Problem-Solving Methods: Making Assumptions for Efficiency Reasons

Dieter Fensel & Remco Straatman[1]
Department of Social Science Informatics (SWI), University of Amsterdam, The Netherlands
{fensel | remco}@swi.psy.uva.nl, http://www.swi.psy.uva.nl/usr/dieter/home.html

Abstract. In this paper we present the following view on problem-solving methods for knowledge-based systems: Problem-solving methods describe an *efficient reasoning strategy* to achieve a goal by introducing *assumptions* about the available domain knowledge and the required functionality. Assumptions, dynamic reasoning behavior, and functionality are the three elements necessary to characterize a problem-solving method.

1 Introduction

The concept *problem-solving method* (PSM) is present in a large part of current knowledge-engineering frameworks (e.g. GENERIC TASKS [Chandrasekaran et al., 1992]; ROLE-LIMITING METHODS [Marcus, 1988], [Puppe, 1993]; KADS [Schreiber et al., 1993] and CommonKADS [Schreiber et al., 1994]; the METHOD-TO-TASK approach [Musen, 1992]; COMPONENTS OF EXPERTISE [Steels, 1990]; GDM [Terpstra et al., 1993]). Libraries of PSM are described in [Benjamins, 1993], [Breuker & Van de Velde, 1994], [Chandrasekaran et al., 1992], and [Puppe, 1993]. In general a PSM describes which reasoning steps and which types of knowledge are needed to perform a task. Such a description should be domain and implementation independent. Problem solving methods are used in a number of ways in knowledge engineering: as a guideline to acquire problem-solving knowledge from an expert, as a description of the main rationale of the reasoning process of the expert and the knowledge-based system, as a skeletal description of the design model of the knowledge-based system, and to enable flexible reasoning by selecting methods during problem solving.

However, a question that has not been answered clearly is the relation between PSMs and *efficiency* of the problem-solving process. Most descriptions of PSM frameworks do point to PSMs as being somehow related to efficiency, however no framework makes this relation explicit. Others claim to have no concern for efficiency since their PSMs are only used to capture the expert's problem-solving behavior. But one must be aware that experts also have to solve the task given their real-life limitations. In fact a large part of expert-knowledge is concerned exactly with efficient reasoning given these limitations. The conceptualization of a domain from an expert differs from the conceptualization of a novice as the former reflects the learning process which yields to efficiency in problem solving.

According to us the main point of a PSM is: *providing the desired functionality in an efficient fashion*. In general, most problems tackled with knowledge-based systems are inherently complex and intractable, i.e., their time complexity is NP-hard (see e.g. [Bylander, 1991], [Bylander et al., 1991], and [Nebel, 1995]).[2] A PSM has to describe not just a realization of the functionality, but one which takes into account the

1. Supported by the Netherlands Computer Science Research Foundation with financial support from the Netherlands Organization for Scientific Research (NWO).

In N. Shadbolt et al. (eds.), Advances in Knowledge Acquisition, Lecture Notes in Artificial Intelligence (LNAI), no 1076, Springer-Verlag, 1996.

constraints of the reasoning process and the complexity of the task. The constraints have to do with the fact that we do not want to achieve the functionality *in theory* but rather *in practice*. When this relation between PSMs and efficiency is ignored or kept as an implicit notion, both the selection and design of PSMs cannot be performed in an informed manner. Besides the efficiency in terms of computational effort of a PSM, there are further aspects which can influence design decisions of appropriate PSM: The efficiency of the entire knowledge-based system, the optimality of the combined problem solver user and system (e.g. minimizing the number of tests a patient has to suffer from in medical diagnosis), and efficiency of the development process of the knowledge-based system.[3]

After stating the claim that PSMs provide functionality in an efficient way, the next question then is: how could this be achieved if the problems are intractable in their general form? In our view, the way problem solving methods achieve efficient realization of functionality is by making *assumptions*. The assumptions put restrictions on the context of the PSM, such as the domain knowledge and the possible inputs of the method or the precise definition of the functionality (i.e., the goal which can be achieved by applying the PSM). These restrictions enable reasoning to be performed in an efficient manner.

The role that assumptions play in the efficient realization of functionality suggests that the process of designing PSMs must be based on these assumptions. [Akkermans et al., 1993] and [Wielinga et al., 1995] introduce a general approach that views the construction process of PSMs for knowledge-based systems as an assumption-driven activity. A formal specification of a task is derived from informal requirements by introducing assumptions about the problem and the problem space. This task specification is refined into a functional specification of the PSM by making assumptions about the problem-solving paradigm and the available domain theory. Further assumptions are introduced in the process of defining an operational specification of the method. A task is decomposed into declaratively described subtasks and the data and control flow between these subtasks are defined. We will use this approach as a general framework and try to make it more concrete. Our focus lies thereby on assumptions which are related to efficiency of a PSM. We propose to view the process of constructing a PSM for a given function as the process of incrementally adding assumptions that enable efficient reasoning. Summarizing, we want to make the following claims in this paper:

- PSMs are concerned with *efficient* realization of functionality. This is an important characteristic of PSMs and should be dealt with explicitly.
- PSMs achieve this efficiency by making *assumptions* about resources provided by their context (such as domain knowledge) and by assumptions about the precise definition of the task. It is important to make explicit these assumptions to reason about PSMs.
- The process of *constructing* PSMs is assumption-based. During this process assumptions are added that facilitate efficient operationalization of the desired functionality.

One type of assumptions of a PSM defines the relation between the method and the domain knowledge which is required by it. These assumptions describe the domain dependency of a PSM in domain-independent terms. The assumptions can be viewed

2. Exceptions are classification problems which have often known polynomial time complexity (see [Goel et al., 1987]).

3. See [Landes & Studer, 1995] for a discussion of further non-functional requirements which could influence design decisions.

as an index of a method since a method can only be chosen if its assumptions are fulfilled by domain knowledge. We can then view the assumptions as proof obligations for the domain knowledge. The assumptions can also define goals for the knowledge acquisition process. Making explicit the assumptions of a PSM about the domain knowledge is a way to deal with the *interaction problem*. The interaction problem [Bylander & Chandrasekaran, 1988] states that domain knowledge cannot be represented independently of how it will be used in reasoning. Vice versa, a PSM and its specific variants cannot be constructed independently of assumptions about the available domain knowledge. Developing reusable PSMs as well as reusable domain theories requires the explicit representation of the assumptions of the method about the domain knowledge; ***and*** the explicit representations of properties of the domain knowledge that can be related to these assumptions.

Ontologies (i.e., meta-theories of domain theories) are proposed as a means to explicitly represent the commitments of a domain theory (cf. [Top & Akkermans, 1994], [Wielinga & Schreiber, 1994]). Ontologies introduce generic terminologies which are instantiated by a domain theory. These generic terminologies can be viewed as representations of the ontological commitments of a domain theory and could define a link to the assumptions of a PSM.

The paper is structured as follows: In section 2 we discuss why and how PSMs introduce efficient reasoning and section 3 sketches the different parts of PSMs as well as their relationships.

2 Why Are Problem-Solving Methods Necessary

We use the task parametric design and the PSM `propose & revise` to illustrate the main points of our paper. We use this example since our experiences in the Sisyphus-II project inspired our current point of view. Sisyphus-II [Schreiber & Birmingham, 1996] aimed at comparing different approaches to knowledge engineering. The task is to configure a vertical transportation system (an elevator) which was originally described in [Marcus et al., 1988] who developed the PSM `propose & revise` to solve this configuration task.

In the following, we formally define the task `parametric design`. Then, we define a PSM `generate & test` that can theoretically be used to solve this task. This method can be derived straightforwardly from the task specification. Because this method is very inefficient, we then discuss a more efficient PSM `propose & revise` as introduced by [Marcus, 1988] and [Marcus et al., 1988]. `Propose & revise` weakens the task and makes additional assumptions about available domain knowledge in order to gain efficiency. The purpose of this section is neither to define the appropriate way to specify the task parametric design nor the PSM `propose & revise`; their only use is to illustrate our ideas on PSM.

2.1 A Definition of Parametric Design

A parametric design problem can be defined by a problem space, requirements, constraints, and a preference (see [Tank, 1992] for more details). The *problem space* describes the space which contains all possible designs. The definition of the problem space is domain-specific knowledge. Further, a finite set of *requirements* is assumed to be given by the user. A design that fulfils all requirements is called a *desired design*. In addition to the case-specific user input, a finite set of *constraints* model additional conditions for a valid design. These constraints are domain knowledge describing the regularities in the domain in which the design is constructed. A design that fulfils all constraints is called a *valid design*. A design that is desired and valid is called a *solution*. The *preference* defines a preference function on the problem space and can be used to discriminate between different solutions.

In the case of *parametric* design, a design artifact is described by a set of attribute-value pairs. Let $A_1,..., A_n$ be a fixed set of parameters (i.e. attributes) with fixed ranges $R_1,...,R_n$.

Def 1. Problem Space

The *problem space* is the cartesian product $R_1 \times ... \times R_n$

Def 2. Requirements and Constraints

The set of *requirements* and *constraints* are represented by two relations *R* and *C* on the problem space defining subsets of the problem space.

Def 3. Possible designs, desired design, valid design, solution

A *possible design* is an element of the problem space, a *desired design* is an element of *R*, a *valid design* is an element of *C*, and a *possible solution* is an element of *R* and *C*.

By applying the *preference P*, an *optimal solution* is selected out of all solutions.

Def 4. Preference

The *preference P* is a partial function on all possible designs.

Def 5. Optimal solution

An *optimal solution* is a solution for which no other solution exists which has a higher preference value.

In general, several optimal solutions could exist. Therefore, one can further distinguish whether the user gets all of them or a non-deterministic selection of some. This definitions can be extended by introducing priorities on requirements and constraints, or by distinguishing between constraints which always hold and constraints which should hold etc, but this is beyond the scope of this paper.

2.2 A Non-Efficient Solution by Generate & Test

A straightforward operationalization of the declarative task specification can be achieved by applying a variant of `generate & test`. The method defines four different inferences and four different types of knowledge that are required by it. The inference structure of this method is given in Fig. 1.

- The inference action `generate` requires knowledge that describes what constitutes a possible design.
- The inference action `R-test` requires knowledge that describes what constitutes a desired designs.
- The inference action `C-test` requires knowledge that describes what constitutes a valid design.
- The inference action `select` requires knowledge that evaluates solutions, i.e., knowledge that describes what constitutes a preferred design.

We have to complete the operational method description by defining its control. Again we do this in a straightforward manner (see Fig. 2). The control flow specifies the following reasoning process: First, all possible designs are derived. Second, all valid designs are derived. Third, all desired designs are derived. Fourth, valid and desired designs are intersected. Fifth, an optimal solution is selected. The sequence of the second and third steps is arbitrary and could also be specified as parallel activities.

The advantage of this method is that it clearly separates the different types of knowledge that are included in the functional specification of the parametric design task. On the other hand, it is precisely this separation that prevents the development of an efficient problem-solver. The knowledge about what is a correct (i.e., valid and desired) and good solution is clearly separated from the generation step, and there is no

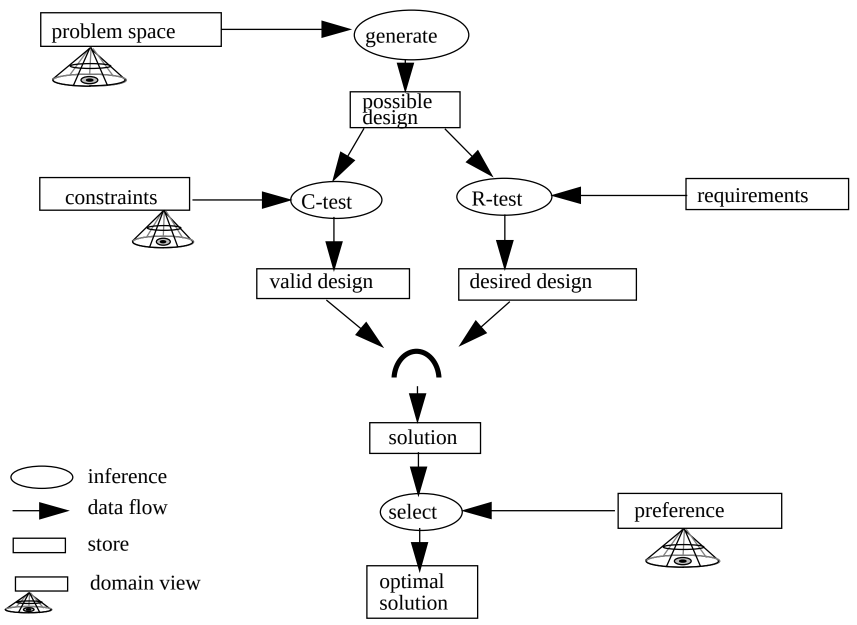

Fig. 1 Inference structure of *generate &test*.

feedback from the results of the test and evaluation step.This method is clearly not very efficient as it has to derive and test all possible designs (i.e., the complete problem space). Still, the method is able to realize the functionality specified by the task if the problem space is finite. With respect to infinite problem spaces, three remarks could be made.

(1) At the knowledge level in its original sense, one should abstract from all computational concerns like limited space or computation time by describing a completely rational agent. Therefore, it is not at all clear whether an infinite search space should be regarded as a problem. In fact the problem of tractability of the method arises not only for infinite search spaces, but also for finite spaces, because the size of the space increases exponentially with the number of parameters and their ranges. Dealing with the size of the search space therefore immediately leads to *limited* rationality[4]. Even for realistic settings with finite search spaces, no computational agent can be implemented that realizes the method in an acceptable way.

(2) One can always transform each infinite search space into a finite one by making some pragmatic assumptions. These domain and task-specific assumptions improve the efficiency of our method by reducing the search space.

```
possible design := generate_all;
valid design := C-test(possible design);
desired design := R-test(possible design);
solution := valid design ∩ desired design;
optimal solution := select(solution)
```

Fig. 2 Control flow I of *generate &test.*

4. Decision procedures with perfect rationality try to find an optimal solution, whereas decision procedures with limited rationality reflect also on the costs to find such an optimal solution.

```
repeat
    possible design := generate_one;
    valid design := C-test(possible design);
    desired design := R-test(possible design);
    solution := valid design ∩ desired design;
    acceptable solution := select(solution)
until ∅ ≠ acceptable solution
```

Fig. 3 Control flow II of *generate &test*.

(3) One could think of *reducing the functionality of the method*. In the task description (see Def. 5), we required that an optimal solution should be found. A weaker definition of the functionality of the method is to require that an *acceptable solution* is a solution which has a preference higher than some threshold *t*.

Def 6. Acceptable solution

An *acceptable solution* is a solution *s* with $P(s) > t$.

We also see here the problem of using worst-case analysis: In the worst case it takes the same effort to find an optimal solution (i.e., a global optimum), or an acceptable solution as defined now. The technique of weakening the task definition to improve the efficiency of the computation is commonly used. A well-known example from the field of model-based diagnosis is the *single-fault assumption* [de Kleer & Williams, 1987]. It assumes that the symptoms of a device are caused by one fault. This can be used to improve the efficiency of the methods, but prevents these methods from dealing with situations where the device suffers from several faults.

The weakened functionality of Def. 6 enables us to define a new control flow for the method that allows the method to deal with infinite problem spaces (see Fig. 3).

The sequence of the four inference actions is repeated until an acceptable solution is found. The inference action `generate` should now derive one possible design per step, which is further on treated by the two test steps and the `select` step. For each given probability $0 < \alpha < 1$ one can guarantee that the method finds a solution (if one exists) in finite time with probability greater than $1 - \alpha$ if each element of the problem space has the same chance to get proposed by `generate`.

Making the search finite by introducing domain-specific assumptions or reducing the functionality by weakening the solution criteria transforms `generate & test` into a method that can solve the problem in theory. Still we cannot expect to get an agent which solves this task in a realistic amount of time by implementing the method. We have not really described a PSM but rather a kind of uninformed theorem prover. Arbitrary generated designs are tested whether they are desired, valid, and preferred or not. Still, we have an operational description of how to achieve the goal. From the point of view that one does not want to care about efficiency, this could be a legal point to describe the essence of the reasoning process of a system that solves the task. For example, [Rouveirol & Albert, 1994] define a knowledge level model of machine-learning algorithms by applying the `generate & test` scheme and [Bredeweg, 1994] uses it to define a top-level view on the diagnostic task.

2.3 An Efficient Solution with Propose & Revise

The main advantages of `generate & test` as it is developed above are:

- It requires only the knowledge given by the functional specification, and the four types of knowledge (considering the requirements as knowledge) are clearly separated: each inference uses precisely one knowledge type. The description of the problem space is used in the generation step, the requirements and the constraints are used in two test steps, and the preference is used in the select step.

- Its inference structure is cycle-free. That is, its operational specification does not contain feedback loops that introduce non-monotonicity into the reasoning process.

`Generate & test` leads to a precise and clear distinction of different conceptual types of knowledge and defines the dynamic behavior of the problem-solving process in a highly understandable manner. On the other hand, these advantages are precisely the reasons that cause the inefficient problem-solving behavior. The PSM `propose & revise` as discussed in [Marcus et al., 1988] adds efficiency to the problem-solving process by regarding the given properties of `generate & test` as disadvantages, and introducing static and dynamic feedback into the problem-solving process. An expert has learned which design decisions led to desired, valid, and preferred solutions and which did not. Therefore, expertise compiles test knowledge into the generation step. New types of knowledge arise that enable the efficient generation of solutions.

- `Generate & test` requires only the knowledge given by the functional specification: An expert includes feedback based on experience from solving earlier design problems. In `generate & test` the knowledge about what is a desired, correct, and preferred solution is clearly separated from the generation step: A much more clever strategy is to use these knowledge types to guide the generation step of possible designs.
- There is `no` dynamic feedback in `generate & test` from the results of the test and evaluation step of a given design: If a design derived during problem solving is not a solution, a new design is derived in the next step. Dynamic feedback would include the reported shortcomings of the first proposed design as guidance for its modification by the next derived design.

The use of the test and evaluation knowledge as guidance for the generation step and the use of the feedback of the test step as input for the generation step are precisely the main improvements which are incorporated into the `propose & revise` method. In a pessimistic manner this can be expressed as destroying the clear conceptual distinctions of `generate & test`. Optimistically, this can be viewed as introducing new types of knowledge which glue these different aspects together, thereby adding expertise.

The `generate` step becomes decomposed into two different activities. The `propose` step derives an initial design based on the requirements and the `revise` step tries to improve an incorrect design based on the feedback of the `C-test` step. To this end, it uses the meta-information that this design is incorrect as well as constraint violations reported by `C-test`. We get the following conceptual structure of the method (see Fig. 4): The `propose` step requires knowledge which enables it to derive desired designs using the requirements as input. The `revise` step requires knowledge which enables it to fix constraint violations of desired designs. Additionally it uses the reported violations to guide the repair process. Revise delivers an acceptable solution as its output. The third inference action `C-test` requires constraints to check desired designs. As output it derives the set of constraints violated by a design.

`Propose & revise` requires a number of assumptions to justify its I/O behavior with the specified task. `Propose & revise` as described in [Marcus et al., 1988] does not require an R-test. That is, designs are not checked on the requirements. Propose is assumed to derive desired designs, instead of possible designs as delivered by the generate step. It is also assumed that the revise step delivers designs that are desired (i.e., this would be an assumptions about the domain specific repair knowledge) or that requirements violations which are not fixed by it must be accepted (i.e., this would weaken the functionality of the method).[5] Finally, `propose & revise` does not contain a selection of a solution using the preferences. That is, it is either assumed that

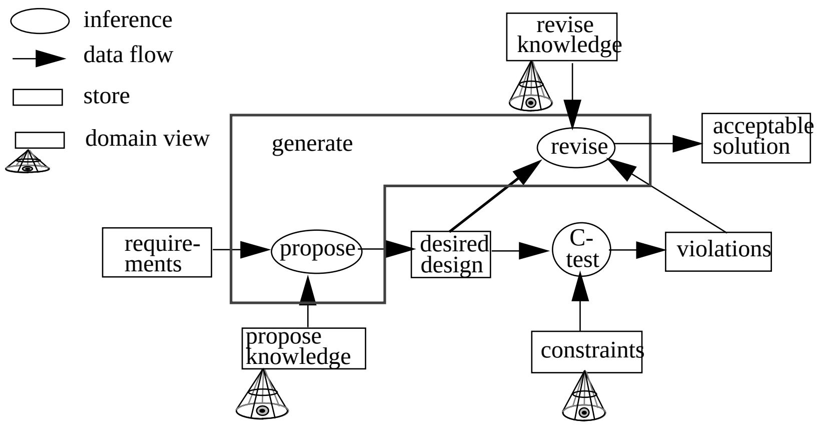

Fig. 4 Inference structure of *propose & revise*.

the propose step as well as the revise step deliver acceptable (or optimal) solutions or that the functionality of the task is reduced to finding just a solution.

When we take a closer look at `revise` by distinguishing several substeps, we see that the `C-test` inference appears also as sub-step of `revise` (cf. [Fensel, 1995a]). After applying some repair rules on an invalid design, `revise` has to check whether the given violations are overcome and whether no new violations are introduced by applying the repair rules. Again, test knowledge that was originally separated from the generation step now appears as sub-activity of it. The `revise` step causes the main computational effort of the method (and also the main effort in precisely specifying the behaviour of the method). The actual efficiency of the method therefore heavily relies on the quality of the repair rules that are required by `revise`, but also on the propose knowledge. The propose knowledge is responsible for ensuring preferred desired designs that require less repair activities. The main point of the method in gaining efficiency is not so much to get rid of the R-test and selection step, but to reduce the search space from the set of all possible designs (i.e., the complete problem space) to the set of preferred desired designs which should be nearly valid.

Two possible control flows for `propose & revise` are shown in Fig. 5. Control flow I tries to find an acceptable solution in one attempt (i.e., the assumption is that this can be done), whereas control flow II includes a loop of *propose*, *test*, and *revise* until an acceptable solution is found.

```
Control flow I of propose & revise
desired design := propose(requirements);
violations := test(desired design);
acceptable solution :=
        revise(desired design,violations)
```

```
Control flow II of propose & revise
repeat
    desired design := propose(requirements);
    violations := test(desired design);
    acceptable solution :=
            revise(desired design,violations)
until $\emptyset \neq$ acceptable solution
```

Fig. 5 Different control flow of *propose & revise*.

`Stepwise propose & revise`. Our current characterization of propose and revise

5. One can also argue that `propose & revise` extends the competence of the method because it can modify the problem space by overwriting requirements. `Generate & test` can only search through the defined problem space.

works with complete designs. But, as discussed in [Marcus et al., 1988], it also makes sense to regard repair activities as soon as possible. That is, instead of proposing a complete design that is then repaired, we can also incrementally develop a design, and repair at each step where a constraint violations occurs. We have not yet exploited the fact that we specify `propose & revise` for a subclass of design tasks, namely, for parametric designs. A natural decomposition of the entire design is provided by the parameters describing it. In each `propose` step we can assign one or some parameters a value; and we can apply `revise` to these incomplete designs before we propose the next parameter values. This *divide & conquer* strategy with intermediate repairs requires that the constraints do not interact much (see [Marcus et al., 1988]). Otherwise, one always has to redo earlier repair activities when new constraint violations are reported for another parameter. The stepwise derivation of incomplete designs requires the introduction of the new inferences `select-parameter` and `check-completeness` which causes a structure-altering transformations of the original version of `propose & revise`.

2.4 A List of Assumptions

[Poeck et al., 1996] present a specification of `propose & revise` applied to the VT problem (configuring a vertical transportation system). [Fensel, 1995a] has analysed this variant of `propose & revise` and reported several assumptions about domain knowledge. This variant of `propose & revise` consists of four steps where each requires different types of knowledge.

`Select`. A parameter is selected which should get a value in the next `propose` step. In the given application domain, a set of propose rules is given. Each rule can be used to derive the value of the parameter that forms its conclusion from the values of the parameters of its premises. Each rule could be further accomplished with guards defining applicability criteria for the rule depending on already derived parameter values. The `select` step uses these implicitly given dependencies between the parameter as domain-specific meta-knowledge, and assumes that this network defines a partial strong ordering on the set of parameters. At each step each parameter that depends only on already computed parameters according to the applicable propose rules is regarded as a possible choice. One parameter is non-deterministically chosen from this set of possible candidates. That is, `select` does not make further assumptions about knowledge that would guide this second selection step. The implicit assumption is that this selection does not change performance and quality of the problem-solving process and its result.

`Propose`. The `propose` step assumes that *either precisely one applicable* propose rule *or one* user input is given to derive the value of the selected parameter. A parameter should not depend on itself (i.e., no recursive derivation rules requiring a fixpoint operation are allowed). This requirement is not as trivial as it seems to be, as it depends on the rules that become applicable during the problem-solving process.[6]

`Test`. The `test` step requires constraints that define a solvable problem and that exclude all non-valid possibilities.

`Revise`. The `revise` step is decomposed into a set of more elementary inferences. A `select` step non-deterministically selects one constraint violation from the set of all violations that were detected by `test`. Again, the implicit assumption is that this selection does not influence performance and quality of the problem-solving process

6. There may exist several propose rules for a parameter, but depending on the already derived values only one should be applicable (see for more details [Fensel, 1995a]).

and its result. This is a very critical assumption because the method does not backtrack from this selection. `Derive` computes the set of all possible fix combinations (i.e., the set of all sets of elementary fixes) that could possibly resolve the selected constraint violation. Each fix combination (i.e., each set of elementary fixes) as well as the set of all fix combinations must be finite. This requirement is not trivial because some fixes (e.g., increment the value by one) can be applied several times, and specific constraints are required to restrict the number of legal repetitions of these fixes to guaranty finiteness. From the set of all possible fix combinations one is selected by another `select` step. A cost function is used to guide this selection step. The application of a fix decreases the quality of a design product because it overwrites user requirements or it increases the cost of the product. The cost function defined on the fixes (more precisely on the fix combinations) must be defined in a way that reflects the preferences between possible designs. `Apply` applies a fix combination. It is again realized by a set of elementary inferences, because it requires the propagation of modified values according to the dependency network of parameter. The precise definition of this step and further aspects of the revise step are beyond the scope of our paper.

2.5 Resume

The `propose` step as well as the `revise` step glues together types of knowledge that were treated separately by `generate & test`. These new knowledge types define strong assumptions about the domain knowledge required by the method. The only reason for doing this is trying to gain *efficiency*. That is, we assume that the “refined” PSM `propose & revise` will be able to find a solution faster than `generate & test` (or a better solution in the same amount of time). Therefore, developing PSMs means to blur conceptual distinctions and to introduce assumptions about new types of domain knowledge for reasons of efficiency. The pure and very clear separation of four types of knowledge in `generate & test` is destroyed by forcing parts of the test and evaluation knowledge into the generation step in order to improve the efficiency of the problem-solving process. We can conclude that `propose & revise` provides the same or less functionality as `generate & test`. It makes stronger assumptions to achieve this functionality. Finally, `propose & revise` is much harder to understand in detail than `generate & test`. Especially the `revise` step requires several levels of refinement to define it precisely (see [Fensel, 1995a]) and “the non-monotonic nature of the *Propose and Revise* method is difficult to capture in intuitively understandable theories.” [Wielinga et al., 1995]. Given this we must face the fact that the only reason why we still would prefer `propose & revise` is for reasons of efficiency.

2.6 Principles of Efficient Reasoning

A large part of the problem types tackled by PSMs are hard problems. This means that there is no hope of finding a method that will solve all cases in polynomial time. [Rich & Knight, 1991] even define AI as “... the study of techniques for solving exponentially hard problems in polynomial time by exploiting knowledge about the problem domain.” Most PSMs in knowledge engineering implement a heuristic strategy to tackle problems for which no polynomial algorithms are known. There are basically three general approaches:

- Applying techniques to define, structure and minimize the search space of a problem. An appropriate definition of the problem space can immediately rule out most of the effort in finding a solution. In `generate & test` this implies the transfer from test knowledge into the generate step to shrink the problem space `generate` is working on, and to define the sequence in which possible solutions are generated. Such techniques cannot change the complexity class of a problem but can drastically change the actual behavior of a system. The ordering on different search alternatives can have a *heuristic* or a *non-heuristic* nature.

- Introducing assumptions about the domain knowledge (or the user of the system) which reduces the functionality or the complexity of the part of the problem that is solved by the PSM. In terms of complexity analysis, the domain knowledge or the user of the system is used as an oracle that solves complex parts of the problem.
- Weakening the desired functionality of the system and reducing therefore the complexity of the problem by introducing assumptions about the precise problem type. An example of this type of change is to no longer require an optimal solution, but only an acceptable one, or the single-fault assumption in model-based diagnosis (see [van Harmelen & ten Teije, 1995] for further examples in diagnostic reasoning).

The second and the third approach tackle the problem by solving a different, more restricted, problem. In the second approach this restriction is already part of the domain which implies that the provided functionality of the entire system does not change. The third approach explicitly changes the functionality. Studying these assumptions and restrictions and their influence on the efficiency defines a link to the work in complexity analysis. [Nebel, 1995] proposes different strategies to deal with complex problems. He proposes different ways to restrict the functionality of the system, but he also mentions that additional domain knowledge can change the complexity class of a problem.

3 Problem-Solving Methods: Parts and Relations

In the following, we briefly sketch the different parts of a description of a PSM. Then we discuss the relationships between these parts and between a PSM and its environment.

3.1 The Different Parts of A Problem-Solving Method

The description of a PSM consists of four main parts: a *functional specification*, a *cost description*, an *operational specification*, and its *assumptions* over available resources for the reasoning process (cf. Fig. 6). The functional specification, PSM_f, is a declarative description of the input-output behavior the PSM was designed for. The functional description can be seen as a description of what can be achieved by the PSM. The functional specification is enriched by a cost description, PSM_c, which describes the costs that are associated with using this PSM. The operational specification, PSM_o, describes how to realize the functionality in a reasoning system. The assumptions, PSM_A describe conditions under which the structure described in the operational description will achieve the functional specification (with the described costs). Given the functional description, the assumptions and the cost description, we are able to express what the utility of the PSM is. The method provides the functionality described in PSM_f with costs PSM_c and it expects domain knowledge in return that fulfils the assumptions described in PSM_A.

The functional description describes the input-output behavior of the PSM. The simplest form of *functional description* is a list of input-output tuples. However, in practice, this will often not be possible or feasible because of the size of such a specification. In practice, some form of mathematical representation of the relation between input and output is needed (see e.g. [Levesque, 1984] and [ten Teije & van Harmelen, 1994] for knowledge-based systems and for [Fensel, 1995b] a survey on

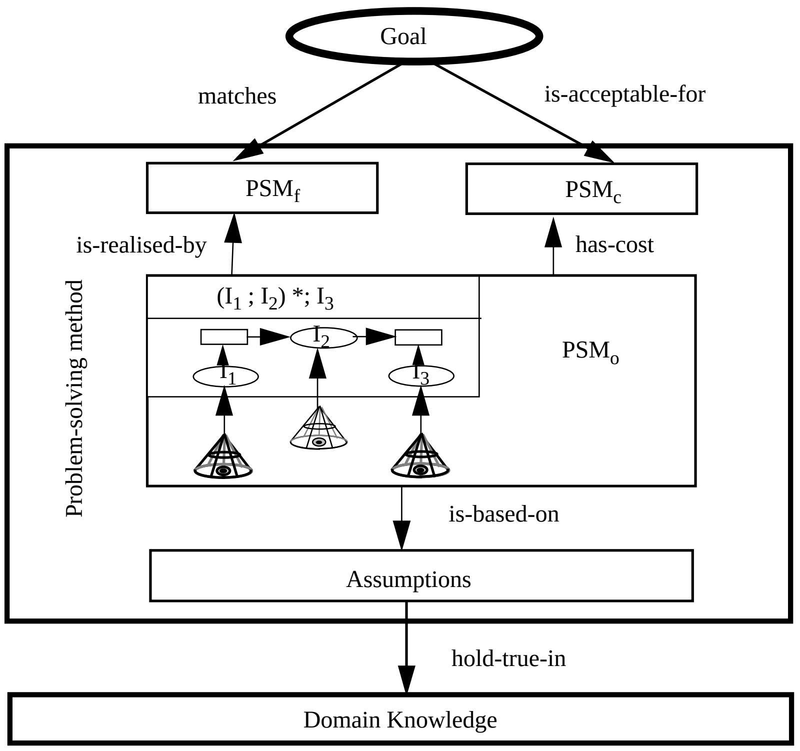

Fig. 6 The PSM and its environment.

functional specification techniques in software engineering).

The cost of a method could include the computing time, the number of interactions with the user, the costs of external tests etc., required by the method. We want to mention however, that in concern to computational complexity we are less interested in the *worst case behavior,* because a significant part of the applied methods are of a heuristic nature which do not improve the worst-case behavior. The worst case is precisely the case where the heuristics do not bring any improvement. Also, *average-case behavior* analysis often introduces assumptions about the problem distribution which are hard to justify. Therefore, we want to look at the complexity of typical cases and assume the expert as oracle that can provide typical cases.

The operational specification consists of inferences and the data- and control-flow between them. The *inferences* specify the reasoning steps that are used to accomplish the functionality of the method. They are described by their input/output relation and can be achieved by a method or a primitive inference. Inferences can be realized by either *methods* (i.e., the description of a PSM can be hierarchically composed) or *primitive inferences*. These primitive inferences are atomic reasoning steps which are not decomposed any further and are described by their input/output relation. In fact one can think of primitive inferences as a special type of problem solving method which has no operational specification. The *roles* are either *stores* that are used to act as input and output of the inferences or *domain views* in which case they get their values from the domain knowledge. A syntactical variant of first-order logic including semantical data modelling primitives is an appropriate mathematical notation for these static aspects of the operational specification (cf. [Kifer et al., 1995]). Finally, the *control* of a PSM describes the ordering of execution of the inferences. Dynamic logic [Kozen, 1990] is a natural candidate for this part of the specification. A survey of languages which were developed to specify KADS models of expertise and which could also be used as a starting point for specifying reusable PSMs can be found in [Fensel & van Harmelen, 1994].

Notice, that we do not make any claim whether the decomposition and control of the reasoning process as defined by the operational specification corresponds to the design model or the structure and control of the implementation of the PSM. The operational specification defines a reasoning process that achieves the desired functionality if the assumptions are fulfilled. That is, the operational specification is the rationalize of these assumptions or the structure of the proof that these assumption enable the functionality (under the specified costs of the reasoning process).

The assumptions of a method are both necessary and sufficient criteria for the application of the method. The assumptions can define restrictions on the possible input of the method, and on the availability and the properties of domain knowledge. Examples of input assumptions are for example the fact that the requirements for a design should not conflict, or that an input list must be sorted according to some criterion. Examples of assumptions about domain knowledge are the availability of heuristics that link violated constraints to possible repair actions, or the fact that a preference relation must describe a complete ordering. As the assumptions describe properties of the domain knowledge, meta-logic seems a good candidate language for formally specifying the assumptions.

3.2 The Different Relationships between the Parts of a PSM and its Environment

The relation between the functional specification, the operational specification, and the assumptions is essential for understanding PSMs. Given that the assumptions hold, the reasoning system defined by the operational specification will exhibit the input-output behavior specified in the functional specification. One can view *the assumptions as the missing pieces in the proof that the behavior of the method satisfies its goal.* Four types of proof obligations arise:

(1) the external relation between the goal (i.e., the task) and the functionality of the PSM has to be established. One has to ensure that the functionality of the method is strong enough to fulfil the goal if its assumptions are fulfilled.

(2) the external relation between the method and the domain knowledge has to be established. One has to ensure that the domain knowledge fulfils the assumptions of the method. Depending on the type of an assumption, we have to ensure either that the domain knowledge implies an assumption or that it does not violate it.

(3) the internal relationship between the functional and operational descriptions of the method has to be established. One has to ensure that, given the assumptions the operational description describes a way to achieve the functionality. Because the description of the operational specification requires a logic over states, we use dynamic logic [Kozen, 1990] to formalize this obligation (that ensures the termination of the program and the desired functionality):

$$\models PSM_A \rightarrow (\langle PSM_o \rangle \text{ true} \wedge [PSM_o]\, PSM_f)$$

(4) a statement about the efficiency of the method has to be made. In the ideal case, given the assumptions, each alternative operational description requires at least the same effort as the chosen one to achieve the functionality of the method.[7] A simpler obligation is to proof a lower bound for the efficiency complexity of a chosen method (see [Straatman & Beys, 1995]).

4 Conclusions

Our paper tries to answer two questions: What are problem-solving methods and why are they necessary? In a nutshell, we provided the following answer: A PSM translates a declarative goal descriptions into a set of assumptions about domain knowledge required to achieve the goal in an efficient manner. The dichotomy of a declarative goal description and an efficient implementation must be bridged by a level where one rationalizes an efficient problem solver, that is, a problem solver with limited resources. A part of expertise is knowledge about achieving goals under bounded rationality. An operational description of a problem-solving method defines the appropriate level to elicit, acquire, interpret, and model this kind of knowledge. Assumptions about domain knowledge or the precise functionality are introduced, strengthened, or modified in order to achieve efficiency. The point of view on problem-solving methods as presented in our paper defines a number of research topics.

(1) An adequate framework for describing problem-solving methods has to be established: A formal notation for the functionality of a method is required. A logic over states is needed to express the operational specification of a method. This language must be able to express control over functionally specified basic building blocks. A formal notation for the assumptions is needed. A variant of meta-logic could be used to specify the assumptions of the method. Finally, a feasible calculus must be provided to specify the computational behavior of a method.

(2) A proof calculus is necessary that enables to prove relationships between the different parts of the specification of a method. A first step into this direction is achieved by [Fensel & Groenboom, 1995] where proof rules are defined for languages like KARL and $(ML)^2$. Based on these proof rules, automated support by theorem provers is possible. As the description formalisms include logic over states like dynamic logic, we will investigate the possibility to use theorem provers like the Karlsruher Interactive Verifier KIV [Reif, 1995] developed for program verification based on dynamic logic.

(3) Methods and tools are necessary that support the cyclic development of appropriate PSMs. This includes a library with problem-solving methods schema indexed by their functionality, assumptions, and cost, and operations working on assumptions and deriving PSM instantiations. [Van de Velde, 1994] defines three components of a modelling library: Modelling components are structures useful for the construction of complete models. Generic models are frames representing a class of complete models. Modelling operators transform a model into another one. Substantiating these ideas seems to be a promising research direction.

Acknowledgments. We would like to thank Ameen Abu-Hanna, Kees de Koning, Guus Schreiber, Annette ten Teije, Peter Terpstra and Bob Wielinga for helpful discussions that enabled this paper. We would also like to thank Manfred Aben, Jürgen Angele, Richard Benjamins, Gertjan van Heijst, and Rudi Studer for comments on earlier versions of the paper. We especially would like to thank Frank van Harmelen, for his contributions to earlier versions of the paper, and Joost Breuker, for discussions that stimulated most of the presented work.

7. This would require perfect rationality of the decision process that constructs the optimal problem solver with limited rationality.

References

[Akkermans et al., 1993] J. M. Akkermans, B. Wielinga, and A. TH. Schreiber: Steps in Constructing Problem-Solving Methods. In N. Aussenac et al. (eds.): *Knowledge-Acquisition for Knowledge-Based Systems*, Lecture Notes in AI, no 723, Springer-Verlag, Berlin, 1993.

[Benjamins, 1993] V. R. Benjamins: *Problem Solving Methods for Diagnosis*, PhD Thesis, University of Amsterdam, Amsterdam, The Netherlands, June 1993.

[Bredeweg, 1994] B. Bredeweg: Model-based diagnosis and prediction of behaviour. In [Breuker & Van de Velde, 1994], pp. 121—153.

[Breuker & Van de Velde, 1994] J. Breuker and W. Van de Velde (eds.): *The CommonKADS Library for Expertise Modelling*, IOS Press, Amsterdam, The Netherlands, 1994.

[Bylander, 1991] T. Bylander: Complexity Results for Planning. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI-91)*, Sydney, Australia, August 1991.

[Bylander & Chandrasekaran, 1988] T. Bylander and B. Chandrasekaran: Generic Tasks in Knowledge-Based Reasoning. The Right Level of Abstraction for Knowledge Acquisition. In B. Gaines et al. (eds.): *Knowledge Acquisition for Knowledge-Based Systems*, vol I, pp. 65—77, Academic Press, London, 1988.

[Bylander et al., 1991] T. Bylander, D. Allemang, M. C. Tanner, and J. R. Josephson: The Computational Complexity of Abduction, *Artificial Intelligence*, 49, pages 25—60, 1991.

[Chandrasekaran et al., 1992] B. Chandrasekaran, T.R. Johnson, and J. W. Smith: Task Structure Analysis for Knowledge Modeling, *Communications of the ACM*, 35(9): 124—137, 1992.

[David et al., 1993] J.-M. David, J.-P. Krivine, and R. Simmons (eds.): *Second Generation Expert Systems*, Springer-Verlag, Berlin, 1993.

[de Kleer & Williams, 1987] J. H. de Kleer and B. C. Williams: Diagnosing Multiple Faults, *Artificial Intelligence*, 32():97—130, 1987.

[Fensel, 1995a] D. Fensel: Assumptions and Limitations of a Problem-Solving Method: A Case Study. In *Proceedings of the 9th Banff Knowledge Acquisition for Knowledge-Based System Workshop (KAW´95)*, Banff, Canada, February 26th - February 3th, 1995.

[Fensel, 1995b] D. Fensel: Formal Specification Languages in Knowledge and Software Engineering, *The Knowledge Engineering Review*, 10(4), 1995.

[Fensel & Groenboom, 1995] D. Fensel and R. Groenboom: A Formal Semantics for Specifying the Dynamic Reasoning of Knowledge-based Systems. In *Proceedings of the Knowledge Engineering: Methods and Languages Workshop (KEML'96)*, January 15-16, 1996.

[Fensel & van Harmelen, 1994] D. Fensel and F. van Harmelen: A Comparison of Languages which Operationalize and Formalize KADS Models of Expertise, *The Knowledge Engineering Review*, 9(2), 1994.

[Goel et al., 1987] A. Goel, N. Soundararajan, and B. Chandrasekaran: Complexity in Classificatory Reasoning. In *6th National Conference on Artificial Intelligence* (*AAAI'87*), Seattle, Washington, July 13-17, 1987, pages 421—425.

[Kifer et al., 1995] M. Kifer, G. Lausen, and J. Wu: Logical Foundations of Object-Oriented and Frame-Based Languages, *Journal of the ACM*, 42:741-843, 1995.

[Kozen, 1990] D. Kozen: Logics of Programs. In J. v. Leeuwen (ed.), *Handbook of Theoretical Computer Science*, Elsevier Science Publ., B. V., Amsterdam, 1990.

[Landes & Studer, 1995] D. Landes and R. Studer: The Treatment of Non-Functional Requirements in MIKE. In *Proceedings of the 5th European Software Engineering Conference ESEC'95*, Barcelona, Spain, September 25-28, 1995.

[Levesque, 1984] H. J. Levesque: Foundations of a functional approach to knowledge representation, *Artificial Intelligence*, 23(2):155—212, 1984.

[Marcus, 1988] S. Marcus (ed.). *Automating Knowledge Acquisition for Experts Systems*, Kluwer Academic Publisher, Boston, 1988.

[Marcus et al., 1988] S. Marcus, J. Stout, and J. McDermott VT: An Expert Elevator Designer That Uses Knowledge-based Backtracking, *AI Magazine*, 9(1):95—111, 1988.

[Musen, 1992] M. A. Musen: Overcoming the Limitations of Role-Limiting Methods, *Knowledge Acquisition*, 4 (2): 165—170, 1992.

[Nebel, 1995] B. Nebel: Artificial intelligence: A Computational Perspective. To appear in G. Brewka (ed.), *Essentials in Knowledge Representation*.

[Poeck et al., 1996] K. Poeck, D. Fensel, D. Landes, and J. Angele: Combining KARL And CRLM For Designing Vertical Transportation Systems. In [Schreiber & Birmingham, 1996].

[Puppe, 1993] F. Puppe: *Systematic Introduction to Expert Systems: Knowledge Representation and Problem-Solving Methods*, Springer-Verlag, Berlin, 1993.

[Reif, 1995] W. Reif: The KIV Approach to Software Engineering. In M. Broy and S. Jähnichen (eds.): *Methods, Languages, and Tools for the Construction of Correct Software*, Lecture Notes in Computer Science (LNCS), no 1009, Springer-Verlag, Berlin, 1995.

[Rich & Knight, 1991] E. Rich and K. Knight: *Artificial Intelligence*, McGraw-Hill, New York, 2nd edition, 1991.

[Rouveirol & Albert, 1994] C. Rouveirol and P. Albert: Knowledge level model of a configurable learning system. In Lecture Notes in Aritificial Intelligence (LNAI), no 867 Springer-Verlag, Berlin, 1994.

[Schreiber & Birmingham, 1996] A. Th. Schreiber and B. Birmingham (eds.): *Special Issue on Sisyphus, The International Journal of Human-Computer Studies*, to appear, 1996.

[Schreiber et al., 1993] A. Th. Schreiber, B. J. Wielinga, and J. A. Breuker (eds.): *KADS: A Principled Approach to Knowledge-Based System Development, vol 11 of Knowledge-Based Systems Book Series*, Academic Press, London, 1993.

[Schreiber et al., 1994] A. TH. Schreiber, B. Wielinga, J. M. Akkermans, W. Van De Velde, and R. de Hoog: CommonKADS. A Comprehensive Methodology for KBS Development, *IEEE Expert*, 9(6):28—37, 1994.

[Steels, 1990] L. Steels: Components of Expertise, *AI Magazine*, 11(2), 1990.

[Straatman & Beys, 1995] R. Straatman and P. Beys: A Performance Model for Knowledge-based Systems. In M. Ayel and M. C. Rousset (eds.): *EUROVAV-95 European Symposium on the Validation and Verification of Knowledge Based Systems*, pages 253—263. ADEIRAS, Universite de Sovoie, Chambery, 26-28 June 1995.

[Tank, 1992] W. Tank: *Modellierung von Expertise über Konfigurationsaufgaben*, Infix, Sankt Augustin, Germany, 1992.

[ten Teije & van Harmelen, 1994] A. ten Teije and F. van Harmelen: An Extended Spectrum of Logical Definitions for Diagnostic Systems. In *Proceedings of DX-94 Fifth International Workshop on Principles of Diagnosis*, 1994.

[Terpstra et al., 1993] P. Terpstra, G. van Heijst, B. Wielinga, and N. Shadtbolt: Knowledge Acquisition Support Through Generalised Directive Models. In [David et al., 1993], pp. 428—455.

[Top & Akkermans, 1994] J. Top and H. Akkermans: Tasks and Ontologies in Engineering Modeling, *International Journal of Human-Computer Studies*, 41():585—617, 1994.

[van Harmelen & ten Teije, 1995] F. van Harmelen and A. ten Teije: Approximations in Diagnosis: Motivations and Techniques. In C. Bioch and Y.H. Tan (eds.), *Proceedings of the Dutch Conference on AI (NAIC'95)*, Rotterdam, June 1995.

[Van de Velde, 1994] W. Van de Velde: A Constructivist View on Knowledge Engineering. In *Proceedings of the 11th European Conference on Artificial Intelligence (ECAI'94)*, Amsterdam, August 1994.

[Wielinga & Schreiber, 1994] B. J. Wielinga and A. Th. Schreiber: Conceptual Modelling of Large Reusable Knowledge Bases. In K. von Luck and H. Marburger (eds.): *Management and Processing of Complex Data Structures*, Springer-Verlag, Lecture Notes in Computer Science, no 777, pages 181—200, Berlin, Germany, 1994.

[Wielinga et al., 1995] B. Wielinga, J. M. Akkermans, and A. TH. Schreiber: A Formal Analysis of Parametric Design Problem Solving. In B. R. Gaines and M. A. Musen (eds.): *Proceedings of the 8th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop* (*KAW-95*), vol II, pp. 31/1—37/15, Alberta, Canada, 1995.