VU VRIJE UNIVERSITEIT AMSTERDAM

VU Research Portal

Maintenance of KBS's by Domain Experts: The Holy Grail in Practice.

van Harmelen, F.A.H.; Kuipers, J.; Bultman, A.

published in
Thirtheenth International Conference on Industrial & Engineering Applications of Artificial Intelligence & Expert Systems IEA/AIE'00
2000

document version
Early version, also known as pre-print

Link to publication in VU Research Portal

citation for published version (APA)
van Harmelen, F. A. H., Kuipers, J., & Bultman, A. (2000). Maintenance of KBS's by Domain Experts: The Holy Grail in Practice. In R. Logananthara, G. Palm, & M. Ali (Eds.), *Thirtheenth International Conference on Industrial & Engineering Applications of Artificial Intelligence & Expert Systems IEA/AIE'00* Springer Verlag.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

E-mail address:
vuresearchportal.ub@vu.nl

Download date: 17. Jun. 2021

Maintenance of KBS's by Domain Experts

The Holy Grail in Practice

Arne Bultman[1], Joris Kuipers[1], and Frank van Harmelen[2]

[1] ASZ Research & Development
Kronenburg A-toren, Postbus 8300, 1005 CA Amsterdam, The Netherlands
{arne.bultman, joris.kuipers}@asz.nl
[2] Dept. of AI, Faculty of Sciences, Vrije Universiteit Amsterdam
Frank.van.Harmelen@cs.vu.nl

Abstract Enabling a domain expert to maintain his own knowledge in a Knowledge Based System has long been an ideal for the Knowledge Engineering community. In this paper we report on our experience with trying to achieve this ideal in a practical setting, by building a maintenance tool for an existing KBS. After a brief survey of various approaches to this problem described in literature, we select a domain- and task-specific modelling approach as the most promising and appropriate. First, we construct a domain ontology and a task model for the KBS system to be maintained, as well as a task analysis of the maintenance tool itself. The maintenance tool is subsequently implemented using a two layer architecture which seperates domain and system concepts. Although no full-scale evaluation has been undertaken, we report on our initial experience with this approach and present our conclusions.

1 Motivation

In Software Engineering it is well-known that the majority of the costs for software projects are not encountered during design or construction phases, but rather during the maintenance phase ([9]). The costs of the maintenance phase have been quoted to be as high as 80% of the total costs over the entire lifetime of a software project.

Although less data are available for Knowledge Based Systems, there is no reason to believe that this situation in Knowledge Engineering is any different. A response to this problem that has appeared over the years in the Knowledge Engineering literature is to "take the Knowledge Engineer out of the loop". The Knowledge Engineering process involves (at least) three parties: the intended end-user of the system, the domain expert who provides the expertise that forms the basis of the system, and the Knowledge Engineer, who acquires the knowledge of the domain expert and uses it to construct the system. The hope of Knowledge Engineering has long been to remove the Knowledge Engineer from the maintenance loop, and provide tools that enable the domain expert to maintain the system and adjust it to changing knowledge requirements.

In the traditional situation, with the Knowledge Engineer in the loop, every required update to the knowledge base must be communicated by the domain expert to the maintenance team of Knowledge Engineers, who implement the change. This change must then be validated by the domain expert, who may suggest further or other changes. Only at the end of this iterated process can the new version of the system be released to field-users. In the ideal situation, a domain expert uses appropriate tools to directly implement the required changes in the system without repeated, time-consuming and error-prone interaction with a Knowledge Engineer.

This paper describes a particular field-deployed Knowledge Based System for which we have tried to live up to this ideal: *given a realistic KBS, is it possible to provide tools that enable the domain expert to take on much of the system maintenance?*

As an indication of the possible gains to be had in our application domain, we mention that the KBS with which we are concerned is used in 28 different locations, while a team of only two Knowledge Engineers is available to deal with several dozens of requested updates per year.

In Sect. 2 we describe the historical developments of the tools investigated to realize the ideal of "maintenance by domain experts". This results in a conclusion on the most appropriate way to tackle the maintenance problem in our application domain. The application domain is briefly described in Sect. 3. Section 4 describes the conceptual modelling effort that was needed to enable maintenance by domain experts. The actual maintenance tool based on this conceptual model is described in Sect. 5. Section 6 briefly discusses the results, and Sect. 7 concludes.

2 Literature

[10] identifies four separate activities which are supported by Knowledge Engineering tools:

1. **Model construction:** constructing an abstract and generic model of the knowledge types required to perform a certain task. Examples of such tasks are classification, diagnosis, configuration, etc.
2. **Model instantiation:** such an abstract and generic model must be instantiated with knowledge from a specific domain. Any tool supporting this phase must of course be informed about the generic model and how the various knowledge types are represented in this model.
3. **Model compilation:** most often, the models constructed in the previous steps are not directly executable. In such a case, the instantiated model must be somehow "compiled" into an executable form.
4. **Updating the instantiated model:** When the resulting system does not (or: does no longer) function as required, either because incorrect knowledge has been entered, or because the domain knowledge has changed, the knowledge in the model must be updated.

It shall be clear to the reader that the tool support that we report on in this paper is concerned with exactly this fourth phase of Van Heijst's list.

In the Knowledge Engineering literature of the past twenty years or so, a number of different approaches can be identified for how to provide tool support for this fourth phase. We will briefly discuss these below.

Rule-Based Editors The first Knowledge Engineering tools were simply editors to update sets of production rules in a knowledge base. Such tools were often based on existing systems (EMYCIN [11], for example, was based on MYCIN [8]), and allowed the user to make updates to an already existing system.

An abstract specification of either task or domain was not available. For example, all that EMYCIN "knew" about MYCIN was that it used backward-chaining rules and hierarchical relations between concepts. As a consequence, little or no support was given for updating the knowledge in such a model, and the user had to be familiar with the internal workings of the system.

Task- and Method-Specific Architectures Because support from such rule-based editors was insufficient, subsequent research was aimed at tools based on a specific task-model. Two important advantages derived from the fact that these tools were informed about the specific task-model that was underlying the system. First, because such task-models closely correspond with the notions that are used by human domain-experts, these domain experts could communicate much more easily with these tools. Second, such tools could provide much more support during the instantiation and updating of these models. An example of such a system was SALT [7], which implemented the *propose-and-revise* method for the parametric design task.

These tools of course required research into such task-specific models. An early example of this was the work by Clancey on hierarchical classification [3]. Much of the Knowledge Engineering research in the 80's was dedicated to identifying such *generic tasks* [2].

Integrated Environments The task-specific architectures from the 80's were mostly aimed at instantiating models (phase 2), and provided little support for the other phases mentioned above. More integrated Knowledge Engineering environments have been constructed which provided integrated support for other phases. Two well-known examples of such an integrated environment are Protégé II [4] and EXPECT [5].

These environments provide support for phase 1 to 3 (constructing, instantiating and compiling models). Typically, they only provide support for updating models (phase 4) when these models were initially constructed within the same environment.

Our Approach In our case-study we are dealing with an *existing* KBS, which has not been designed or engineered with any of the existing KBS support environments, which means that no ready-made tool support for phase 4 is available.

Nevertheless, it is clear from the literature that the only approach to supporting model adaptation by domain experts is on the basis of a conceptual model of the system that is close enough to the concepts that are familiar to domain experts.

As a result, we have decided to proceed by first building a conceptual model of the existing application, and subsequently using this conceptual model as the basis for a maintenance tool.

Before we discuss the conceptual model of our application (Sect. 4) and the maintenance tool built on top of this model (Sect. 5), we briefly describe our application domain in the next section.

3 The ISB-system

In this section we will give an overview of the ISB-system in its current state. We will start with a short history of the system, explaining why it was built and what it does. Furthermore, we will describe the deployment of the system and its users.

ISB stands for "IndelingsSysteem Bedrijven" which means Company Classification System. Development of ISB began as part of a graduation project in 1994 and was later expanded into a full system and came into use two years ago. Its task is to classify employers into one of fifty-five sectors. Classification of an employer is necessary to determine the height of various insurance contributions for the Dutch social security system and is based on the primary activity of the employer. Because of a lack of consistency in the classifications various people made and a decreasing number of experts in this domain, the decision to build this system was made.

Over the years, the size of the system has grown from a small prototype to a fully fledged application containing over 1500 rules organized in approximately 250 different modules. Initially the system ran on the VAX platform and was build using AionDS 6.4[1]. Nowadays it is developed using AionDS 7 and runs on the Windows platform using a GUI.

ISB is used on a daily basis in 28 offices of the Gak company, a Dutch social security administrator. The classification process is, in theory, completely covered by legislation. In practice, however, the law leaves a lot of room for interpretation, especially when 'new' activities, such as the publishing of CD-Roms, are concerned. Moreover, ISB is not (yet) fully complete and correct; users of the system often report bugs and shortcomings of the system. As a result, a lot of maintenance is performed on the system.

Since the application has been developed and maintained by many different programmers, each having a different programming style, the structure of the application has degraded over time. Added functionality and the existence of deprecated functions contributed to this process. This means that the conceptual model on which the original prototype was based can hardly be recognized in the current application.

4 Conceptual Model of the ISB-system

In this section we will discuss a conceptual model we extracted from the ISB system in cooperation with the domain expert. A good conceptual model of the system to be maintained is important, especially when this system is as inconsistent in its implementation as the ISB-system. As we already explained in the previous section, the original conceptual model on which ISB was founded is outdated and incomplete. Therefore, we constructed a new conceptual model, consisting of a task decomposition of the system and a domain ontology. These will be discussed in this section.

4.1 Task Decomposition

A new task decomposition of the system was made, representing the tasks the system performs. Once the various tasks are recognized, it becomes possible to acknowledge the tasks on which maintenance is performed regularly. These tasks can then be examined to see if they are suitable for tool supported maintenance.

We will briefly discuss the tasks the ISB system performs.

First, the system performs some pre-checks. These determine for example if the employer doesn't have a main office in The Netherlands. These cases are treated differently and often do not need further classification.

After this, the user is asked to enter one or more entries which describe the employer's activities. These entries can be descriptions or verb-noun combinations. There are several thousands of possible entries. Because many entries are very similar, they can often be mapped onto a single entry which will be used in the reasoning process. Furthermore, several further entries can be added for use in the reasoning process based on the given entries. When this is done, dependencies between the existing entries are determined to see if certain activities are performed in service of other activities (e.g., delivery of fabricated goods).

Based on these dependencies, the primary activities of the employer are determined. This is done by asking the user about the nature of these activities. This dialog between the system and the user is guided by a decision tree inside the ISB system. Only one of these primary activities determines the employer's social function. Each primary activity is classified into its corresponding sector. If there are several primary activities which are classified differently, the social function is determined by the activity for which the highest wages are payed or, if wages are equal, by the expectation of the height of these wages.

After studying the system and talking to the domain expert, it soon became clear that the task on which maintenance was performed most often is *Classify Primary Activities*. Also, maintenance on other tasks often originates from this maintenance. Thus, the primary focus of our maintenance tool is on this task.

4.2 Domain Ontology

An ontology describing the domain in which the ISB-system operates served as a means of communication with the domain expert. This includes the communication with us during the design process as well as the communication with the

tool during the performance of maintenance. Using concepts from this ontology allows one to communicate with the expert using his own view on the domain, instead of using implementational concepts such as rules, inference mechanisms, etc.

An ontology also makes dependencies between domain concepts explicit. These dependencies clarify how changes involving certain concepts influence other concepts.

Figure 1 shows a part of the domain ontology we constructed. We used UML as our modelling language. For those not familiar with UML, a good introduction can be found in [6]. Note that relations are read from left to right, unless stated otherwise; a social function determines a classification, not the other way around. To give an impression of the size: the figure shown represents about one third of the total ontology.

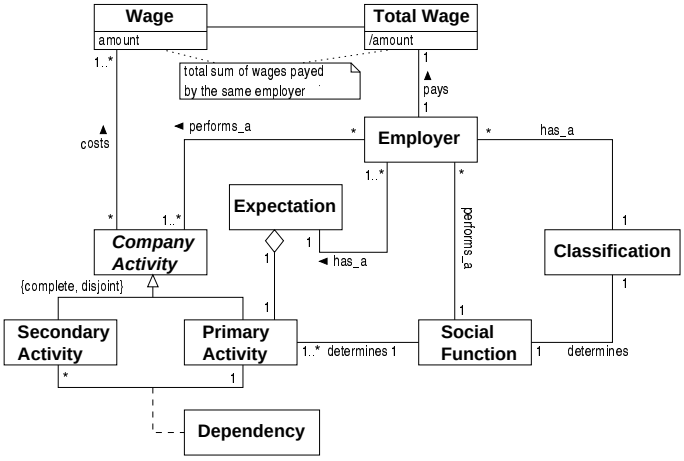

Figure1. Part of the domain ontology

5 The Maintenance Tool

In this section we discuss the maintenance tool. First, we present a task composition of the tool to give an overview of the tasks the tool performs. Following this, we describe the global design we used to implement the tool. Finally we will give a short impression of the final prototype.

5.1 Task Composition of the Tool

In order to be able to perform maintenance on the ISB system, the tool has to perform a number of tasks. These are depicted in Fig. 2.

The task *Knowledge Acquisition* involves the interactive acquisition of the expert's knowledge as well as acquiring the knowledge resident in the ISB system. The task *Knowledge Representation* consists of visualizing the acquired knowledge for the expert and translating this knowledge to its ISB representation. The acquired knowledge also needs to be interpreted in order to understand how the system classifies collected entries (*Entry Interpretation*) and, if this is done by means of a dialog with the user, how this dialog is constructed (*Dialog Interpretation*). Lastly, the acquired knowledge needs to be verified to ensure that it represents a valid classification (this is not to be confused with *validation* of the knowledge!).

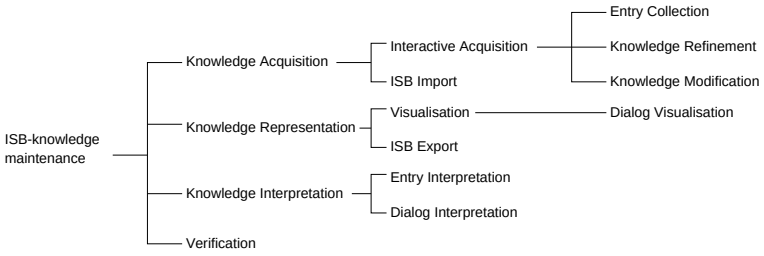

Figure2. Task composition of the maintenance tool

5.2 Global Design of the Tool

We wanted the design of the tool to be as generic as possible. More specifically, the design should be flexible enough to ensure that changes in the maintained system as well as changes in the user's concepts only require changes to their respective implementation in the tool. This requires a strict separation of these two concept spaces. This was realized by using the global design shown in Fig. 3.

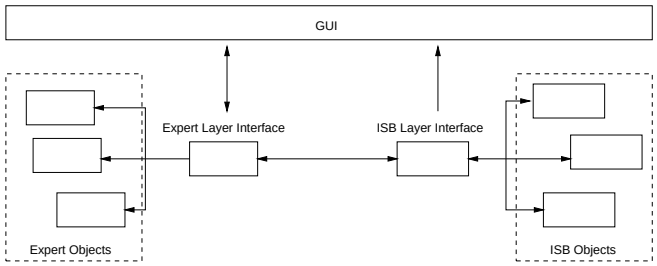

Figure3. Global Model of the maintenance tool

On the left, there's an *expert layer* which contains the objects which represent the expert's concepts, such as *decision trees* and *dialog based classifica-*

tion. On the right, there's an *ISB layer* which contains objects which represent ISB-specific concepts such as *rules* and *parameters*. To enable communication between the two layers, there are two interfaces: the *expert layer interface* (ELI) contains knowledge about the objects in the expert layer and the *ISB layer interface* (ILI) about the objects in the ISB layer. On top of this, there is a GUI which is used to communicate with the user. It shows graphical representations of *expert objects* and enables the user to manipulate them.

Let us give an example to illustrate the principle behind this design. Suppose the expert has created a new dialog. What happens when he presses the Save-button is the following. The dialog asks the ELI to save it. The ELI then requests the ILI to create the necessary ISB-concepts in order to save the dialog. The ILI will then create the needed objects, asking the ELI about the structure of the dialog. The created objects are then saved in the ISB system.

We found that using this design has several advantages. First, changes made to one layer only require changes in its corresponding interface. When, for instance, the ISB system would be reconstructed so that the syntax of the rules changes, the only thing which would have to be adapted is the method in the ILI which generates this rules. Also, this construction would make it easier to implement different views on the system for different types of users. Additionally, the separation of the two layers allows developers to work on the layers individually, as long as the interfaces are defined.

5.3 The Prototype

Using the global design and the task analysis, a prototype which implemented most of the required features was build. The prototype allows the expert to make new classifications for given entries, representing them in terms of the expert layer that are understandable to the expert. Figure 4 shows the most interesting screen of the prototype, the *dialog editor*. The dialog editor enables the expert to view and construct a decision tree which represents the dialog to be executed by the ISB system. It consists of three parts:

- **Decision tree window.** This part of the screen shows the actual tree. Buttons represent parameters and conclusions. The possible values of a parameter are determined by its associated question's possible answers. These values are shown in the white boxes on the lines between the buttons. Leaves of the tree represent a conclusion and are printed italic.
- **Edit window.** Whenever a button is pressed in the decision tree window, the corresponding properties of its parameter/conclusion are shown here. Each type of button has its own property tab. Using this tab, the user can select the question to be asked or create a new question, for example.
- **Status window.** As long as the tree is not finished, the missing or incorrect items are reported here. Suppose a conclusion does not yet have a classification, then a warning is given. The tree can only be saved when no more warnings are present. This ensures that the tree is logically complete and correct and that every conclusion is valid (NOT necessarily the right one).

Whenever a new classification is saved, it is translated to a collection of rules (ISB concepts).

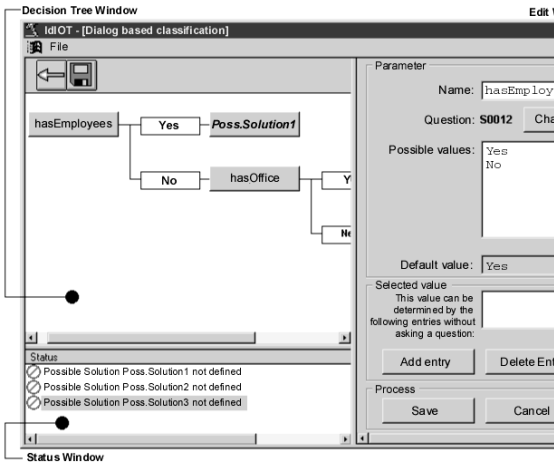

Figure4. Screenshot of the dialog editor (translated)

6 Evaluation

No full scale evaluation was performed using the tool described in the previous sections, but some initial results with domain experts were obtained, and these are reported in this section.

A domain expert (who is not an IT specialist) was able to build several new classification dialogs corresponding to existing update-requests using the tool after just 5 minutes of explanation.

This result is all the more encouraging since the domain expert that evaluated the maintenance tool was not the same domain expert that we used during the knowledge acquisition stages. The fact that this did not cause any substantial difficulties suggests to us a common frame of mind for both experts. It seems that the underlying ontology used (implicitly) by both experts is sufficiently similar that such an ontology is a good basis for a usable maintenance tool, thereby confirming our main hypothesis.

Notwithstanding this early success, a serious shortcoming was identified in the prototype of our tool. Because of irregularities in the implementation of the ISB system, it was not possible to provide a uniform translation procedure from the ISB layer to the expert layer. As a result, existing dialog-based classifications can only be updated by providing an entirely new definition for the dialog, the existing dialog cannot be reconstructed and modified.

7 Conclusions

The main points we have argued in this paper are that

1. it is possible to build maintenance tools that are usable by domain experts;
2. such tools should be based on conceptual models that are close to the domain experts (comprising both task-model and domain ontology of the system to be maintained);
3. such task-models and domain ontologies can be (re)constructed after the fact for existing systems, if this is required;
4. a principled two-layer architecture can be used to implement the connection between a maintenance tool (based on a conceptual model) and the performance system that is to be maintained.

In particular the construction of a domain ontology has been essential in the development of a maintenance tool that was usable by domain experts. Such an ontology captures the conceptual notions that the expert is familiar with. This ontology can then be connected with the structures in the actual implementation of the performance program in order to realize actual maintenance operations on this performance program. This does not amount to simply adding an ontology to an existing system. Instead, the far reaching effect of introducing the explicit ontology is that the domain expert performs maintenance operations on the conceptual model (which he understands), rather than on the implementation system.

References

[1] Computer Associates. PLATINUM Aion. *HTTP://www.ca.com/products/platinum/appdev/aion_ps.htm*.

[2] B. Chandrasekaran. Generic tasks in knowledge based reasoning: High level building blocks for expert system design. *IEEE Expert*, 1(3):23–30, 1986.

[3] W. J. Clancey. Heuristic classification. *AI*, 27:289–350, 1985.

[4] Henrik Eriksson, Angel R. Puerta, Mark A. Musen, John H. Gennari, Thomas E. Rothenfluh, and Samson W. Tu. Custom-tailored development tools for knowledge-based systems. *Knowledge Systems Laboratory, Medical Computer Science*, January 1994.

[5] Yolanda Gil and Marcelo Tallis. Transaction-based knowledge acquisition: Complex modifications made easier. In *Proceedings of the Ninth Knowledge Acquisition for Knowledge-Based Systems Workshop*. Banff, February 1995.

[6] Craig Larman. *Applying UML and Patterns*. Prentice Hall PTR, 1997.

[7] S. Marcus and J. Mcdermott. Salt: A knowledge acquisition language for propose-and-revise systems. *Artificial Intelligence*, 39(1):1–38, 1989.

[8] E. H. Shortliffe. *Computer-Based Medical Consultations: Mycin*. American-Elsevier, New York, 1979.

[9] I. Sommerville. *Software Engineering*. Addison Wesley, Bonn, Germany, 1987.

[10] G. van Heijst, A.Th. Schreiber, and B.J. Wielinga. Using explicit ontologies in kbs development. *International journal of human-computer studies*, 45:183–292, 1997.

[11] W. van Melle. A domain independent production rule system for consultation programs. *IJCAI*, 1979.