived surface-water pollutants on aquatic systems — a review. *Environ. Qual.*, 22:402–408, 1993.

[35] Stephen Cranefield and Martin Purvis. UML as an Ontology Modelling Language. In *Proceedings of the Workshop on Intelligent Information Integration, 16th International Joint Conference on Artificial Intelligence (IJCAI-99)*, 1999.

[36] Cynthia A. Thompson and Raymond J. Mooney. Inductive Learning for Abductive Diagnosis. In *Proceedings of the 13th Annual National Conference on Artificial Intelligence (AAAI-94), Vol. 1*, pages 664–669, 1994.

[37] Francisco J. Díez. Parameter Adjustment in Bayesian Networks: The Generalized Noisy Or-Gate. In *9th Conference on Uncertainty in Artificial Intelligence (UAI '93)*, pages 99–105, 1993.

[38] David C. Dugdale and Mickey S. Eisenberg. *Medical Diagnostics*. W.B. Saunders Company, 1992.

[39] Hans-Peter Eich and Christian Ohmann. Internet-Based Decision-Support Server for Acute Abdominal Pain. *Artificial Intelligence in Medicine*, 20(1):23–36, 2000.

[40] Dieter Fensel, Jurgen Angele, and Rudi Studer. The Knowledge Acquisition and Representation Language KARL. *Knowledge and Data Engineering*, 10(4):527–550, 1998.

[41] Kenneth D. Forbus and Johan de Kleer. *Building Problem Solvers*. MIT Press, 1992.

[42] Martin Fowler. *Refactoring. Improving the Design of Existing Code*. Addison-Wesley, 1999.

[43] Anthony G. Francis and Ashwin Ram Jr. Computational Models of the Utility Problem and their Application to Utility Analysis of Case-Based Reasoning. In *Proceedings of the Workshop on Knowledge Compilation and Speedup Learning (KCSL 93)*, 1993.

[44] Erich Gamma and Kent Beck. Test Infected: Programmers Love Writing Tests. *Java Report*, 3(7), 1998.

[45] Aldo Gangemi, Domenico M. Pisanelli, and Geri Steve. An Overview of the ONIONS Project: Applying Ontologies to the Integration of Medical Terminologies. *Data and Knowledge Engineering*, 31(2):183–220, 1999.

[46] Fausto Giunchiglia and Toby Walsh. A Theory of Abstraction. *Artificial Intelligence*, 57(2-3):323–389, 1992.

[47] Klaus Goos. *Fallbasiertes Klassifizieren – Methoden, Integration und Evaluation [Case-Based Classification – Methods, Integration and Evaluation]*. Number 127 in diski. infix Verlag, Berlin, 1995.

[48] Perry Groot, Annette ten Teije, and Frank van Harmelen. A Quantitative Analysis of the Robustness of Knowledge-Based Systems through Degradation Studies. *Knowledge and Information Systems*, (to appear), 2004.

[49] Perry Groot, Frank van Harmelen, and Annette ten Teije. Torture Tests: A Quantitative Analysis for the Robustness of Knowledge-Based Systems. In *Knowledge Acquisition, Modeling and Management*, LNAI 1319, pages 403–418, Berlin, 2000. Springer Verlag.

[50] William Grosso, Henrik Eriksson, Ray W. Fergerson, John H. Gennari, Samson W. Tu, and Mark Musen. Knowledge Modeling at the Millennium – The Design and Evolution of Protégé-2000. In *Proceedings of the 12th International Workshop on Knowledge Acquisition, Modeling and Management (KAW 1999)*, Banff, Canada, 1999.

[51] Thomas R. Gruber. A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition*, 5(2):199–220, 1993.

[52] David Hand, Heikki Mannila, and Padhraic Smyth. *Principles of Data Mining*. MIT Press, 2001.

[53] Jr. Harry E. Pople. *Heuristic Methods for Imposing Structure on Ill-Structured Problems: The Structuring of Medical Diagnostics*, chapter 5. AAAS, Westview Press, 1982.

[54] Frederick Hayes-Roth, Donald A. Waterman, and Douglas B. Lenat. *Building Expert Systems*. Addison-Wesley, 1983.

[55] David Heckerman. Probabilistic Interpretation for MYCIN's Certainty Factors. In *Proceedings of the 1st Annual Conference on Uncertainty in Artificial Intelligence (UAI 1985)*, pages 167–196, 1985.

[56] David E. Heckerman. *Probabilistic Similarity Networks*. MIT Press, 1991.

[57] Alexander Hörnlein, Christian Betz, and Frank Puppe. Redesign eines generativen, fallbasierten Trainingssystems für das WWW in d3web.Train. In *Rechnergestützte Lehr- und Lernsysteme in der Medizin - Proceedings zum 6. Workshop der GMDS AG Computergestützte Lehr- und Lernsysteme in der Medizin*, Aachen, 2002. Shaker Verlag.

[58] 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*, (in press), 2004.

[59] IEEE Standard Glossary of Software Engineering Terminology, IEEE Standard 610.12-1990, ISBN 1-55937-067-X, 1990.

[60] 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.

[61] Rainer Knauf. *Validating Rule-Based Systems: A Complete Methodology*. Shaker, Aachen, Germany, 2000.

[62] 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.

[63] Paul Kogut, Stephen Cranefield, Lewis Hart, Mark Dutra, Kenneth Baclawski, Mieczyslaw Kokar, and Jeffrey Smith. UML for Ontology Development. *The Knowledge Engineering Review*, 17(1):61–64, 2002.

[64] Janet Kolodner. *Case-Based Reasoning*. Morgan Kaufmann Publisher, 1993.

[65] Dieter Landes. DesignKARL - A Language for the Design of Knowledge-Based Systems. In *Proceedings of the 6th International Conference on Software and Knowledge Engineering (SEKE 1994)*, Jurmala, Latvia, 1994.

[66] Mario Lenz. *Case Retrieval Nets as a Model for Building Flexible Information Systems*. Number 236 in diski. infix Verlag, Berlin, 2000.

[67] Matthias Liess and Ralf Schulz. Linking Insecticide Contamination and Population Response in an Agricultural Stream. *Environmental Toxicology and Chemistry*, 18:1948–1955, 1999.

[68] Karl-Werner Lorenz, Joachim Baumeister, Clemens 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.

[69] Peter Lucas, Astrid Tholen, and Geeske van Oort. An Intelligent System for Pacemaker Reprogramming. *Artificial Intelligence in Medicine*, 17:249–269, 1999.

[70] Alexander Maedche, Viktor Pekar, and Steffen Staab. Ontology Learning Part One - On Discovering Taxonomic Relations from the Web. In Ning Zhong, editor, *Web Intelligence*. Springer Verlag, 2002.

[71] Sandra Marcus. *Automating Knowledge Acquisition for Expert Systems*. Kluwer Academic Publisher, 1988.

[72] Tim Menzies. Knowledge Maintenance: The State of the Art. *The Knowledge Engineering Review*, 14(1):1–46, 1999.

[73] Dennis Merritt. *Building Expert Systems in Prolog*. Springer Verlag, Berlin, 1989.

[74] Timm Michael. Komponenten zum Refactoring von wissensbasierten Systemen [Components for the Refactoring of Knowledge Systems]. Master's thesis, University Wuerzburg, Department for Computer Science VI, 2001.

[75] JP Miller, AS Lambert, and WA Shapiro. The Adequacy of Basic Intraoperative Transesophageal Echocardiography performed by Experienced Anesthesiologists. *Anesthesia and Analgesia*, 92:1103–1110, 2001.

[76] Randolph A. Miller, Harry E. Pople, and J. Myers. INTERNIST-1, an Experimental Computer-Based Diagnostic Consultant for General Internal Medicine. *New England Journal of Medicine*, 307:468–476, 1982.

[77] Robert Milne and Charlie Nicol. TIGER: Continuous Diagnosis of Gas Turbines. In *Proceedings of the 14th European Conference on Artificial Intelligence*, Berlin, Germany, 2000.

[78] Robert Milne, Charlie Nicol, Louise Travé-Massuyès, and Joseba Quevedo. TIGER: Knowledge Based Gas Turbine Condition Monitoring. *AI Communications*, 9(3):92–108, 1996.

[79] Michael Neumann and Joachim Baumeister. A Rule-Based vs. a Model-Based Implementation of the Knowledge System LIMPACT and its Significance for Maintenance and Discovery of Ecological Knowledge. In *Proceedings of the 3rd Conference of the International Society of Ecological Informatics (ISEI-02)*, Rom, Italy, 2002.

[80] Michael Neumann, Joachim Baumeister, Matthias Liess, and Ralf Schulz. An Expert System to Estimate the Pesticide Contamination of Small Streams using Benthic Macroinvertebrates as Bioindicators, Part 2: The Knowledge Base of LIMPACT. *Ecological Indicators, Elsevier Science*, 2(4):391–401, 2003.

[81] Michael Neumann, Joachim Baumeister, and Frank Puppe. ILMAX: A System for Managing Experience Knowledge in a long-term Study of Stream Ecosystem Regeneration - an Application of Ecological Informatics. In *Proceedings of the First International NAISO Symposium on Information Technologies in Environmental Engineering (ITEE)*, Gdansk, Poland, 2003.

[82] Michael Neumann and David Dudgeon. The impact of agricultural runoff on stream benthos in hong kong, china. *Water Research*, 36(12):3093–3099, 2002.

[83] Michael Neumann, Matthias Liess, and Ralf Schulz. An Expert System to Estimate the Pesticide Contamination of Small Streams using Benthic Macroinvertebrates as Bioindicators, Part 1: The Database of LIMPACT. *Journal Ecological Indicators, Elsevier Science*, 2(4):379–389, 2003.

[84] Natalya Fridman Noy, William Grosso, and Mark Musen. Knowledge-Acquisition Interfaces for Domain Experts: An Empirical Evaluation of Protégé-2000. In *Proceedings of the 12th International Conference on Software Engineering and Knowledge Engineering (SEKE 2000)*, Chicago, USA, 2000.

[85] Christian Ohmann and et al. Clinical Benefit of a Diagnostic Score for Appendicitis: Results of a Prospective Interventional Study. *Archives of Surgery*, 134:993–996, 1999.

[86] Agnieszka Onisko, Marek J. Druzdzel, and Hanna Wasyluk. Extension of the Hepar II Model to Multiple-Disorder Diagnosis. In *Intelligent Information Systems*, Advances in Soft Computing Series, pages 303–313, Heidelberg, 2000. Physica-Verlag.

[87] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, Urbana-Champaign, IL, USA, 1992.

[88] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publisher, San Mateo, California, 1988.

[89] Yun Peng and James A. Reggia. *Abductive Inference Models for Diagnostic Problem-Solving*. Springer Verlag, Berlin, 1990.

[90] Robert Plant and Rose Gamble. Methodologies for the Development of Knowledge-Based Systems, 1982–2002. *The Knowledge Engineering Review*, 18:47–81, 2003.

[91] Karsten Poeck and Ute Gappa. Making Role-Limiting Shells more Flexible. In *Knowledge Acquisition for Knowledge Based Systems (EKAW 1993)*, pages 103–122, 1993.

[92] Klaus Poeck. *Neurologie*. Springer Verlag, Berlin, 9 edition, 1994.

[93] Alun Preece. Building the Right System Right. In *Proceedings of KAW'98 Eleventh Workshop on Knowledge Acquisition, Modeling and Management*, 1998.

[94] Alun Preece and Rajjan Shinghal. Foundation and Application of Knowledge Base Verification. *International Journal of Intelligent Systems*, 9:683–702, 1994.

[95] Alun Preece, Rajjan Shinghal, and Aida Batarekh. Principles and Practice in Verifying Rule-Based Systems. *The Knowledge Engineering Review*, 7 (2):115–141, 1992.

[96] Alun Preece, Rajjan Shinghal, and Aida Batarekh. Verifying Expert Systems. A Logical Framework and a Practical Tool. *Expert Systems with Applications*, 5(3/4):421–436, 1992.

[97] Asunción Gómez Pérez and V. Richards Bejamins. Overview of Knowledge Sharing and Reuse Components: Ontologies and Problem-Solving Methods. In V.R. Bejamins, B. Chandrasekaran, A. Gómez Pérez, M Guarino, and M. Uschold, editors, *Proceedings of the IJCAI-99 Workshop on Ontologies and Problem-Solving Methods (KRR5)*, 1999.

[98] Frank Puppe. Requirements for a Classification Expert System Shell and their Realization in MED2. *Applied Artificial Intelligence*, 1:163–171, 1987.

[99] Frank Puppe. *Systematic Introduction to Expert Systems*. Springer Verlag, Berlin, 1993.

[100] Frank Puppe. Knowledge Reuse among Diagnostic Problem-Solving Methods in the Shell-Kit D3. *International Journal of Human-Computer Studies*, 49:627–649, 1998.

[101] Frank Puppe. Meta Knowledge for Extending Diagnostic Consultation to Critiquing Systems. In *Proceedings of the 11th European Workshop on Knowledge Acquisition, Modeling and Management (EKAW'99)*, pages 367–372, Berlin, 1999. Springer Verlag.

[102] Frank Puppe. Knowledge Formalization Patterns. In *Proceedings of PKAW 2000*, Sydney, Australia, 2000.

[103] Frank Puppe and Ute Gappa. Knowledge Acquisition with the Classification Shell D3/CLASSIKA. In *Proceedings of the IJCAI-89-Workshop "Diagnostic Systems for Manufacturing"*, 1989.

[104] Frank Puppe, Ute Gappa, Karsten Poeck, and Stefan Bamberger. *Wissensbasierte Diagnose– und Informationssysteme. Mit Anwendung des Experten–Shell–Baukasten D3*. Springer Verlag, Berlin, 1996.

[105] Frank Puppe and Bernhard Puppe. Overview on MED1: An Heuristic Diagnostics System with an Efficient Control Structure. In *Proceedings of the German Workshop on Artificial Intelligence (GWAI-83), Informatik-Fachberichte 76*, pages 11–20. Springer Verlag, 1983.

[106] Frank Puppe, Susanne Ziegler, Ulrich Martin, and Jürgen Hupp. *Wissensbasierte Diagnosesysteme im Service-Support [Knowledge-Based System for Service-Support]*. Springer Verlag, Berlin, 2001.

[107] James Reggia. Computer-Assisted Medical Decision Making. In *Schwartz (ed.): Applications of Computers in Medicine*, pages 198–213. IEEE, 1982.

[108] 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.

[109] Thomas Reinartz, Ioannis Iglezakis, and Thomas Roth-Berghofer. Review and Restore for Case Base Maintenance. *Computational Intelligence: Special Issue on Maintaining CBR Systems*, 17(2):214–234, 2001.

[110] Michael Richter. The Knowledge contained in Similarity Measures. Invited talk at the International Conference on Case-Based Reasoning (ICCBR-95), http://www.cbr-web.org/documents/Richtericcbr95remarks.html, 1995.

[111] Michael M. Richter. Classification and Learning of Similarity Measures. In *Proceedings der Jahrestangung der Gesellschaft für Klassifikation, Studies in Classification, Data Analysis and Knowledge Organisation*, 1992.

[112] Michael M. Richter. Fallbasiertes Schliessen – Vergangenheit, Gegenwart, Zukunft [Case-Based Reasoning – Past, Present, Future]. *Informatik Spektrum*, 26(3):180–190, June 2003.

[113] Mark Richters and Martin Gogolla. On Formalizing the UML Object Constraint Language OCL. In Tok-Wang Ling, Sudha Ram, and Mong Li Lee, editors, *Proceedings 17th International Conference on Conceptual Modeling (ER 1998)*, volume 1507, pages 449–464, Berlin, 1998. Springer Verlag.

[114] Thomas Roth-Berghofer and Ioannis Iglezakis. Six Steps in Case–Based Reasoning: Towards a Maintenance Methodology for Case–Based Reasoning Systems. In Hans-Peter Schnurr, Steffen Staab, Rudi Studer, Gerd Stumme, and York Sure, editors, *Professionelles Wissensmanagement — Erfahrungen und Visionen (Includes Proceedings of the 9th German Workshop on Case–Based Reasoning, (GWCBR 2001)), Baden–Baden, Germany*, pages 198–208, Aachen, 2001. Shaker–Verlag.

[115] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[116] Manfred Schramm and Wolfgang Ertel. Reasoning with Probabilities and Maximum Entropy: The System PIT and its Application in LEXMED. In K. Inderfurth et al., editor, *Operations Research Proceedings*, pages 274–280. Springer Verlag, 1999.

[117] Guus Schreiber, Hans Akkermans, Anjo Anjewierden, Robert de Hoog, Nigel Shadbolt, Walter Van de Velde, and Bob Wielinga. *Knowledge Engineering and Management - The CommonKADS Methodology*. MIT Press, 2 edition, 2001.

[118] Guus Schreiber, Bob Wielinga, and Joost Breuker. *KADS - A Principled Approach to Knowledge-Based System Development*. Academic Press, 1993.

[119] Guus Schreiber, Bob Wielinga, Robert de Hoog, Hans Akkermans, and Walter Van de Velde. CommonKADS: A Comprehensive Methodology for KBS Development. *IEEE Expert*, 9(6):28–37, 1994.

[120] Dietmar Seipel, Joachim Baumeister, and Marbod Hopfner. Declarative 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.

[121] Edward H. Shortliffe, Leslie E. Perreault, Gio Wiederhold, and Lawrence M. Fagan. *Medical Informatics*. Springer Verlag, second edition, 2000.

[122] David Sidebotham, Alan Merry, and Malcolm Legget. *Practical Perioperative Transoesophageal Echocardiography*. Elsevier Science Health Science div, 2003.

[123] Barry Smyth and Mark T. Keane. Remembering To Forget: A Competence-Preserving Case Deletion Policy for Case-Based Reasoning Systems. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence (IJCAI-1995)*, pages 377–383, 1995.

[124] Ian Sommerville. *Software Engineering*. Addison-Wesley, 5 edition, 1996.

[125] John F. Sowa. *Knowledge Representation: Logical, Philosophical, and Computational Foundations*. Brooks Cole Publishing Co., 2000.

[126] Steffen Staab and Rufi Studer, editors. *Handbook on Ontologies*. International Handbooks on Information Systems. Springer Verlag, Berlin, 2004.

[127] Craig Stanfill and David Waltz. Towards Memory-Based Reasoning. *Communications of the ACM*, 29(12):1213–1228, 1986.

[128] Mark Stefik. *Introduction to Knowledge Systems*. Morgan Kaufmann Publisher, 1995.

[129] Peter Struss, Martin Sachenbacher, and Claes Carlén. Insights from Building a Prototype for Model-based On-board Diagnosis of Automotive Systems. In *Proceedings of the International Workshop on Principles of Diagnosis (DX-00)*, 2000.

[130] Rudi Studer, V. Richard Benjamins, and Dieter Fensel. Knowledge Engineering: Principles and Methods. *Data and Knowledge Engineering*, 25(1-2):161–197, 1998.

[131] Rudi Studer, Dieter Fensel, Stefan Decker, and V. Richard Benjamins. Knowledge Engineering: Survey and Future Directions. In *Puppe, F. (ed.): XPS-99: Knowledge-Based Systems – Survey and Future Directions. 5th Biannual German Conference on Knowledge-Based Systems*, pages 1–23, 1999.

[132] E. Burton Swanson. The Dimensions of Maintenance. In *Proceedings of the 2nd International Conference of Software Engineering*, 1976.

[133] Gertjan van Heijst, Sabina Falasconi, Ameen Abu-Hanna, Guus Schreiber, and Marlo Stefanelli. A Case Study in Ontology Library Construction. *Artificial Intelligence in Medicine*, 7:227–255, 1995.

[134] Gertjan van Heijst, Guus Schreiber, and Bob J. Wielinga. Using explicit Ontologies in KBS Development. *International Journal of Human-Computer Studies*, 46(2-3):183–292, 1997.

[135] Stefan Wess, Klaus-Dieter Althoff, and Guido Derwand. Using k-d Trees to Improve the Retrieval Step in Case-Based Reasoning. In *Proceedings of the 1st European Workshop on Case-Based Reasoning (EWCBR-93)*, pages 167–181, 1993.

[136] Dietrich Wettschereck and David W. Aha. Weighting Features. In Manuela Veloso and Agnar Aamodt, editors, *Case-Based Reasoning, Research and Development, 1st International Conference*, pages 347–358, Berlin, 1995. Springer Verlag.

[137] D. Randall Wilson and Tony R. Martinez. Improved Heterogeneous Distance Functions. *Journal of Artificial Intelligence Research*, 6:1–34, 1997.

[138] Ian H. Witten and Eibe Frank. *Data Mining*. Morgan Kaufmann Publisher, 1999.

[139] Jun Zhu and Qiang Yang. Remembering to Add: Competence-preserving Case-Addition Policies for Case Base Maintenance. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI-1999)*, pages 234–241, 1999.

[140] `http://www.echodoc.net`.

[141] `http://logic.stanford.edu/kif/kif.html`.

[142] `http://www.medicoconsult.de/html/sonoConsult.html`.

[143] `http://jakarta.apache.org/tomcat`.

A. Restructuring Methods – a Catalog

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

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

For the convenient integration of restructuring methods into a knowledge modeling environment, specialized tools need to be offered to the user. For this reason, the degree of automatization is an interesting property of restructuring methods. We distinguish between the three different degrees of automatization: *A* (auto), *D* (default), and *I* (interactive). A restructuring method is labeled with degree *A*, if the method can be applied automatically without any user interaction, and can be fully executed by a specialized tool. A method is labeled with *D*, if default values can be used for an automatic executing of the restructuring method. If defaults are known, then full tool support can be provided. If the method causes conflicts within the modeled knowledge, that cannot be solved by defaults, then the method needs to be executed interactively (*I*) by the knowledge developer. However, even for interactive restructuring methods, tools can give helpful support during execution.

It is worth noticing, that the degree of automatization often depends on the currently modeled knowledge, e.g., a restructuring method may cause conflicts for one knowledge base, but will be fully axiomatizable for another knowledge base. Therefore, the degree of automatization is attached to the method that can be guaranteed at least.

We will provide a detailed description of the following restructuring methods; the examples motivating the methods are taken from an exemplary knowledge base for the diagnosis of car faults.

Name of method	Degree of automation
ComposeQuestionSets (p. 223)	A
ExtractQuestionSet (p. 225)	A
MoveQuestionValue (p. 227)	I
RemoveDiagnosis (p. 230)	D
RemoveQuestion (p. 232)	D
ShrinkValueRange (p. 235)	I
TransformMCIntoYN (p. 239)	A
TransformNumIntoOC (p. 242)	I
TransformYNIntoMC (p. 245)	I

COMPOSEQUESTIONSETS $\boxed{A}$

The question or question sets contained in two question sets are merged in a new question set.

Motivation

There exist two possibly small question sets that are correlating by indication rules between the questions of these question sets. An aggregated question set containing both may simplify the design and structure of the knowledge base.

Consequences

Similarly to the EXTRACTQUESTIONSET method the rule actions of indications rules need to be considered. Then, all rules containing one of the combined question sets in their rule action are modified, so that they now indicate the composed question set.

Let $QS_1 = (Q_1, \dots, Q_n)$ and $QS_2 = (Q'_1, \dots, Q'_m)$ be abstract question sets containing question and other question sets, respectively. The newly created question set QS is inserted at the position of the original question set QS_1 and is defined as $QS = (Q_1, \dots, Q_n, Q'_1, \dots, Q'_m)$.

Strategic Knowledge: Indication Rules
All indication rules indicating QS_1 or QS_2 are modified so that they are indicating the new question set QS instead.

Mechanics

The restructuring method is performed by the following procedure:

1 Apply test suite to the knowledge system and abort, if errors are reported. Especially, consider the strategic test knowledge.
2 Select the question sets QS_1 and QS_2 that should be combined.
3 Create a new question set QS at the position of the question set QS_1.
4 Move questions and question sets contained in QS_1 and QS_2 to the new question set QS by preserving the original order of the contained questions (sets).
5 Modify indication rules that target the question sets QS_1 and QS_2 as described in the *Consequences* section.
6 Remove the old question sets QS_1 and QS_2.
7 Apply the test suite to the restructured knowledge system and cancel the restructuring method, if errors are reported; alternatively start a debug session. Especially, consider the strategic test knowledge.

Example

The question sets ”exhaust observations” (EO) with questions ”exhaust fumes” (EF) and ”exhaust pipe color” (EPC), and the question set ”engine observations” (EG) with the questions ”engine noise” (EN) and ”engine start” (ES) and should be aggregated into a new question set called ”observations” (O). Let $EO = (EF, EPC)$ and $EG = (EN, ES)$, and the following indication rules are given
$r_1 = cond(r_1) \rightarrow indicate(EO)$.
$r_2 = cond(r_2) \rightarrow indicate(EO, EG)$.
With the transformation the new question set O is generated with $O = (EF, EPC, EN, ES)$ and the rules are modified as follows
$r'_1 = cond(r_1) \rightarrow indicate(O)$.
$r'_2 = cond(r_2) \rightarrow indicate(O)$.

Related Methods

EXTRACTQUESTIONSET (inverse).

EXTRACTQUESTIONSET

A

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.

Motivation

The number of questions contained in one question set may accumulate during the continuous development of the ontological knowledge container. In order to facilitate a more compact and meaningful representation of the available questions large question sets can be partitioned into smaller chunks that contain semantically related questions.

Consequences

The semantics of the implemented strategic knowledge is affected, since extracted questions are not indicated any more. Let $QS = (Q_1, \ldots, Q_n)$ be the original question set, and $QS' = \{Q_k, \ldots, Q_m\}$ be the questions extracted from QS, i.e., $QS' \subseteq QS$.

Strategic Knowledge: Indication Rules
During the execution of the restructuring method we need to consider all indication rules targeting the question set QS. All indication rules indicating QS are modified so that they are also indicating the extracted question set QS'. 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, i.e., $Q_k = Q_1$, then we indicate QS' before QS; otherwise, QS is indicated before QS'. With this procedure the original indication sequence can be preserved.
If follow-up questions Q are extracted without their parent question, then indication rules targeting Q are modified so that they are indicating the extracted question set QS'.

Mechanics

The restructuring method is performed by the following procedure:

1 Apply test suite to the knowledge system and abort, if errors are reported. Especially, consider the strategic test knowledge.
2 Select the question set QS and the questions $\{Q_k, \ldots, Q_m\}$ to be extracted.
3 Create new question set QS' at the position after the question set QS.
4 Move questions $\{Q_k, \ldots, Q_m\}$ to the question set QS.
5 Modify indication rules that target the original question QS and follow-up questions contained in QS (see *Consequences* section).
6 Apply the test suite to the restructured knowledge system and cancel the restructuring method, if errors are reported; alternatively start a debug session. Especially, consider the strategic test knowledge.

Example

The question set ”car observations” (CO) with questions ”exhaust fumes” (EF), ”exhaust pipe color” (EPC), ”engine noise”, and ”fuel” (F) should be simplified. Let $CO = (EF, EPC, F)$, and the questions ”EF” and ”EPC” should be extracted to a new question set considering the exhaust pipe (”EP”). The following rule is contained in the knowledge base
$r_1 = cond(r_1) \rightarrow indicate(CO)$.
With the transformation the rule is modified as follows
$r'_1 = cond(r_1) \rightarrow indicate(EP, CO)$.
The generated question set is indicated before the original question set because the first question of ”CO” is extracted to ”EP”.

Related Methods

COMPOSEQUESTIONSETS (inverse).

MoveQuestionValue

A value of a choice question is moved to another choice question.

Motivation

During the development of the knowledge system the requirement may arise that the detail of a choice question need to be increased. Typically, the granularity is enhanced by dividing the question into a set of questions covering specialized aspects of the finding. As a consequence, it may happen that one or mores values of the original question need to be moved to a newly created choice question.

Consequences

Let $Q \in \Omega_Q$ the original choice question and $v \in dom(Q)$ the value to by moved to the choice question Q'.

Conflicts can be caused by ambivalent rule conditions, inconsistent cases, and cyclic indication rules.

Knowledge with Rule Conditions

Ambivalent *and* condition: If Q and Q' are used by an *and* condition, then it can happen, that the restructuring causes an ambivalent rule condition, e.g., the condition
$and\big(choiceEqual(Q, v), choiceEqual(Q', v')\big)$
will be modified by the method
$and\big(choiceEqual(Q', v), choiceEqual(Q', v')\big)$,
which is an ambivalent rule condition never evaluating to true for an one-choice question Q'. Consequently, for a one-choice question Q' the method will be aborted. However, the conflict can be resolved by converting the question into a multiple-choice question.

Ambivalent *min/max* condition: As described above for an *and* condition the application of the restructuring method can cause an ambivalent *min/max* condition. Then, more than one constrained value for the same one-choice question is contained in the min/max condition, e.g.,
$minMax(choiceEqual(Q', v), choiceEqual(Q', v'), \ldots, min, max)$.
For an one-choice question Q' the first two sub-conditions can never evaluate to true at the same time. Then, a warning is reported but the condition is not altered by default.

Structural Knowledge: Abstraction Rules

The application of the method can cause an abstraction rule to become cyclic, e.g., the rule
$r = condEqual(Q', v') \rightarrow Q{:}v$,
will be modified to

$r = condEqual(Q', v') \rightarrow Q'{:}v\,.$
In such a case a warning is reported but the rule is retained by default.

Structural Knowledge: Set-Covering Knowledge

Due to the restructuring the local similarity measure and the abnormality function defined for the questions Q and Q' need to be revisited.

Structural Knowledge: Case-Based Knowledge

Due to the restructuring the local similarity measure and the abnormality function defined for the questions Q and Q' need to be revisited. If the value v is assigned to the question Q in a case c, i.e., $Q{:}v \in \mathcal{F}_c$, then this finding is removed from c and a new finding $Q'{:}v$ is added to $\mathcal{F}_c$. However, if in the case a value $v' \in dom(Q')$ has been already assigned to question Q', then we cannot automatically move the value v to question Q'. Therefore, we need to decide by an default value, how to proceed with conflicting cases. We either

1 overwrite the original value of question Q' with the value v, or
2 retain the original value of question Q', or
3 remove the conflicting case c from the case base CB.

Strategic Knowledge: Indication Rules

Cyclic indication paths: Moving a value to another question can create cyclic indication paths, e.g., the indication rule
$r = choiceEqual(Q, v) \rightarrow indicate(Q')\,,$
will become the cyclic rule
$r' = choiceEqual(Q', v) \rightarrow indicate(Q')\,.$
Beside this direct indication cycle, other cyclic dependencies of multiple indication rules may be generated. Cyclic indication paths are reported as an error and have to be resolved manually be the developer.

Broken indication paths: The restructuring method may break indication paths, e.g., described by decision trees, if there exist indication rules constraining the value v of question Q in their rule condition. In such a case, a warning should be reported and the rules should be presented to the user.

Mechanics

The restructuring method is performed by the following procedure:

1 Apply test suite to the knowledge system and abort, if errors are reported.
2 Select the choice question $Q \in \Omega_Q$, the value $v \in dom(Q)$, and the choice question $Q' \in \Omega_Q$.
3 Add value v to the value range $dom(Q')$ and remove the value from $dom(Q)$.
4 Adapt the available test knowledge, e.g., modify test cases containing findings $Q{:}v$.
5 Modify the knowledge attached with question Q and value v During the modification check for conflicts as described in the *Consequences* section.

6 Apply the test suite to the restructured knowledge system and cancel the restructuring method, if errors are reported; alternatively start a debug session.

Example

The value ”knocking engine noise” (ne) of the one-choice question ”general observations” (GO) should be moved to the multiple-choice question ”engine noises” (EN). Originally, the following rules exist:

$$\begin{aligned} r_1 &= \mathit{and}\big(\mathit{choiceEqual}(GO, en), \mathit{choiceEqual}(EN, \mathit{knocking})\big) \rightarrow \mathit{Bad\ ignition\ timing} \\ r_2 &= \mathit{choiceEqual}(GO, en) \rightarrow \mathit{indicate}(EN) \end{aligned}$$

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

$$\begin{aligned} r'_1 &= \mathit{and}\big(\mathit{choiceEqual}(EN, en), \mathit{choiceEqual}(EN, \mathit{knocking})\big) \rightarrow \mathit{Bad\ ignition\ timing} \\ r'_2 &= \mathit{choiceEqual}(EN, en) \rightarrow \mathit{indicate}(EN) \end{aligned}$$

Since ”EN” is a multiple-choice question the rule r'_1 will not cause an ambivalent rule condition (as is would be in the case of an one-choice question). Rule r'_2 describes a cyclic indication rule and is presented to the developer to be resolved manually.

Related Methods

n/a

D REMOVEDIAGNOSIS

A diagnosis is removed from the knowledge base.

Motivation

Due to a specialization or generalization of the diagnosis hierarchy a particular diagnosis has become redundant. Alternatively, it has turned out that a diagnosis is never used in real world environment.

Consequences

The deletion of a diagnosis can cause conflicts in the knowledge base that can be resolved by default values.

Knowledge with Rule Conditions

We have to consider three alternatives when removing a diagnosis D from a rule condition:

1 Remove any rule that contains D in its rule condition.
2 Only remove that sub-condition of the rule that contains D. If D is contained in the only sub-condition defined by the rule, then this is equivalent to Alternative 1.
3 Interactively browse the rules that contain the diagnosis D and the developer decides manually, if the entire rule or sub-condition should be removed.

For the discussed alternatives the developer has to define a default value during the development of the knowledge system.

Structural Knowledge: Categorical/Scoring Rules

All categorical rules and scoring rules deriving a state of diagnosis D are removed from the knowledge base.

Structural Knowledge: Cases-Based Knowledge

For any case c and case solution $\mathcal{D}_c$, respectively, delete the diagnosis D. If the solution of c only consists of the diagnosis D, then remove the entire case. The deletion of a diagnosis can cause deficiencies of the case base, i.e., the deletion of a diagnosis or case can yield deficiency of the case base, redundant and ambivalent cases (p. 77).

Structural Knowledge: Set-Covering Knowledge

All set-covering relations $r = D \rightarrow F$ with $F \in \Omega_{\mathcal{F}}$ are removed from the set-covering model.

Support Knowledge: Linked Content

If there exist support knowledge for the diagnosis D, then remove links to the content.

Mechanics

The restructuring method is performed by the following procedure:

1 Apply test suite to the knowledge system and abort, if errors are reported.
2 Select the diagnosis $D \in \Omega_{\mathcal{D}}$ to be removed from the knowledge base.
3 Adapt the available test knowledge with respect to the removed diagnosis.
4 Check for conflicts as discussed in the *Consequences* section, and abort if conflicts are reported.
5 Remove the diagnosis D and adapt corresponding knowledge as described above.
6 Apply the test suite to the restructured knowledge system and cancel the restructuring method, if errors are reported; alternatively start a debug session.

Example

The diagnosis ”exhaust pipe” (EP) should be removed from the knowledge base, since it appeared to be not necessary. The following rules contain the diagnosis D:

$r_1 = established(EP) \rightarrow indicate(Q_1, Q_2)$
$r_2 = and(established(EP), choiceEqual(Q_3, v)) \rightarrow indicate(Q_4, Q_5)$
$r_3 = cond(r_3) \rightarrow EP$,

where $Q_i \in \Omega_Q$. The developer sets the default for removing only sub-conditions that contain the removed diagnosis. After the deletion of the diagnosis EP the rules r_1 and r_3 are removed and the rule r_2 is modified as follows:

$r_2 = and(choiceEqual(Q_3, v)) \rightarrow indicate(Q_4, Q_5)$

The rule condition can be further simplified by removing the *and* condition.

Related Methods

n/a

$\boxed{\text{D}}$ REMOVEQUESTION

A question is removed from the knowledge base.

Motivation

A question may have become obsolete due to simplifications of the knowledge base. Alternatively, a question should be removed because it turned out that is was never used in the real world application of the knowledge system.

Consequences

Although the deletion of a question appears as an easy task at first sight, this restructuring operation may cause conflicts in the knowledge base.

Knowledge with Rule Conditions

We have to consider three alternatives when removing a question Q from a rule condition:

1 Remove any rule that contains Q in its rule condition.
2 Only remove that sub-condition of the rule that contains Q. If Q is contained in the only sub-condition defined by the rule, then this is equivalent to Alternative 1.
3 Interactively browse the rules that contain Q and the developer decides manually, if the entire rule or sub-condition should be removed.

For the discussed alternatives the developer has to define a default value during the development of the knowledge system.

Structural Knowledge: Abstraction Rules

All abstraction rules that derive a value for Q are removed from the knowledge base.

Structural Knowledge: Cases-Based Knowledge

If available, then remove the weight function, the abnormality function, and the similarity function defined for Q. Consequently, for any case c and problem description $\mathcal{F}_c$, respectively, delete the finding $Q{:}v$. If the problem description of c only consists of the finding $Q{:}v$, then remove the entire case. However, the deletion of a question can cause deficiencies of the case base, i.e., the deletion of a finding or a case can yield redundant and ambivalent cases (p. 77).

Structural Knowledge: Set-Covering Knowledge

If available, then remove the weight function, the abnormality function, and the similarity function defined for Q. Furthermore, all set-covering relations $r = D \rightarrow Q{:}v$ with $D \in \Omega_{\mathcal{D}}$ are removed from the set-covering model.

Strategic Knowledge: Indication Rules

For any indication rule r containing Q in their indication action, i.e.,
$r = cond(r) \rightarrow indicate(Q_1, \dots, Q_k, Q, Q_k + 1, \dots, Q_n)$
remove Q from the rule action. If the rule action only contains the question Q, then remove the entire rule.

Support Knowledge: Linked Content

If there exists support knowledge for question Q, then remove links to the content.

Mechanics

The restructuring method is performed by the following procedure:

1 Apply test suite to the knowledge system and abort, if errors are reported.
2 Select the question $Q \in \Omega_Q$ to be removed from the knowledge base.
3 Adapt the available test knowledge with respect to the removed question.
4 Check for conflicts as discussed in the *Consequences* section, and abort if conflicts are reported.
5 Remove question Q and adapt linked knowledge as described above.
6 Apply the test suite to the restructured knowledge system and cancel the restructuring method, if errors are reported; alternatively start a debug session.

Example

The question ”car color” (CC) should be removed from the knowledge base, since it appeared to be not necessary. The following rules are using the question Q:
$r_1 = cond(r_1) \rightarrow indicate(exhaust\,pipe, CC)$
$r_2 = cond(r_2) \rightarrow indicate((CC))$
$r_3 = choiceEqual(Type, boxster) \rightarrow CC{:}silver$
$r_4 = or(choiceEqual(CC, silver), choiceEqual(CC, black),$
$choiceEqual(engine, boxer)) \rightarrow D\,,$
where $D \in \Omega_{\mathcal{D}}$ and ”exhaust pipe” $\in \Omega_Q$. The developer sets the default for removing only sub-conditions that contain the removed question. After the deletion of CC the rules are modified as follows:
$r'_1 = cond(r_1) \rightarrow indicate(exhaust\,pipe)$
$r'_4 = or(choiceEqual(engine, boxer)) \rightarrow D$
The indication rule r_2 was removed from the knowledge base since CC was the only indicated question. Furthermore, the abstraction rule r_3 was removed from the rule base. The rule condition of r'_4 can be simplified by removing the redundant *or* condition.

Related Methods

n/a

ShrinkValueRange $\boxed{\text{I}}$

Decrease the value range of a choice question in order to scale down the granularity.

Motivation

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

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.

Conflicts can be caused due to the mapping to a smaller value range. To detect conflicts, the applied knowledge containers are investigated. For the particular containers the following conflicts can arise:

Knowledge with Rule Conditions

Creation of identical sub-conditions: Due the restructuring two rules are generated with equal sub-conditions.

Choice question, *or* condition : $\circlearrowleft$

For arbitrary choice-questions, the rule condition can contain an *or* condition of two equal sub-conditions that were originally referring to different choice values and have been mapped to the same value. One of the two equal sub-conditions is deleted automatically in order to resolve this conflict.

MC question, *and* condition : $\circlearrowleft$

If the restructured question is a multiple-choice question, then the rule condition can contain an *and* condition of two equal sub-conditions targeting the equal transformed choice value. This conflict is automatically resolved by removing one of the equal sub-conditions.

MC question, *minMax* condition : $\circlearrowleft_d$

Two equal sub-conditions are generated due to the restructuring. When performed automatically the double-entry remains in the condition. In a later step, the developer has to decide if one condition should be deleted or adapted with respect to the *minMax* boundaries.

Structural Knowledge: Abstraction/Categorical/Scoring Rules

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

Redundant rules (p. 96) : $\circlearrowleft$

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

Ambivalent rules (p. 97) : $\ominus$

> The restructuring method is canceled if ambivalent rules are created, i.e., rules with equal rule condition but ambivalent rule action.
>
> **Abstraction rules** A different value of an equal question is derived by two rules with the same condition.
>
> **Categorical rules** A diagnosis is established and excluded by two rules with the same condition.
>
> **Scoring rules** A diagnosis is attached with a positive and negative confirmation category by two rules with the same condition.

Structural Knowledge: Case-Based Knowledge

Local similarity knowledge and abnormality functions need to be adapted according to the transformation function. Restructured cases can cause redundant and ambivalent cases.

Case with redundant findings : $\circlearrowleft$

> A case contains a multiple-choice question, which is assigned to two equal values. Then, one value is redundant and can be deleted automatically.

Redundant cases (p. 77) : $\circlearrowleft_d$

> Two cases c, c' have a subsuming set of findings and an equal set of diagnoses, i.e., $\mathcal{F}_c \subseteq \mathcal{F}_{c'}$ and $\mathcal{D}_c = \mathcal{D}_{c'}$. Per default, the cases remain in the case base, and the developer has to decide manually about the deletion.

Ambivalent cases (p. 78) : $\circlearrowleft_d$

> Two cases c, c' have a subsuming set of findings but a different set of diagnoses, i.e., $\mathcal{F}_c \subseteq \mathcal{F}_{c'}$ and $\mathcal{D}_c \neq \mathcal{D}_{c'}$. Per default, the cases remain in the case base. However, for specialized case bases, e.g., test cases, ambivalence denotes a semantic contradiction, and therefore this conflict can cause the method to be canceled.

Structural Knowledge: Set-Covering Knowledge

Local similarity knowledge and abnormality functions need to be adapted according to the transformation function. Restructured set-covering models can yield redundant and ambivalent set-covering relations.

Redundant set-covering relation (p. 120) : $\circlearrowleft_d$
: A set-covering relation $r = D \rightarrow F$ was created, that already exists in the set-covering model.

Subsumed set-covering relation (p. 120) : $\circlearrowleft_d$
: A set-covering relation $r = D \rightarrow F$ was created, that subsumes or that is subsumed by another existing set-covering relation.

Ambivalent set-covering relation (p. 120) : $\ominus$
: A set-covering relation $r = D \rightarrow F$ was created by the restructuring method, but there exists an exclusion condition $(\neg D \wedge F)$.

Strategic Knowledge: Indication Rules

For indication rules we have to consider the conflicts discussed in the section *Knowledge with Rule Conditions*.

Support Knowledge: Linked Content

Support knowledge linked by the values v_i of question Q is then linked by the values $t(v_i)$.

Mechanics

The restructuring method is performed by the following procedure:

1 Apply 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, and 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 $v' \in dom'(Q)$.
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 with 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 restructured knowledge system and cancel the restructuring method, if errors are reported; alternatively start a debug session.

Example

The one-choice question ”mileage” (M) with the value range $dom(M) = \{very\ low, low, normal, high, very\ high\}$ is too detailed and should be simplified by the value range $dom'(M) = \{low, normal, high\}$. The developer defines a transformation function given by the following table:

$dom'(M)$	$dom(M)$
very low	*low*
low	*low*
normal	*normal*
high	*high*
very high	*high*

Originally, the following rules with question M are contained in the knowledge base (with diagnosis ”*clogged air filter*” and question set QS):

$$\begin{aligned} r_1 &= or\big(choiceEqual(M, high), choiceEqual(M, very\ high)\big) \rightarrow clogged\ air\ filter \\ r_2 &= choiceEqual(M, high) \rightarrow indicate(QS) \\ r_3 &= choiceEqual(M, very\ high) \rightarrow indicate(QS) \end{aligned}$$

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

$$\begin{aligned} r'_1 &= or\big(choiceEqual(M, high), choiceEqual(M, high)\big) \rightarrow clogged\ air\ filter \\ r'_2 &= choiceEqual(M, high) \rightarrow indicate(QS) \\ r'_3 &= choiceEqual(M, high) \rightarrow indicate(QS) \end{aligned}$$

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:

$$\begin{aligned} r''_1 &= choiceEqual(M, high) \rightarrow clogged\ air\ filter \\ r''_2 &= choiceEqual(M, high) \rightarrow indicate(QS) \end{aligned}$$

Related Methods

n/a

TRANSFORMMCINTOYN

Convert a multiple-choice question into a set of semantically equal yes/no questions by mapping each value of the multiple-choice question to a yes/no question.

Motivation

The value range of a multiple-choice question contains answers with no semantical relation. With the TRANSFORMMCINTOYN method such a question can be converted into a set of yes/no questions. Each yes/no question corresponds to a value of the multiple-choice question. Furthermore, this method facilitates the move of the converted yes/no questions into different question sets afterwards.

Consequences

In general, the method cannot perform any conflicts, and we describe the particular modifications for the implemented knowledge. Let $Q \in \Omega_Q$ be a multiple-choice question with $v_i \in dom(Q)$, and Qv_i are generated yes/no questions.

Knowledge with Rule Conditions
Each condition $choiceEqual(Q, v_i)$ is replaced by the new condition $choiceEqual(Qv_i, yes)$.

Structural Knowledge: Abstraction Rules
If the value v_i is derived for question Q by an abstraction rule, i.e.,
$r = cond(r) \rightarrow Q{:}v_i$,
then the rule is replaced by
$r' = cond(r) \rightarrow Qv_i{:}yes$.

Structural Knowledge: Case-Based Knowledge
The global weight w_g and abnormality function of the generated yes/no question $Qv_i{:}yes$ is adapted with respect to the weight of Q and the abnormality of the value v_i, i.e., $w_g(Qv_i) = w_g(Q)$ and $abn(yes) = abn(v_i)$.
The findings of question Q in cases $c \in CB$ are replaced by the generated findings $Qv_i{:}yes$.

Structural Knowledge: Set-Covering Knowledge
Each set-covering relation
$r = D \rightarrow Q{:}v_i$
with $D \in \Omega_{\mathcal{D}}$ is replaced by a new set-covering relation
$r' = D \rightarrow Qv_i{:}yes$.
Global weight function is adapted so that $w_g(Q) = w_g(Qv_i)$.

Strategic Knowledge: Indication Rules

Each indication rule with Q in its rule action
$r = cond(r) \rightarrow indicate(Q'_1, \ldots, Q'_n,$ **Q**, $Q'_{n+1}, \ldots, Q'_m)$
is replaced by a new indication rule
$r = cond(r) \rightarrow indicate(Q'_1, \ldots, Q'_k,$ ***Qv***$_1, \ldots,$ ***Qv***$_n, Q'_{n+1}, \ldots, Q'_m)$ so that now the generated yes/no questions are indicated.

Support Knowledge: Linked Content

Support knowledge linked by the question Q is then linked by the generated yes/no questions $Qv_1, \ldots, Qv_n$.

Mechanics

The restructuring method is performed by the following procedure:

1 Apply test suite to the knowledge system and abort, if errors are reported.
2 Select the multiple-choice question Q with the value range $dom(Q) = \{v_1, \ldots, v_n\}$.
3 Insert n yes/no questions mit names corresponding to the values $Qv_1, \ldots, Qv_n$ at the same position of the original question Q.
4 Adapt test suite with respect to the new yes/no questions.
5 Select the knowledge that contains the multiple-choice question Q and replace Q by the new yes/no questions $Qv_1, \ldots, Qv_n$ according to the following schema: Replace all Q:v_i by the new finding Qv_i:yes.
6 Delete the multiple-choice question Q.
7 Apply the test suite to the restructured knowledge system and cancel the restructuring method, if errors are reported; alternatively start a debug session.

Example

The multiple-choice question ”engine noises” (E) with the value range $dom(E) = \{knocking, ringing, normal\}$ should be converted to the three yes/no questions ”engine noises-knocking” ($E\_knocking$), ”engine noises-ringing” ($E\_ringing$), and ”engine noises-normal” (E_normal). The following rules are contained in the knowledge base with $D \in \Omega_{\mathcal{D}}$:

$$\begin{aligned} r_1 &= cond(r) \rightarrow indicate(E) \\ r_2 &= and\big(choiceEqual(E, ringing), \\ &\qquad not\big(choiceEqual(E, normal)\big)\big) \rightarrow D \end{aligned}$$

After the transformation the rules are modified as follows:

$$\begin{aligned} r'_1 &= cond(r) \rightarrow indicate(E\_knocking, E\_ringing, E\_normal) \\ r'_2 &= and\big(choiceEqual(E\_ringing, yes), \\ &\qquad not\big(choiceEqual(E\_normal, yes)\big)\big) \rightarrow D \end{aligned}$$

Related Methods

TRANSFORMYNINTOMC (inverse).

$\boxed{\text{I}}$ TRANSFORMNUMINTOOC

A numerical question is converted into a more abstract choice question.

Motivation

At the beginning of system development the numerical representation of a question was assumed to be reasonable. Later the numerical question was felt to be too detailed or an exact numerical value could not be gathered. As a consequence, a more qualitative representation of the findings by an one-choice question is preferred. Another motivation for this restructuring may be the requirement to reduce the time the user spends answering these questions, since qualitative values for a question typically are easier and faster to acquire.

Consequences

The described method is only applicable to standard questions that are not derived by abstraction rules. Furthermore, the method can only be applied if the value range of the numerical question can be divided into disjunctive partitions. For each partition a new choice value is created. The generation of the partitions is defined as follows: If the numerical question is constrained to a specified value, e.g., in a rule condition, then a partition is defined containing exactly this value. If the numerical question is constrained to an interval, then one or more (smaller) partitions are generated that cover the constrained interval. The generated partitions are mapped to choice values, i.e., for a point interval $[x;x]$ we generate the choice value v_x, and for the interval $[x,y]$ we generate the choice value $v_{[x,y]}$. Consequently, the generated value range is an ordered sequence. It is worth noticing, that often a post-processing of the generated partitions is reasonable, e.g., by defining a more coarse value range. However, the adaptation should be performed after a successful implementation of the restructuring method.

Knowledge with Rule Conditions

For all rule conditions containing the numerical question Q we convert the rule condition according to the following schema; Q' is the new choice question with value range $dom(Q') = \{v_0, \ldots, v_n\}$.

$$\begin{array}{ll}
numEqual(Q,x) & choiceEqual(Q', v_x) \\
numLess(Q,x) & choiceIn(Q', \{v_i, v_{i-1}, \ldots, v_0\}) \\
 & \quad \text{with } v_i = v_{(x,y]} \\
numLessEqual(Q,x) & choiceIn(Q', \{v_i, v_{i-1}, \ldots, v_0\}) \\
 & \quad \text{with } v_i = v_{[x,y]}
\end{array}$$

$numGreater(Q, x)$	$choiceIn(Q', \{v_i, v_{i+1}, \ldots, v_n\})$ with $v_i = v_{(x,y]}$
$numGreaterEqual(Q, x)$	$choiceIn(Q', \{v_i, v_{i+1}, \ldots, v_n\})$ with $v_i = v_{[x,y]}$
$un/known(Q)$	$un/known(Q')$

Structural Knowledge: Abstraction Rules
If the numerical question is contained in the rule action of an abstraction rule, then the method is aborted.

Structural Knowledge: Case-Based Knowledge
If the numerical question Q is contained in the problem description of a case c, then it is replaced by the choice question Q' according to the following schema: For Q:$x \in \mathcal{F}_c$ we insert Q':v_x, if there exists a point interval $v_x \in dom(Q')$. Otherwise, we insert Q':$v_{[m,n]}$ with the value $v_{[m,n]} \in dom(Q')$ and $x \in [m, n]$.

Structural Knowledge: Set-Covering Knowledge
If the numerical question Q is contained in a set-covering relation, then it is replaced according to the schema defined for rule conditions.

Mechanics

The restructuring method is performed by the following procedure:

1 Apply test suite to the knowledge system and abort, if errors are reported.
2 Select a numerical question $Q \in \Omega_Q$ and generate distinct partitions of the value range.
3 Abort method, if no distinct partition can be generated.
4 Insert one-choice question Q' after the position of Q.
5 Adapt test suite with respect to the partitioning information.
6 Select the knowledge that contains the selected numerical question Q and replace it with Q'. Adapt knowledge according to the discussion in the *Consequences* section.
7 Delete the numerical question Q.
8 Apply the test suite to the restructured knowledge system and cancel the restructuring method, if errors are reported; alternatively start a debug session.

Example

The numerical question ”average milage” (NM) should be converted into the choice question ”average mil.” (CM). The following categorical rules exist with $D, D', D'' \in \Omega_\mathcal{D}$:
$r_1 = numGreater(NM, 15) \rightarrow D$
$r_2 = numLessEqual(NM, 5) \rightarrow D'$
$r_3 = numEqual(NM, 10) \rightarrow D''$

The partition $\{[-\infty, 5], (5, 10), [10, 10], (10, 15], (15, \infty)\}$ is generated automatically, and consequently the value range
$dom(Q') = \{v_{[-\infty,5]}, v_{(5,10)}, v_{[10,10]}, v_{(10,15]}, v_{(15,\infty)}\}$.
The rules are converted according to the discussion in the *Consequences* section:
$r'_1 = \mathit{choiceIn}(CM, \{v_{(15,\infty)}\}) \rightarrow D$
$r'_2 = \mathit{choiceIn}(CM, \{v_{(-\infty,5]}\}) \rightarrow D'$
$r'_3 = \mathit{choiceEqual}(CM, v_{[10,10]}) \rightarrow D''$
The conditions of rule r'_1 and r'_2 can be simplified in a post-processing step by exchanging $\mathit{choiceIn}$ by $\mathit{choiceEqual}$.

Related Methods

n/a

TransformYNIntoMC

I

A sequence of yes/no questions is converted into one semantically equivalent multiple-choice question. The name of each yes/no question corresponds to a value of the generated multiple-choice question.

Motivation

The number of yes/no questions can be reduced by aggregating them in a single multiple-choice question. Such a transformation can yield a more compact representation of the ontological knowledge. Furthermore, the dialog efficiency can be improved since typically answering one multiple-choice question takes less time than a set of yes/no questions.

Consequences

The method only causes conflicts for strategic knowledge with ambivalent indication rules. We describe the particular modifications for the implemented knowledge. Let $\{Qv_1, \ldots, Qv_n\} \subseteq \Omega_Q$ be a set of yes/no questions and let Q be the generated multiple-choice question with corresponding value range $dom(Q) = \{v_1, \ldots, v_n\}$.

Knowledge with Rule Conditions

Each condition $choiceEqual(Qv_i, yes)$ is replaced by the new condition $choiceEqual(Q, v_i)$; conditions $choiceEqual(Qv_i, no)$ are replaced by the new condition $not\big(choiceEqual(Q, v_i)\big)$.

Structural Knowledge: Abstraction Rules

The restructuring method is aborted, if there exists an abstraction rule deriving the no value of a selected yes/no question Qv_i. Such an abstraction cannot be converted to an abstraction rule for a multiple-choice question.

Otherwise, if the yes value is derived for question Qv_i by an abstraction rule, i.e.,
$r = cond(r) \rightarrow Qv_i{:}yes$, then the rule is replaced by
$r' = cond(r) \rightarrow Q{:}v_i$.

Structural Knowledge: Case-Based Knowledge

If the global weight function w_g is defined for the selected yes/no questions, then we need to assign an aggregated weight to the generated multiple-choice question Q, e.g., the mean value of the aggregated weights.
The findings $Qv_i{:}yes$ in cases $c \in CB$ are replaced by a finding of the generated multiple-choice question $Q{:}v_i$. If more than one finding for Q is contained in c, then these findings are aggregated to a common multiple-choice finding.

Structural Knowledge: Set-Covering Knowledge

Each set-covering relation
$r = D \rightarrow Qv_i{:}yes \quad / \quad r = D \rightarrow Qv_i{:}no$
with $D \in \Omega_{\mathcal{D}}$ is replaced by a new set-covering relation
$r' = D \rightarrow Q{:}v_i \quad / \quad r' = D \rightarrow not(Q{:}v_i)$.
The global weight function w_g for the question Q is defined by the aggregated global weights of the converted yes/no questions, e.g., by the mean value.

Strategic Knowledge: Indication Rules

The method replaces the selected yes/no questions Qv_i contained in indication rules by the generated multiple-choice question Q. The restructuring method is aborted, if there exist two rules with an equal rule condition and Qv_k, Qv_m are selected yes/no questions:
$r_1 = c \rightarrow indicate(Qv_k)$
$r_2 = c \rightarrow \neg indicate(Qv_m)$
After the transformation we obtain the following ambivalent indication rules:
$r_1' = c \rightarrow indicate(Q)$
$r_2' = c \rightarrow \neg indicate(Q)$

Support Knowledge: Linked Content

Support knowledge linked by a yes/no question Qv_i is then linked by the generated multiple-choice question Q.

Mechanics

The restructuring method is performed by the following procedure:

1 Apply test suite to the knowledge system and abort, if errors are reported.
2 Select the list of yes/no questions $(Qv_1, \ldots, Qv_n)$ to be transformed.
3 A multiple-choice question Q with value range $dom(Q) = \{v_1, \ldots, v_n\}$ is inserted at the position of the first yes/no question Qv_1.
4 Adapt test suite with respect to the new multiple-choice question.
5 Select the knowledge that contains the selected yes/no questions Qv_i and replace Qv_i by the generated yes/no question Q according to the following schema: Replace all $Qv_i{:}yes$ by $Q{:}v_i$, and $Qv_i{:}no$ by $not(Q{:}v_i)$.
6 Delete the yes/no questions $Qv_1, \ldots, Qv_n$.
7 Apply the test suite to the restructured knowledge system and cancel the restructuring method, if errors are reported; alternatively start a debug session.

Example

The selected yes/no questions ”engine knocking” ($E\_knocking$), ”engine ringing” ($E\_ringing$), and ”engine noise normal” (E_normal) should be converted in a single

multiple-choice question ”engine noise” (E). Consequently, the value range of E is given by the yes/no questions, i.e.,
$dom(E) = \{e\_knocking, e\_ringing, e\_normal\}$.

The following rules are contained in the knowledge base with $D \in \Omega_{\mathcal{D}}$:

$$\begin{aligned} r_1 &= cond(r) \rightarrow indicate(E\_knocking, E\_ringing) \\ r_2 &= and\big(choiceEqual(E\_knocking, yes), \\ &\qquad choiceEqual(E\_normal, no)\big) \rightarrow D \end{aligned}$$

After the transformation the rules are modified as follows:

$$\begin{aligned} r'_1 &= cond(r) \rightarrow indicate(E) \\ r'_2 &= and\big(choiceEqual(E, e\_knocking), \\ &\qquad not\big(choiceEqual(E, e\_normal)\big)\big) \rightarrow D \end{aligned}$$

Related Methods

TRANSFORMMCINTOYN (inverse).

B. Test Methods in a Nutshell

One of the key practices of the agile process model introduced in this work is the application of automated test methods. In this appendix we summarize the presented test methods in a tabular form. For each test method we give the name of the test method and a brief description of the method's purpose (*Test Name and Description*). Furthermore, the targeted knowledge container of the test (*Container*) and the required test knowledge (*Test Knowledge*) is given.
The targeted knowledge containers are abbreviated according to the following notation:

OntK	The ontological knowledge container
StrucK	The structural knowledge container
StrucK/CBR	The structural knowledge container with case-based knowledge
StrucK/SCM	The structural knowledge container with set-covering knowledge
StrucK/RB	The structural knowledge container with rule-based knowledge
StratK	The strategic knowledge container
SuppK	The support knowledge container

For a more detailed discussion of the particular methods we refer to the given page numbers. The test methods are listed in alphabetical order.

Test Methods

Test Name and Description	Container	Test Knowledge
Ambivalent Link Testing: Tests, if the linkage of the ontological objects has an ambivalent meaning. (p. 150)	SuppK	Ontological knowledge
Case-Based Ontology Testing: Determines the usage of the ontological objects under real world conditions. (p. 54)	OntK	Cases
Case-Based Structure Testing: Gives an overview of the current case base w.r.t. appearance of diagnoses and questions. (p. 78)	StrucK/CBR	–
Diagnosis-Related Question Sets: Tests the correctness of strategic knowledge for given diagnoses. (p. 141)	StratK	DRQS, cases, structural knowledge
Dynamic Rule Base Testing: Determines the usage of rules w.r.t. to a given case base. (p. 99)	StrucK/RB	Cases, threshold values
Empirical Testing: Compares the derived solutions of solved test cases with the correct solutions of the given cases. (p. 125)	StrucK	Cases, threshold values
Inferential Constraints: Tests the local correctness of structural knowledge. (p. 129)	StrucK	Partial cases
Partial-Ordered Question Sets: Tests the correctness of strategic knowledge using typical dialog sequences. (p. 140)	StratK	POQS, possibly cases
Plain Link Testing: Tests for all ontological objects, if the linked content is accessible. (p. 149)	SuppK	–
Sequenzialized Empirical Testing: Compares the derived and the intermediate solutions of test cases with the correct solutions of the given cases. (p. 128)	StrucK	Sequentialized cases, threshold values
Standardization Testing: Checks the ontology against a given standard ontology. (p. 55)	OntK	Standardized ontology
Static Link Testing: Provides an overview of the available links defined for the support knowledge entries. (p. 149)	SuppK	–
Static Ontology Testing: Generates a statistics of the implemented ontological entities. (p. 53)	OntK	–
Static Rule Base Testing: Gives an overview of the implemented rule base. (p. 98)	StrucK/RB	–
Static SC-Model Analysis: Gives a statistics of the implemented set-covering model. (p. 119)	StrucK/SCM	Threshold values
Static SC-Model Verification: Tries to detect anomalies in set-covering knowledge. (p. 119)	StrucK/SCM	Possibly threshold value

continued on next page

Test Name and Description	Container	Test Knowledge
Static Verification (Case Base): Tries to detect anomalies in case-based knowledge. (p. 77)	StrucK/CBR	Threshold values
Static Verification of Rule Base Integrity: Investigates the rule base for anomalies. (p. 96)	StrucK/RB	Possibly ambivalence constraints
Test Case Duration: Determines the averaged dialog duration of the implemented knowledge system. (p. 142)	StratK	Cases, benchmark value
Torture Tests: Determines the robustness of the implemented knowledge system w.r.t. the input quality and the input quantity. (p. 130)	StrucK	Cases, threshold values
Torture Tests (Rule Base): Determines the robustness of the implemented rule base. (p. 98)	StrucK/RB	Cases, threshold values
Torture Tests (SCM): Determines the robustness of the implemented set-covering model. (p. 121)	StrucK/SCM	Cases, threshold values

Index