# An Agile Process Model for Developing Diagnostic Knowledge Systems

Joachim Baumeister, Frank Puppe, Dietmar Seipel

The development of knowledge systems has been investigated since the seventies and many process models have been proposed to reduce knowledge acquisition costs and to increase system quality. Recently, agile process models, e.g., eXtreme programming, have discussed extensively in the software engineering community. They appeared to be appropriate for small-sized teams and proved to be very successful in various projects. In this paper, we introduce an agile process model for developing diagnostic knowledge systems, which can provide an alternative to well-accepted methodologies like CommonKADS, e.g., if the team size is small or specifications are vague.

## 1 Introduction

The development of diagnostic knowledge systems has been investigated since the seventies and has experienced changing acceptance by industry and research. Today, diagnostic knowledge systems are well established and have been deployed in various problem domains, e.g., in medical, technical or service-support domains.

However, the development and maintenance of such systems is still a difficult task for the following reasons:

- the systems are costly to develop, since in most projects the required time and resources are not predictable in advance.
- systems show unexpected and faulty behavior when applied in routine use
- sometimes the systems do not fulfill the customers' requirements and expectations
- many systems appeared to be hardly maintainable, when delivered in routine use

A great deal of research has been done to cope with the problems stated above. According to the experiences faced in software engineering practice process models have been investigated to introduce a more structured and approved development of knowledge systems; examples are the COMMON-KADS [Sch:01] or the MIKE approach [Ang:98]. For an extensive survey and discussion of knowledge engineering methodologies we refer to [Stu:98, PG:03].

However, for small-size or mid-size development projects such document-centered approaches seem to be too cumbersome and costly for the domain experts. Verbose specifications and necessary decisions about the project design often deter experts from starting or continuing a knowledge system project. In fact, we have experienced experts to be motivated, if early results can be taken out from small specifications and, if systems could grow incrementally from a small pilot.

In this paper we present a novel process model for developing diagnostic knowledge systems, which was inspired by the agile process model *eXtreme programming* (XP). In software engineering research and practice XP [Bec:00] has attracted huge attention and discussion, and showed its significance in numerous projects.

The presented 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

In contrast to XP, which considers the process of implementing general software using an all purpose programming language (e.g., C or Java), the presented agile process model is focused on the development of knowledge systems, that are declaratively build by the insertion, the change, and the review of knowledge represented in a declarative language. Then, a detailed object design corresponds to an abstract design at the knowledge level. It is easy to see that an appropriate knowledge modeling environment is needed to support the agile development of knowledge systems.

The rest of this paper is organized as follows: In Section 2 we introduce the agile process model as a cycle of steps and we will explain these steps in detail. Each iteration considers the extension or redesign of knowledge, and in Section 3 we briefly discuss the concept of knowledge containers, which can simplify this process. The efficiency and value of a process model strongly depends on its support by adequate tools. In Section 4, we present the knowledge modeling and management environment d3web.KnowME, which provides a visual and easy-to-use interface for knowledge developers. Section 5 concludes the paper with a description and a discussion of early experiences we have made with the process model in the ECHODOC project.

## 2 The Agile Process Model

In this section we will introduce an agile process model for developing diagnostic knowledge systems. The agile process model is a light-weight process model and consists of the following cyclic steps depicted in Figure 1: Analysis of the system metaphor, Design of the planning game, Implementation (plan execution: including tests, restructuring and maintenance), and Integration.
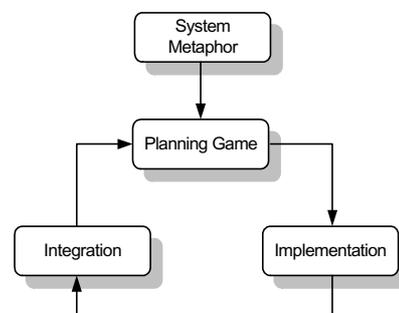


Figure 1: The steps of the agile process model.

A new knowledge system project starts with the analysis of the system metaphor, which should include an overall plan of the intended 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 process model is executed during the development phase and lasts as long as the system lasts.

The particular steps and terms were taken from the XP methodology, and in various projects it has been proven that they describe an appropriate way of developing software. In the context of the presented process model we have adapted these steps with respect of the requirements of developing a knowledge system.

### 2.1 The System Metaphor

The system metaphor describes the basic idea and designated aims of the knowledge system to be implemented. It is used as a means of communication between the developer and the

user 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.

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

### The 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 inferring solutions, which explain the given input. The local metaphor consists of a set of object types for the basic entities and we see, that there exists 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):* The class of entities, which are mostly given as input to the diagnostic system, and which 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 / case:* A problem is defined as a set of findings co-occurring in a specific situation. The instance of a case consists of a problem and a solution of the problem. Furthermore, a set of additional information (mostly unstructured and informal) can be attached to a case describing the solved case in more detail.

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

### The Global System Metaphor

Beyond the basic entities for 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 will describe the four most typical application classes:

*Documentation System.* A documentation system focuses on a high quality data acquisition and usually implements a detailed dialog control. This kind of class is often implemented, when a high quality data entry is the most important feature of the knowledge system. In this context, the term quality is defined by the minimal number of required findings to gather, the 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.

*Embedded System.* Sometimes the diagnostic knowledge system is embedded into a larger context. Then, the system is tightly connected to another application, from which it receives its findings and to which it reports the derived diagnoses. Because usually data is not entered manually, often no dialog control is needed. Embedded systems are also called closed-loop systems.

*Consultation System.* Consultation systems are the typical kind of a diagnostic knowledge system: The user is guided through a structured dialog in order to enter findings for the given problem. Then, the system applies the findings to infer diagnoses

which are presented as solutions to the user. We can see, that a consultation system can be combined with a documentation system, although the focus of the data entry may vary.

*Information Center.* The information center does not consider a well elaborated diagnostic inference (e.g., as needed for a consultation system), but mostly consists of informal knowledge like documents, multimedia content or structured cases. Usually, this content is highly 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 system in order to find helpful content for a given problem. In comparison to a consultation system the user is more flexible and independent, but also responsible for obtaining a suitable solution for his problem.

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.

## 2.2 The Planning Game

In summary, the overall aim of the planning game is to maximize the value and to minimize the development costs of the system to be built. The value is mainly defined by the consumer satisfaction derived from the usability, the functionality, and the correctness of the system. The main purpose of the planning game is to decide about the scope and the priority of future development. Furthermore, costs of the planned implementation are estimated and the development plans are scheduled. It is also useful to provide a benchmark for feedback in order to enable adaptation of plan estimation in the future. In principle, it is not advisable to make detailed long-range plans, because the priorities of intended functionality can change or customer requirements can evolve during the project. Thus, it is suggestive to discuss long-range plans only as coarse artifacts and to 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 user. 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.

It is worth noticing, that plans do not only carry out new functionality of the knowledge system, but they can extend 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.* The user and the developer think about the system functionality that should be changed or added. For each functionality a story is written down describing the task in more detail. For this we use story cards as described above. Then, the development costs of each story are estimated and stories are partitioned into smaller grained stories, if necessary.

*Commitment.* The user and the developer decide about the realization of the stories recorded during the exploration. The user assigns a priority to each story with possible values "essential", "significant", and "nice to have". The developer sorts the stories independently by their implementation risks with the possible values "certain", good", and "unsafe". Based on these estimations the developer and the user 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 does not always develop as expected. Therefore, the steering phase updates and refines the plan currently under implementation. The steering phase offers three moves:

- *Iteration:* During the implementation of the plan stories are picked iteratively, i.e., after the 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 velocity. 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 user and the developer need to decide about the most valuable set of stories, which should remain in the current plan and which cannot be moved to a later planning game.
- *Re-estimate:* In general, during the implementation of the plan the developers can re-estimate the remaining stories, if the plan no longer provides a sufficiently precise table of the development.

The planning game provides a flexible method for guiding the development process of knowledge systems. On the one side, 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 side, the steering phase enables the process to flexibly adapt the currently implemented plan by inventing new stories, by re-estimating the predicted plan costs or by reordering the priority of the remaining unimplemented stories (iteration). Furthermore, an accurate feedback can be given to assess the whole development process by providing methods for estimating and documenting the implementation costs (derived from the implementation velocity).

## 2.3 The 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 splits the story into distinct tasks, which are sequentially handled. A *task* is defined as a 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 the 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. Both phases are favorably performed using powerful tools as described in Section 4.

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 for making the instant validation of the knowledge system possible. In contrast to general programming languages, it is easier to define tests for declarative knowledge. Some tests can be even applied without an additional test specification, e.g., testing for anomalies.

In the *code-implementation phase* the developer acquires and implements the new functionality described in the task, e.g., by adding new knowledge or restructuring existing knowledge. 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: A reason for failing tests is obviously an error in the new implementation. But often old tests also need to be adapted according to newly implemented 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 system passes the tests, then the developer and the user 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 in conjunction with the user will clarify story or task definitions. Thereby, often ambiguous definitions 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 before completing the implementation of a story. Thus, side effects can easily be discovered, i.e., a new functionality has accidentally changed the behavior of a previously implemented functionality.

## 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 onto 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. Since the integration is done continuously, we always can access a running system at the integration unit.

To guarantee the practicability of the integration builds an integration test suite ensures that the system is not broken because of the new functionality. For a reasonable integration additional tests need to be available, which often are too time consuming to include them into the working test suite, but are applied during the 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. Running a number of 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 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.

## 3 Knowledge Containers and the Agile Process Model

In general, the presented process model can be applied to arbitrary classes of knowledge systems, e.g., diagnostic systems and configuration systems. Thus, each iteration of the process considers the extension or redesign of existing knowledge. Since this procedure is a complex and difficult task we introduce the concept of knowledge containers. We partition the knowledge classified according to its use into four parts, which we call *knowledge containers*. For diagnostic knowledge systems we can identify the following four containers:

The *Ontological Knowledge* consists of the basic entities used in the implemented system, e.g., the specific findings as input parameters, grouping question sets, and the diagnoses to be inferred by the system. These entities can be structured hierarchically with respect to their semantical relationships.

The *Structural Knowledge* is used for deriving a solution for a given input. In the past, 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* is applied for controlling the user dialog of the knowledge system. To avoid needless data acquisition costs, the user is typically guided through a dialog path, which is appropriate for the problem entered by the user (goal directed to find a solution). Often, strategic knowledge is implemented by indication rules, which activate the questionary of specified findings or question sets according to the occurrence of other findings or already hypothesized solutions.

The *Support Knowledge* is used to supply ontological knowledge with additional information, typically consisting of text book entries or multimedia content (e.g., pictures, movies) used for explanation.

For an agile application of the knowledge containers, appropriate test and restructuring methods need to be available, which depend of the used knowledge representation. In [BSP:04] a discussion of test and restructuring methods in the context of the agile process model is given.

It is easy to see, that other classes of knowledge systems, e.g., configuration systems, may imply a different categorization of knowledge containers, which however is not the scope of this paper.

The concept of knowledge containers can simplify the development in several ways:
- they provide an organized view of the whole knowledge base
- they simplify the maintenance and the validation of the knowledge base, since for each container appropriate test and restructuring methods can be defined
- they simplify the planning game due to a better identification of the intended knowledge extension
- they focus on the intended usage of the included knowledge in contrast to the actual knowledge representation

The classification of knowledge with respect to its usage goes back to Clancey [Cla:83] introducing the terms *structural knowledge*, *strategy knowledge* and *support knowledge*. We apply this classification with nearly the same meaning, but postulate for each container a distinct and explicit representation attached with appropriate test and restructuring methods. In our framework we further will separate *ontological knowledge* from structural knowledge, since ontological knowledge is also applied in the remaining knowledge containers. Thus, all knowledge containers are independent from each other with the exception of the ontological container, which is the basis of all remaining containers. Furthermore, the presented framework is not restricted to a rule-based representation of knowledge.
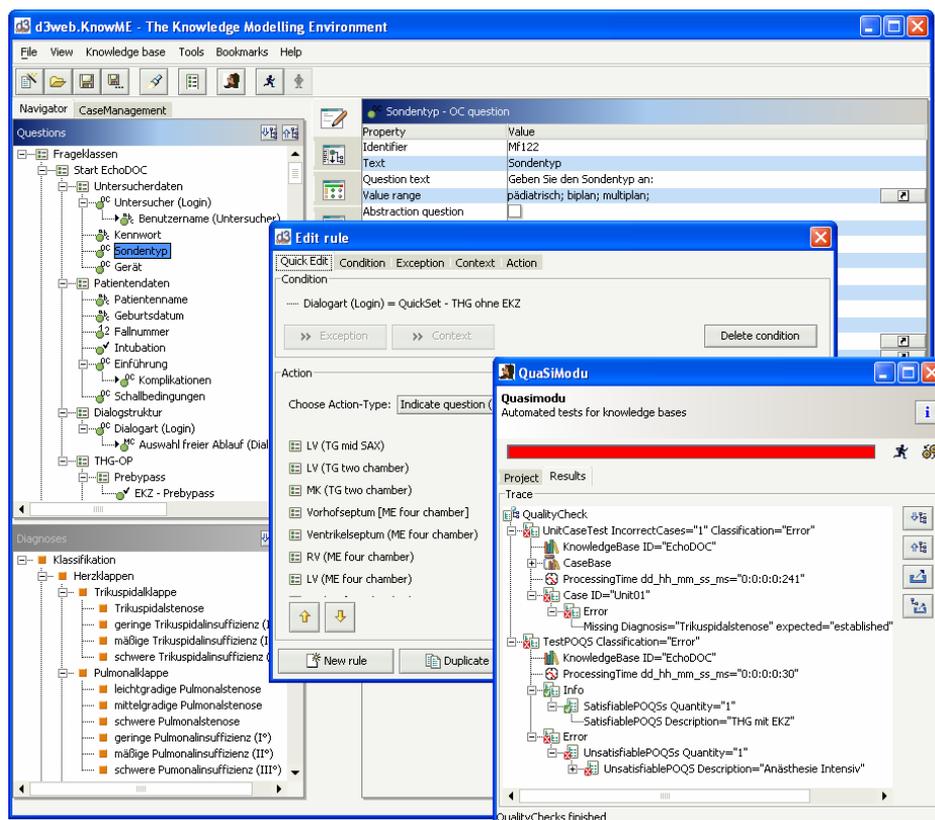


Figure 2: The knowledge modeling environment d3web.KnowME.

## 4 Agile Knowledge Modeling with d3web.KnowME

As mentioned in the introduction the practical success of process models strongly depends on the availability of appropriate tools. In the following, we briefly discuss the knowledge modeling environment d3web.KnowME [d3web], which provides a highly integrated workbench for the agile development of diagnostic knowledge systems. d3web.KnowME (depicted in Figure 2) is the successor of the knowledge acquisition tool of D3 [Pup:98], which has been successfully applied in many medical, technical, and other domains.

The workbench offers visual editors for implementing ontological knowledge, structural knowledge (rules, set-covering models, cases), strategic knowledge (indication rules), and support knowledge (texts, hyperlinks). Furthermore, there exist editors for documenting the planning game, for editing corresponding test knowledge, and for implementing basic restructurings on knowledge.

For an agile development the integration of appropriate test methods is a significant requirement. Consequently, a tool for automated tests of the implemented knowledge is also included. Currently, methods are offered for empirical testing, i.e., running and comparing previously solved test cases, for static and case-based ontological testing, and for the static analysis of structural and strategic knowledge. 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 for any of the tests errors are reported, then the color of the status bar changes to red. This metaphor is known from JUnit, a well-known test framework for the programming language Java. The convenient analysis of a verbose result trace is provided by an export to MS-Excel format.

Besides this basic functionality d3web.KnowME also offers more sophisticated features, e.g., a visual debugger working with test cases and specialized editors for several types of structural knowledge.

## 5  Conclusions

We have introduced an agile process model for developing diagnostic knowledge systems. The process model was inspired by the eXtreme programming methodology known from the software engineering community. Furthermore, we have motivated and presented a framework for classifying knowledge into distinct containers, i.e., knowledge containers.

Recently, we have started an agile knowledge system project with the medical documentation system ECHODOC. Two domain experts are building the knowledge base themselves using the visual development environment. Typically, the experts are not working on the project full-time, and therefore the planning game with the resulting story cards is taken as a valuable organization help. The step-wise implementation of plans always resulting in a running system appeared to be very motivating for the experts. At the moment, the physicians have finished the strategic knowledge container as their first milestone, i.e., the development of a documentation system for high-quality examinations. Preliminary studies and an evaluation of ECHODOC have been undertaken. Furthermore, there exists a mid-range plan on extending the system by structural knowledge, which will result in a consultation system. For a more detailed introduction of the presented work, i.e., the agile process model together with evaluating project descriptions, we refer to [Bau:04].

Practical experiences show promising directions for future work: Developers should be supported by a simplified planning game, e.g., offering pre-defined templates of typical plans for the construction of a diagnostic system. Another open problem is the (semi-)automatic adaptation of test cases after the restructuring of the knowledge base. If basic entities of the knowledge base are modified, then often corresponding test cases become invalid and currently have to be adjusted manually. To the knowledge of the authors, software engineering research faces the corresponding problem, and it has not been solved in a sufficient manner.

## 6  References

[Ang:98] 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

[Bau:04] Joachim Baumeister: *Agile Development of Diagnostic Knowledge Systems*, PhD Thesis (submitted), University Würzburg, Germany, 2004

[Bec:00] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Longman Publishing Co., Inc., 2000

[BSP:04] Joachim Baumeister, Dietmar Seipel, and Frank Puppe: *Using Automated Tests and Restructuring Methods for an Agile Development of Diagnostic Knowledge Systems*, In Proc. of 17th International Florida Artificial Intelligence Research Society Conference (FLAIRS-2004), AAAI Press, 2004.

[Cla:83] William J. Clancey. *The Epistemology of a Rule-Based Expert System: A Framework for Explanation*. Artificial Intelligence, 20:215–251, 1983

[d3web] URL of the d3web project: http://www.d3web.de

[PG:03] Robert Plant and Rose Gamble: *Methodologies for the Development of Knowledge-Based Systems, 1982–2002*, The Knowledge Engineering Review, 18:47-81, 2003

[Pup:98] Frank Puppe. *Knowledge Reuse among Diagnostic Problem-Solving Methods in the Shell-Kit D3*. International Journal of Human-Computer Studies, 49:627–649, 1998

[Sch:01] 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

[Stu:98] Rudi Studer, V. Richard Benjamins, and Dieter Fensel. *Knowledge Engineering: Principles and Methods*. Data and Knowledge Engineering, 25(1-2):161–197, 1998

**Contact:**
Department of Computer Science VI, Am Hubland,
D-97074 Würzburg
http://www.ai-wuerzburg.de

**Joachim Baumeister** is research assistant at the University of Würzburg since 1999. His research focuses on practical knowledge engineering techniques including validation, restructuring, and semi-automatic learning of diagnostic knowledge.



**Frank Puppe** is a full professor at the University of Würzburg since 1992. His research interests comprise all aspects of knowledge-based systems and their practical use. He has authored and co-authored 5 books and more than 100 articles on the topic.



**Dietmar Seipel** is an associate professor at the University of Würzburg since 1995. His research areas are databases, knowledge bases, deductive databases, and logic programming. Currently, he is also investigating XML-based knowledge representations in different fields of applications.