A Case Study: Assumptions and Limitations of a Problem-Solving Method

Dieter Fensel
Department SWI, University of Amsterdam, Roeterstraat 15, NL-1018 WB Amsterdam, The Netherlands
phone: +31-20-5256791, e-mail: dieter@swi.psy.uva.nl

Institu AIFB, University of Karlsruhe, 76128 Karlsruhe, Germany

Subject of the paper: Expert systems, knowledge acquisition, knowledge engineering

Summary. The paper attempts a step in the direction of *competence theories* of reusable problem-solving methods for knowledge-based systems. In fact, we examine a variant of the psm *propose-and-revise*. The variant was developed as a solution to the current Sisyphus task (VT-task) which defined an elevator configuration problem. We decompose the entire method into its individual subtasks and examine each subtask for its underlying assumptions concerning the available domain knowledge. By doing this, we determine assumptions and limitations of the entire method. In addition, we examine how different control flows between these subtasks can influence the efficiency and effectiveness of the method. Again, we show how these differences are related to the assumptions of the method. Making these assumptions explicit defines interesting goals for validation and verification efforts: First, we show that under some circumstances two variants of a problem-solving methods behave equally well or that one is more suitable than another. Second, we have to prove for a given domain knowledge whether it fulfils the assumptions of a selected problem-solving method. We believe that such studies are necessary to allow the reuse of problem-solving methods and, conversely, that the reuse of psm`s justifies the effort of such activities.

Introduction

Two important requirements can be postulated for *second generation expert systems* (see David, Krivine and Simmons, Eds., 1993 for a survey): First, the separation of symbol level and knowledge level by using a set of models to describe a system independently from its implementation. Significant for knowledge-based systems is the model of expertise which describes the different types of knowledge required by such a system (see Schreiber, Wielinga, Akkermans, Van de Velde and de Hoog, 1994). Second, the use and reuse of generic problem-solving methods to describe the reasoning process of a knowledge-based system (see Breuker & Van de Velde, Eds., 1994).

In the past years, significant progress has been made to support the specification of a model of expertise (a survey is given in Fensel and van Harmelen, 1994). Several knowledge specification languages have been developed which can be used to formally (and operationally) describe a model of expertise or similar types of conceptual models. These languages allow a precise and unambiguous specification of knowledge-based systems at the knowledge level and support the evaluation of a specification by testing and theorem proving. Compared to the amount of work which has been done in formal foundations of conceptual descriptions of knowledge-based systems less research effort has been spent in supporting the *reuse of generic problem-solving methods* by formal methods. Since Clancey´s (1985) description of the problem-solving method heuristic classification several additional generic problem-solving methods have been abstracted from different applications but they are either represented by implemented pieces of code or by informal textual descriptions.

The most popular approach on reusing generic problem-solving methods with several applications is the *role-limiting method* approach. For example, MOLE, SALT, SIZZLE (see Marcus, Ed., 1988), D3/CLASSIKA (Puppe, 1993), or PROTÉGÉ (Musen, 1989) are expert-system shells with a fixed problem-solving method. These expert-system shells can be

applied to a fixed type of tasks. The problems of this approach are:

- A high effort is required to develop a knowledge acquisition tool for every problem-solving method, i.e., for every type of task.
- Tasks that require the combination of several problem-solving methods are not supported.

To overcome these shortcomings of application generators, Chandrasekaran and Johnson (1993); Marques, Dallamagne, Klinker, McDermott and Tung (1992); Musen (1992); Poeck & Gappa (1993), and Steels (1990) propose libraries of reusable methods or generic tasks. These methods have a finer grain size than conventional expert system shells. A complete problem-solving process must be modelled by several methods. These approaches are analogous to the *source-code libraries* idea in software engineering (see Krueger, 1992). The main characteristics of these approaches is that currently these methods are only described by code and by informal descriptions of the code. Therefore, there is little support for the selection, specialization, and integration of these methods. The lack of descriptions that abstract from implementational details, but do not lead to imprecise natural-language descriptions, makes it difficult to compare such methods, and to provide a precise description of their problem-solving ability.

A different level of reuse is provided by the results of the KADS-I and CommonKADS projects. Libraries of *semiformally* specified reusable problem-solving methods for a broad variety of tasks are given in Breuker, Wielinga, van Someren, de Hoog, Schreiber, de Greef, Bredeweg, Wielemaker & Billault (1987); Breuker & Van de Velde, Eds. (1994); and Benjamins (1993). As these problem-solving methods are only described informally reuse is provided only at a conceptual level. In addition, natural language descriptions lack from preciseness and disambiguity which makes it very hard to judge the semantics of a specific problem-solving methods as well as its applicability for a given task and domain.

For reusing these problem-solving methods, one must be aware of the assumptions underlying such a method. Each method defines requirements on the available domain knowledge and the given task. These requirements can concern the effect of the method (i.e., its correctness) and the efficiency of it. Making these assumptions explicit several advantages:

- First, the can be use to prove properties of a specification. The formal specification of a method together with the specification of its assumptions should enable to formally prove correctness and efficiency of such a method. Only reliable components can be reused.
- Second, these assumptions must be proven to hold in a given domain if the method should be applied to it. Otherwise, reuse of given methods is very questionable as it is not at all clear whether they will produce a correct result (in an efficient manner).
- Third, these assumptions can be used to index the reusable components and so support the selection and adoption of methods for a given domain and task.

A progress in this area requires two different research activities. First, a conceptual framework for describing the functionality, the efficiency, the dynamic behavior, and the assumptions of reusable building blocks. Second, as for validation and verification of formal specifications, a proof calculus is required which enable proofs like:

- Correctness proof:

  $\phi_{assumptions} \wedge \phi_{method\text{-}description} \models \phi_{functionality}$

- Applicability proof:

  $\phi_{domain\text{-}layer} \models \phi_{assumptions}$

- Efficiency proof:

  $\phi_{assumptions} \wedge \phi_{method\text{-}description} \models \text{Effort}(\phi_{method\text{-}description}) < \alpha$

The paper is aimed at being a step toward determining a *competence theory of a problem-solving method* (see Akkermans, Wielinga and Schreiber, 1993; Wielinga, Akkermans and Schreiber, 1995) by analysing the psm *propose-and-revise*. Akkermans et al. (1993) and Wielinga et al. (1995) propose a general framework for developing reusable problem-solving methods by successive conceptual refinement. Starting from a task description, the final problem-solving method is derived in a top-down manner by introducing additional assumptions about the available knowledge and the precise nature of the task. But still, these proposal lack from significant detail. Therefore, we want to add more preciseness to this enterprise by re-engineering an existing specification of propose-and-revise and by analysing its implicit and explicit assumptions.

In fact, we used the KARL specification of propose-and-revise as given by Poeck, Fensel, Landes and Angele (1994) as an input for our study. We tried to detect as many assumptions of this method as possible. It is clear that many of them are not related to the propose-and-revise method in general, but to the task-specific (and domain-specific) variant we specified. Otherwise, it is hard to justify the assumptions of a method described only informally as in Marcus (1988) and Marcus, Stout and McDermott (1988). In fact, one of the results of this paper is to outline the way in which different variations of the same psm have different assumptions or, conversely, how different assumptions holding in a domain can be used to derive the appropriate variant of propose-and-revise for a given application. By making assumptions explicit, it becomes possible to check whether an application domain really fits to an available psm and, conversely, which of the several variants of the methods fits (best) to it. We assume such a description as an absolute necessity, especially if the psm`s are designed for reuse.

The KARL specification which we used as an empirical resource for our case study was the result of a reengineering activity. First, a configurable-role-limiting-method shell (Poeck and Gappa, 1993) for *propose-and-exchange* (see Poeck and Puppe, 1992) was adopted to

propose-and-revise according to its informal description by Yost (1992). Then a formal specification of the reasoning process of this shell was provided in KARL (see Fensel, 1995). Finally, we examined the KARL specifications for assumptions which are implicitly encoded in it. These assumptions which we detected do not reflect specific features of KARL, but are based on (implicit) decisions which were made by the shell authors Poeck & Puppe (1992) or by the VT-task description in Yost (1992). The KARL specification was just a precise and unique description of the problem-solving process which abstracted from implementational details. As the analysis of hidden assumptions in the specification was mainly a conceptual activity, the conceptual model underlying a KARL specification was very helpful. Again, it was the *integration* of a specification at the conceptual level (based on the KADS model of expertise) and at the formal level (which eliminates ambiguity and impreciseness of informal specifications) which provided the necessary input for our undertaking.[1]

The paper is organized as follows. First we give a conceptual description of our variant of propose-and-revise which we called *Select-Propose-Check-Revise Method (SPCR-Method)*. Then we examine the different assumptions underlying each of these four steps. Finally we provide an look at further work. The reader is assumed to be familiar with the propose-and-revise method as described by Marcus et al. (1988) and Yost (1992).

Asked to give the message of the paper in a nutshell, we would answer:

- We show several assumptions and limitations of the psm. *Making these assumptions explicit is necessary for the reuse of psm`s*. First, a given domain and task must be checked to see whether these assumptions are fulfilled. Second, these assumptions can also be used to solve the indexing problem of problem-solving method as they can be

1. Fensel, Eriksson, Musen and Studer (1993) started this undertaking for the *Board-game method*. This method can be applied to similar tasks.

used as guidance for selecting a suitable psm for a given application.

- *We show how assumptions about the available domain knowledge lead to different control flows (task layers) for the same psm (i.e., for the same inference structure).* These assumptions can therefore be used to derive variants of a psm. Or in other words: The defined control flow of a method is just an (implicit) way to express its assumptions about the domain knowledge.

A similar study is reported by Zdrahal and Motta (1995). Their in-depth analysis of propose-and-revise elicits assumptions of the method which will also be discussed in our paper. The main difference between the two papers lies in their interests. Zdrahal and Motta (1995) focus on how to improve the efficiency of a propose-and-revise method by combining it with constraints-satisfaction techniques. They examine the application of efficient algorithms and generic heuristics for individual inference steps and for the entire method. From our point of view this is an issue which is more related to the design activity of an expert system.[2] The purpose of the model of expertise is to establish a conceptual framework for eliciting and interpreting the expert knowledge required for effective and efficient problem solving. The purpose of the design activity is to develop an efficient realization by adding appropriate algorithms and generic heuristics (see Landes, 1995). But it should be clear that it is not possible to determine the exact borderline between the two activities.

1 A Knowledge Level Sketch of The Select-Propose-Check-Revise Method

The SPCR-Method is applicable for assignment tasks where values are to be given to a set of parameters fulfilling several constraints. Examples for this type of task are:

2. Landes (1995) introduces a Design model for KARL. For example, declarative descriptions of elementary inference steps like *propose* are replaced by efficient algorithms and data structures which achieve the same functionality in an efficient manner.

- Sisyphus-I (see Linster, 1994), where employees are assigned to places (i.e., values).
- ECAI´92 workshop example (see Treur & Wetter, 1993): a simple scheduling task where activities have to be assigned to time slots (i.e., values)
- Sisyphus-II or VT-task (Yost, 1992; Schreiber & Birmingham, 1994), where an elevator is configurated by choosing components and assigning values. Actually the problem is viewed as a parametric design problem. That is, the design artefact is describedby a set of parameters and the design process must determine a value each of them which fulfill requirements and constraints.

The entire method decomposes the whole task of assigning values to a set of parameters in four subtasks (see Figure 1):

- A *select* subtask chooses the parameter which should be processed next.
- A *propose* subtask proposes a value for the selected parameter.

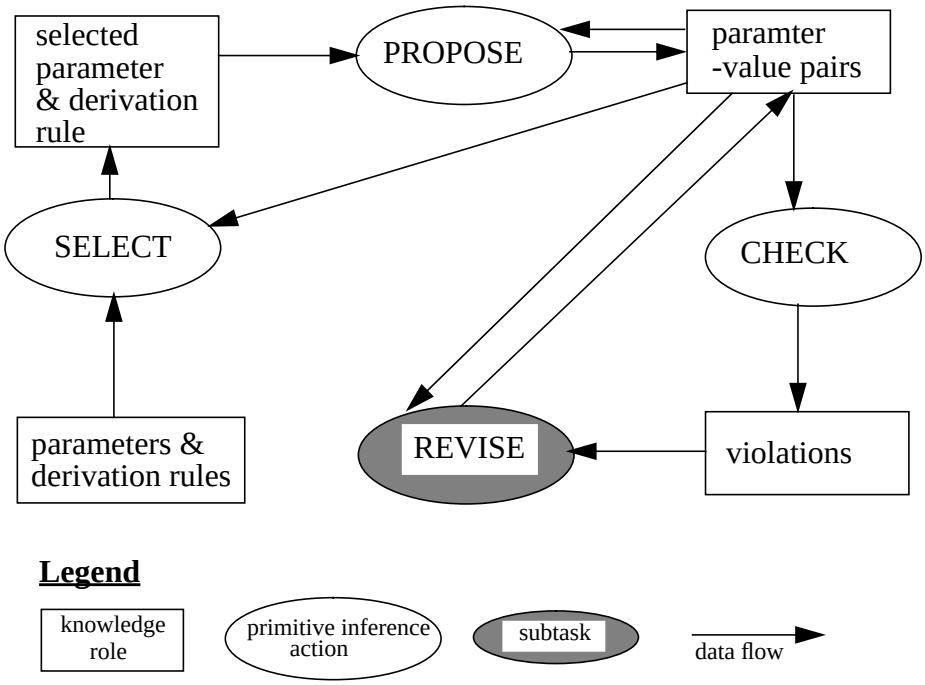

Fig. 1 dataflow diagram of the SPCR-Method

- A *check* subtask checks the currently given partial assignment (i.e., the old one which is enriched by the new parameter-value pair) as to whether it fulfils the given constraints.
- A *revise* subtask corrects the partial assignment if constraint violations were detected by the *check* step.

Figure 1 gives a dataflow diagram of the entire method. We model *select*, *propose*, and *check* using primitive inferences. That is, we are neither interested in breaking them down into subtasks of finer grain size nor do we want to impose internal control on these steps. *Revise* is a more complex subtask. We therefore want to model its internal subtasks and their control at the knowledge level. *Revise* is modelled by a composed inference which will be refined later. The question may arise as to whether check is really an inference step or more a kind of branching condition for the control flow which is represented at the task layer. But it fulfils a twofold purpose:

- If constraint violations can be detected, the method must apply the *revise* step. If not, the method can go on by choosing the next parameter in the *select* step.
- This step collects all violated constraints and provides this as an input for the *revise* step. Therefore, there is a dataflow from the *check* step to the *revise* inference step and we chose to model it using an inference action.

The control flow of the SPCR-Method is shown in Figure 2. It consists of a loop over the four subtasks and a branch depending on the necessity of revising a partial solution. Written in dynamic logic (cf. Harel, 1984; Kozen, 1990) enriched with some syntactical sugar, we obtain the following expression:

while ¬all? **do** select; propose; check; **if** violations? **then** revise **fi od** (1)

It should be noted that (1) is not the only possible control flow. Another possibility is given

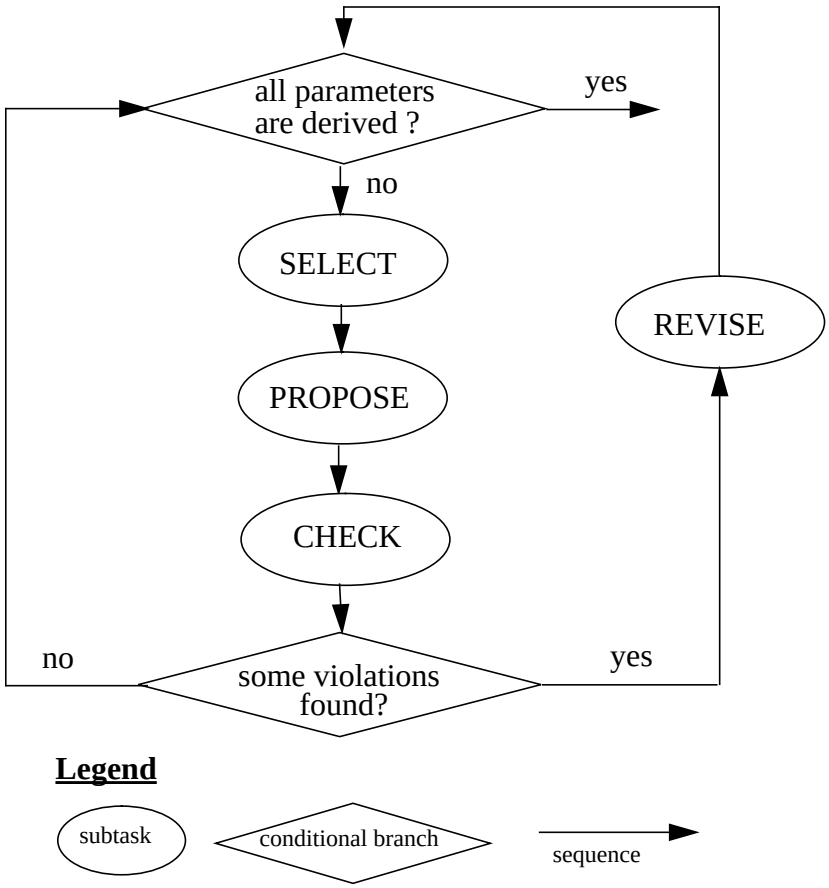

Fig. 2 Control flow of the SPCR-Method

by (2).

while ¬all? **do** select; propose **od** check; **while** violations? **do** revise **od** (2)

Zdrahal and Motta (1995) call (1) *Extend Model then Revise* and (2) *Complete Model then Revise*. The obvious difference between (1) and (2) concerns the select and propose step. Whereas (1) selects and derives only one parameter and its value, (2) selects and proposes all parameters and their values before starting the revision activity. That is, (1) revises incomplete assignments whereas the revision step in (2) only works on the complete (but incorrect) assignment. A further possibility, which has not been mentioned yet, lies between those given above. In this model the *select* step would not deliver one or all, but some parameters. The select step would select all parameters which can be given a value in the next step. That is, it would regard all parameters with a propose rules which only depends on

already derived parameter values.

It should become clear that it would be very helpful to have proven lemmas in regard to the relationships of the different variants of the SPCR-Method as sketched above. How are the effect and efficiency of the different methods related? Based on a particular assumption one could show that some variants lead to the same results (but differ in their efficiency) and some produce different outputs. Making these assumptions explicit provides us with a strong guidance in selecting or differentiating an appropriate variant of the method for a given task and application domain.

In the following, we will examine each subtask and its assumptions about the available knowledge in more detail. In fact, we will have to examine four different types of knowledge.

- *Select* knowledge:

  We need knowledge to select a parameter. In our case, we assume a network formed by propose rules. This domain-specific meta-knowledge must fulfil some requirements if our method is to function correctly. This knowledge is meta-knowledge as each propose rule presents a piece of (domain-specific) object-knowledge, whereby we use the knowledge which is encoded in the relationship of these rules.

- *Propose* knowledge:

  We need knowledge which enables us to derive a value for a given parameter. Our method assumes a user input or a propose rule for this purpose.

- *Check* knowledge:

  We require knowledge which can be used to check a partial solution to determine whether it is correct. In our case, we assume a set of constraints over the parameter values.

- *Revise* or repair knowledge:

We require knowledge which can be used to repair a partial solution if constraint violations were detected. In our case, we assume a set of fixes.

2 Assumptions About the Select Knowledge

The *select* step selects the parameters which should receive a value next. In addition, the derivation rule which can be applied for this parameter must be selected. This derivation rule can be either a user input or a propose rule.

In our case, we select one parameter per time. For this purpose we need generic and domain-specific knowledge.We use the following *generic heuristics*:

- Select non-deterministically a (not already chosen) parameter for which user input is given;
- if no further parameter of this kind exist, choose non-deterministically a (not already chosen) parameter for which a propose rule is given whose premises depend only on already derived values

Two assumptions appear.[3] First, we show which implicit assumptions are encoded by our generic heuristics. Second, we need to know the relationship between the propose rules and user input parameters to select the parameters. This is *domain-specific meta knowledge*. Our method makes some strong explicit assumptions about this knowledge:

2.1 Assumptions of the Generic Heuristics

As we select one possible parameter in a non-deterministic manner, we do not require any

3. Actually, two shortcomings of our specification language KARL also became visible. KARL could neither specify non-deterministic choice nor domain-specific meta-knowledge in an elegant manner (see Poeck et al., 1994).

knowledge for this step. This also means that we could not use just any available knowledge for this step. In fact, we implicitly assume that the selection of the next parameter neither influences the effect nor the efficiency of the problem-solving process.

An initial alternative is to regard all parameters which can be given a value in one step. This value can either be provided by a user input or by a propose rule, which can be applied as all its premises have given values. Instead of choosing one out of all those which can be determined next, we would deal with them parallel. We could give all of them a value in the *propose* step and then *check* and *revise* their constraint violations. Actually we believe that this a very meaningful method as there is no real reason for selecting exactly one parameter for the next step. The only reason is perhaps that the psm was derived by an expert (see Marcus et al., 1988) and that humans tend to (artificially) sequentialize activities, as they are not very good at parallel problem-solving. However, computers are great in doing several things parallel, and sequentiality should be eliminated when not required by the problem.[4]

The second alternative to non-deterministic choice would be to assume any domain-specific heuristics which guide the selection process of the next parameter. Such a heuristics would be a reason for selecting not all possible parameters which can be given a value in the next step but just some (or one) of them.[5]

2.2 Assumptions About the Domain-specific Meta-knowledge

Our method makes assumptions about the available domain-specific meta knowledge. An

4. In fact, we see here how assumptions about the agent which will carry out the computation at the symbol level (i.e., whether humans or computers are the agents) influence the knowledge level model of expertise.

5. We already mentioned the third alternative, which would apply the *propose* step several times as long as all values are derived. But this does not only concern the *propose* step but also the control flow of the entire method.

initial straightforward restriction would be to allow only one propose rule or user input per parameter. We could then depict each propose rule by a set of directed arcs which connects the premises of a rule with its conclusion (see Figure 3). The nodes of the directed graph would be the parameters and one start node would be the *user*. The user inputs would be directed arcs from the user-node to parameter nodes. The assumptions of the *propose* step can now be related to that graph:

- (1) The user must give an input for each parameter he is assigned by the graph.
- (2a) It is forbidden to have two or more propose rules for the same parameter.
- (2b) It is forbidden to have a user input and a propose rule for the same parameter.
- (3) Each node must be reachable (strongly connected).
- (4) The graph formed by the propose rules must be non-cyclic.

(1) relates the graph with the external world. The user has to give a value if he is assigned. The method can therefore not work with incomplete input. The user does not have to assign a value to all parameters, but to all parameters he is assumed to do. (2a+b) should prevent

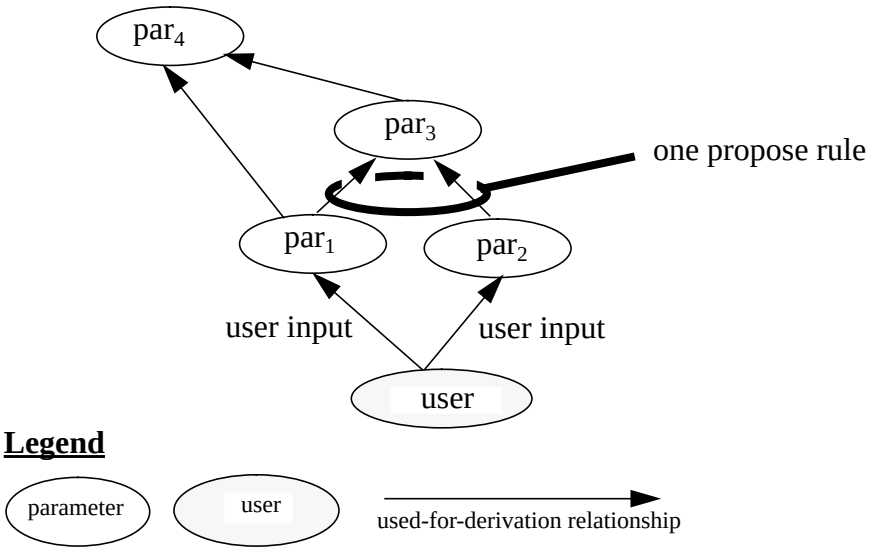

Fig. 3 The restricted graph of propose rules

contradictions. Only one propose rule or the user should be used to derive a value for a parameter. In fact, we could not even present two different propose rules in our graph, as this difference would disappear. It is easy to statically check assumption (2a+b). Assumption (3) can also be checked statically. We just have to check, whether each node has an input arc. Assumptions (4) can also checked statically, but requires more computational effort. We have to ensure that each permissible path through the graph does not include cycles. (4) allows us to stratify the set of parameters to get an order in which we select and propose values for them. A parameter which depends on user input is at level equal zero. A parameter *p* is at level *i*

$\text{level}(p) = i \text{ with } i > 0$

if it is not given by a user input and if it depends on parameter $p_1,..., p_n$ with

$\max(\{\text{level}(p_j) \mid j = 1,...,n\}) = i - 1.$

This stratification is not possible if a parameter depends directly or indirectly on itself. Parameters which directly or indirectly depend on themselves would never be selected by our psm.

A significant restriction of this representation is that only one propose rule per parameter can be given. This is not fulfilled by the VT-domain. Different derivation rules for further elevator parameters exist depending on the selected subcomponent of an elevator [Yos92].

For example, if we have chosen the motor model *10HP* then

$$\text{(1) peak} = \begin{cases} 1.25 * M, & \text{if } M \leq 40 \\ 1.333 * M - 3.333, & \text{if } 40 < M \leq 62.5 \\ 1.6 * M - 20, & \text{otherwise} \end{cases}$$

but in the case of the motor model *15HP* we have the propose rule

$$\text{(2) peak} = \begin{cases} 1.286 * M, & \text{if } M \leq 70 \\ 1.4 * M - 8, & \text{if } 70 < M \leq 120 \end{cases}$$

$\lfloor$ 1.6 * M + 60, otherwise.

M denotes the maximum motor torque.

There are several propose rules for the same parameter and the applicability of a propose rule depends on the values of other parameters. Therefore, we have to use a different representation of the domain-specific meta-knowledge which makes less strong assumptions about this knowledge. Figure 4 shows a Petri net representation of the propose rules where each rule corresponds to a transition and each parameter is represented by a node. Actually, we have to use predicate-transition nets, as the transitions are labelled by conditions and the tokens are of different types.

We still require that the propose rules can be used to define a partial ordering on the parameters so as to guide the order in which they are selected, but proving this assumption now requires more effort. Now we have to prove the behavior of a predicate-transition net. We no longer have to only regard static properties, but we have to reason in accordance to already selected parameters *and their derived values* if we want to prove this assumption. The question as to whether all parameter are reachable is also much more complicated. We have to check whether under some circumstances (chosen parameter values) a path in the Petri net will never be reached and parameters will remain without a value. We now have to consider all possible admissible executions of the Petri net formed by the propose rules. This is the price we have to pay for relaxing the very strong assumptions of the first graphical representation.

2.3 Guidance of the Knowledge Acquisition Process by Assumptions: Creating Fixes to Resolve Cycles

We assume the graph of propose rules to be non-cyclic. A parameter should neither directly nor indirectly depend on its own value, as this would lead to an inconsistency in regard to its value.[6] This assumption can be used as guidance in the course of knowledge acquisition. If

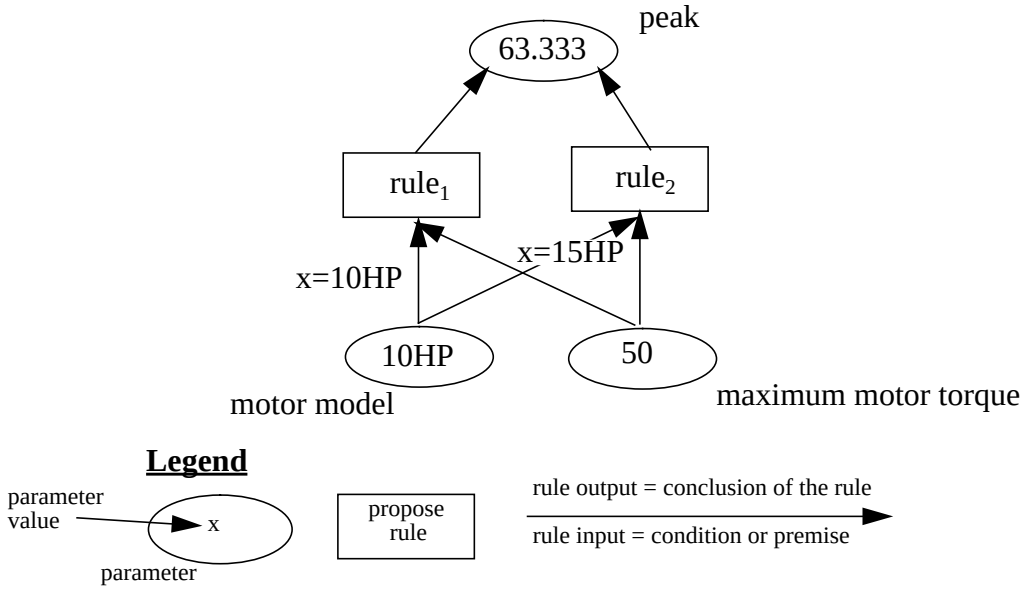

Fig. 4 The representation of the propose rules graph by a Petri net.

we detect such direct or indirect cycles we can ask the expert about them. In general, one alternative exists:

- Our chosen variant of our (chosen) psm makes assumptions which are not fulfilled by

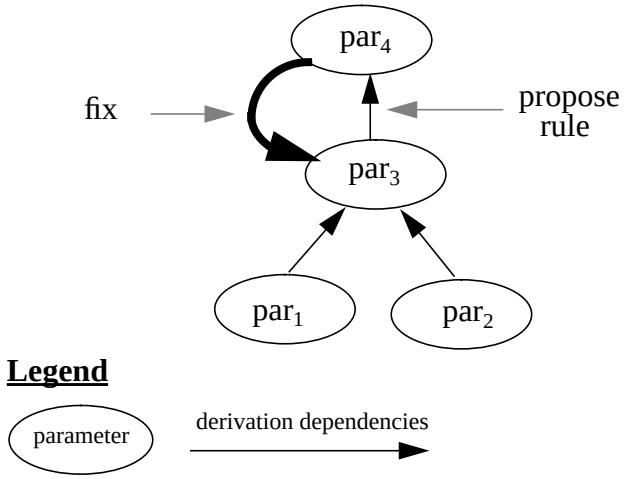

Fig. 5 Resolving cycles in the propose graph.

6. Not in general, but given the way in which our method would process the case. A parameter which depends directly on itself would not be assigned a value at all. A parameter which indirectly depends on itself is a candidate in two *propose* steps and is assigned a second value. This would lead to an inconsistency if the later value differs from the former one.

the domain. Then we have to choose a different variant.[7]

- We have to resolve the conflict. We delete one propose rule, or we realize that one rule was not intended so much as a propose rule, but more as a fix which can be used to repair an assignment when constraint violations appear (see Figure 5). In that way, the assumption would be used as a guidance for reformulating parts of the domain knowledge during knowledge elicitation and interpretation.

3 Assumptions about the Propose Knowledge

After having selected the next parameter we have to give it a value. For this we need a user input or a propose rule. More precisely, the result of the selection step is not only the parameter which should obtain a value next, but also the selected propose rule (or the user input) which can be used for this purpose. The main assumption of the propose step concerns the *uniqueness* in the graph of the propose rules: there must be only one admissible path (i.e., propose rule) in our graph to reach the current parameter. This assumption ensures that the *propose* step delivers a unique value for the parameter.

We could easily assume a different domain where this assumption does not hold. That is, several possible propose rules which are applicable for a parameter and we have to choose one of them. Again, this could be done non-deterministically, we could assume the existence of some heuristic knowledge, or we could regard several different assignments parallel. Again, if we were to make a *weaker* assumption then the question, for example, of whether under all circumstances all parameters still remain reachable would become even *more difficult*.

7. We could also choose a different application domain.

4 Assumptions about the Check Knowledge

The check knowledge is formed by constraints over the parameter values. Our first assumption is that there exists a solution. That is, the constraints define a solvable problem. This can be proven by showing that at least one possible combination of parameter values exists which fulfils every constraint.

Actually, the scope of our psm is much more limited. As we do not search completely, we cannot guarantee that we will find a solution even if one exists. In fact, we assume that the constraints define a problem which can be solved by our limited search. We use hill-climbing with limited (i.e., goal-directed or knowledge-based) backtracking by fix knowledge. This assumption does not define a requirement for the constraints alone but for the relationship between constraints and fixes. The fixes must be strong enough to find a solution and the constraints must be "weak" enough to enable us to find such a solution.

A strong limitation of our methods concerns the way we deal with constraints. All constraints are viewed as strong constraints. It is neither possible to express weak constraints which may be violated nor to place a penalty function on them. We cannot differentiate between an assignment which violates one unimportant constraint slightly and an assignment which violates several important constraints significantly.

A second strong shortcoming of our check step is that we do not assume the existence of knowledge which would enable us to distinguish between different solutions. We have no knowledge as to whether one of them is more preferable than the other. We only check whether an assignment fulfils all constraints. But it is clear that, in general, different solutions which fulfil all constraints exist but that they are of different quality.

5 Assumptions of Revise Knowledge

Figure 6 shows the refinement of the *revise* subtask. It consists of five subtasks where four

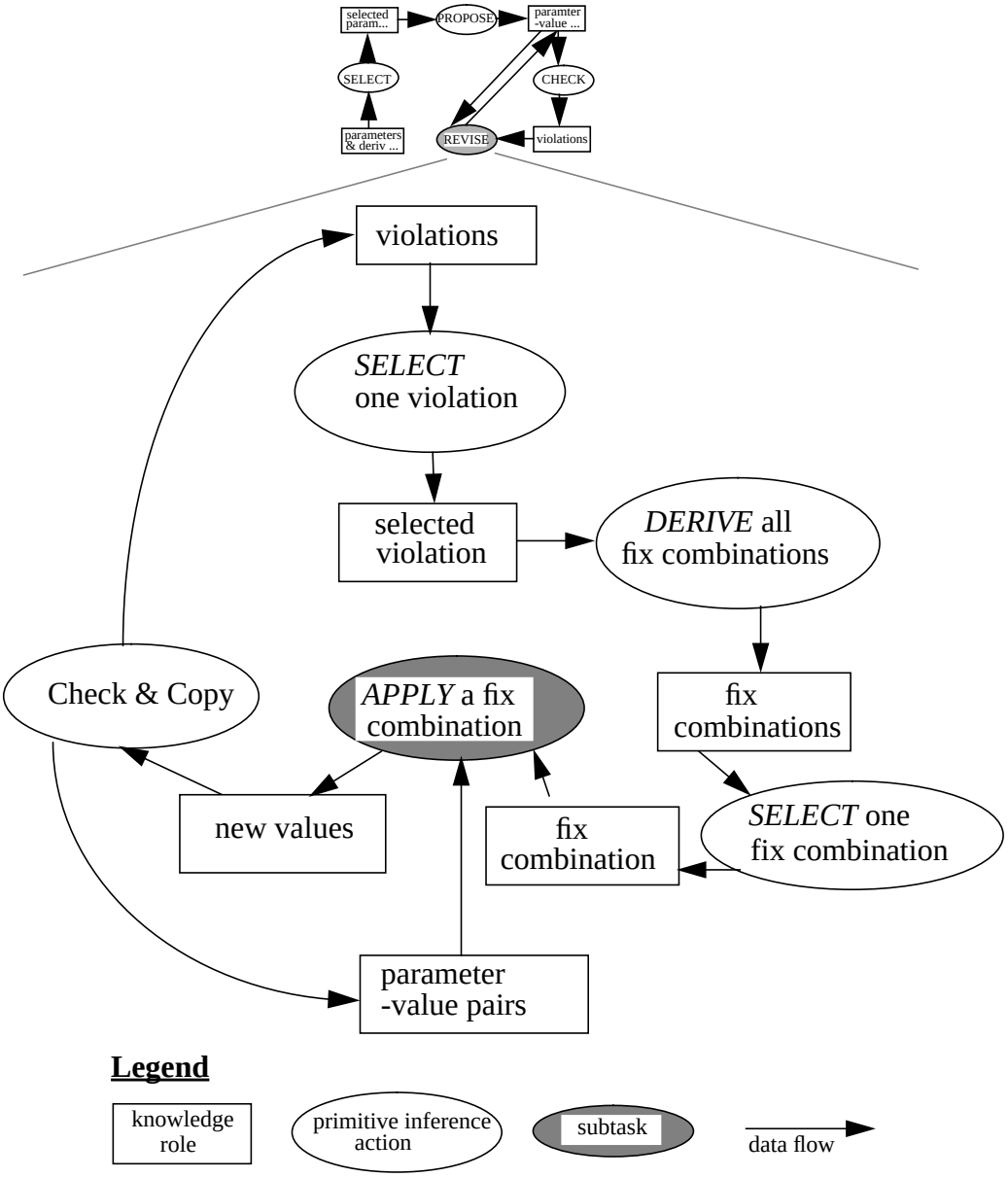

Fig. 6 Data flow of the *revise* subtask.

are solved by primitive inferences and one is again a composed inference action. The substeps of the *revise* step are:

- *Select* one violated constraint from the set of all violated constraints.
- *Derive* all fix combinations for the selected violation.
- *Select* one fix combination.

- *Apply* the fix combination.
- *Check* the new derived assignment to see whether it only violates a subset of the original violated constraints. If yes, regard it as a new assignment with new violations.

The control flow of the *revise* step is defined in Figure 7. First a violated constraint is chosen. Second, all possible fix combination which can be use to repair this violation are derived. Third, a fix combination is selected and applied until the violation is repaired (or until no further applicable fix combination exists). Then the new values and constraint violations are used as a starting point for the next iteration until all conflicts are resolved. An implicit assumption is that we can always find a solution. The search process is complete in the sense that it tries to resolve a violation as long as a further possible fix combination exists. But the search is incomplete as it does not backtrack behind the selection of the currently investigated constraint violation.[8] The selection cannot be undone even if the search runs into a dead end. Also, a successful fix application cannot be undone. The implicit assumption is that this restricted search is powerful enough to find a solution (because of the available domain knowledge) and that more powerful search strategies would reduce the efficiency of the problem-solver. On the other hand, if no solution can be found at all, backtracking behind the selection step (i.e., the investigation of different orders in which the violations are chosen) should be possible.

In the following, we will examine the different assumptions of the single steps of the entire revise subtask.

5.1 Assumptions of Select-a-Violation Knowledge

We select non-deterministically a constraint from all violations. Again, no assumption is

8. In fact, as we will see in the apply-a-fix-combination step, the search is also incomplete in the sense that only the first successful fix combination can be applied.

```
while ¬empty-violations? do
    Select one violation;
    Derive all fix combinations;
    Select the first fix combination;
    Apply the fix combination;
    while ¬Check? do
        Select the next fix combination;
        Apply the fix combination
    od
    Copy;
od
```

Fig. 7 Control flow of the *revise* subtask.

made about available knowledge but we implicitly assume that the sequence has no influence on the effect and efficiency of the psm. As there is no backtracking behind this selection step, this is a very crucial assumption.

5.2 Assumptions of Derive-All-Fix-Combinations Knowledge

For each (violated) constraint *c* a set of applicable fixes $F_c = \{f_1,..., f_n\}$ must be given which can be used to repair the violation. In the VT-domain, we had three types of fixes: replace a component by another, change a value, increment or decrement a value. As increment or decrement fixes can be applied several times, F_c is actually the transitive closure of the given fixes, i.e.,

$$F_c = \{(f_1)^*,..., (f_n)^*\}$$

A repair activity is defined by any possible (non-empty) subset of F_c. An example from [Yos92] should illustrate this.

Counter weight over travel: For safety reasons, the *counter weight overtravel* must be at least $R + 1.5 * S + 6$ inches, where *R* is the *car runby* and *S* is the *car buffer stroke*. If it is not, four fixes are possible:

- F_1: Decrease the *counterweight bottom reference* by the amount by which the

overtravel falls short of its minimum. This is a change-a-value fix.

- F_2: Decrease the *counterweight frame height* in one inch steps. This is a decrease fix.
- F_3: Increase the *overhead* by the amount the *overtravel* falls short of its minimum. This is a change-a-value fix.
- F_4: Decrease the *pit depth* in one inch steps. This is a decrease fix.

$F = \{F_1, 1 * F_2, 2 * F_2, 3 * F_2, ..., F_3, 1 * F_4, 2 * F_4, 3 * F_4, ...\}$

A strong assumption of our psm is that there must be *finitely many fix combinations* and each *fix combination* must be finite. This is due the fact, that we derive all possible fix combinations and this can only be done in finite time under the two given assumptions. In the VT case, additional constraints which restrict the legal domain of a parameter value were introduced for this purpose. These constraints are handled differently from normal constraints. Normal constraints are used to determine whether a partial assignment is valid. The new introduced constraints are used in the Derive-All-Fix-Combination step to define upper and lower bounds for the application of increment and decrement fixes.

In [Yos92] a constraint is given to define a lower bound for the application of F_2. *Counterweight frame height*: The *counterweight frame height* must be between 90 and 174 inches, inclusive. The maximal number of times fix F_2 can be applied is therefore determined by the difference between the current value of the *counterweight frame height* and 90 inches. Such additional constraints, which ensure finiteness of fix applications, have no fixes of their own (i.e., they are always assumed to be true). A similar constraint is given the constraint fix F_4.

5.3 Assumptions of Select-one-Fix-Combination Knowledge

In this step, we select one fix combination out of all possible fix combinations for one

constraint violation. For this purpose, we used a cost model of the VT-domain. Therefore, we do not have to non-deterministically choose an element but we can use this domain-specific heuristics. Actually, there were ten cost levels for grouping the fixes. Examples are [Yos92]:

- level one: no problem in applying the fix;
- level two: applying such a fix increases maintenance requirements;
- ...
- level nine: changing the building dimensions;
- level ten: changes major contract specification.[9]

After having assigned each fix to a cost level we have to determine the total cost of a fix combination and define an order for them which we then use to determine the cheapest fix combination. Each fix combination *f* is described by a ten-tuple (c_1,..., c_{10}) where c_i is the number of fixes at cost level *i* which are contained in the fix combination *f*. A lexicographical ordering of these fix combinations is defined by:

A fix combination f' with (c'_1,..., c'_{10}) is more costly than *f* if there is an *i* with

$c_i < c'_i$ and

no $c'_j < c_j$ for each $i < j$.

This cost model can be modelled by ordinal numbers $\{o_1,..., o_{10}\}$ where

- $n * o_i < o_j$ for each $i < j$
- $n * o_i < m * o_i$ if $n < m$.

9. Therefore, preference knowledge concerning solutions is expressed as preference during for revision steps.

```
while ¬empty-violations? do
    Select one violation;
    Derive one fix combination;
    Apply the fix combination;
    while ¬Check? do
        Derive next fix combination;
        Apply the fix combination;
    od
    Copy;
od
```

Fig. 8 Alternative control flow of the *revise* subtask.

In the following, we will discuss how different assumptions about this selection heuristics lead to differences in effect and efficiency of different control flows for the *revise* subtask. In Figure 7 we already gave one possible control flow for the *revise* subtask. An alternative flow is given in Figure 8. The method with the second control flow has the *same effect* but is *much more efficient* as it does not derive all possible fix combination but only those which are required. Actually, here was the only significant difference between the KARL specification of the SPCR-Method and its implementation by a configurable role-limiting shell[10] (see Poeck et al., 1994). This kind of efficiency was regarded as a non-functional requirement which can be added during design and implementation but which is of no concern during the

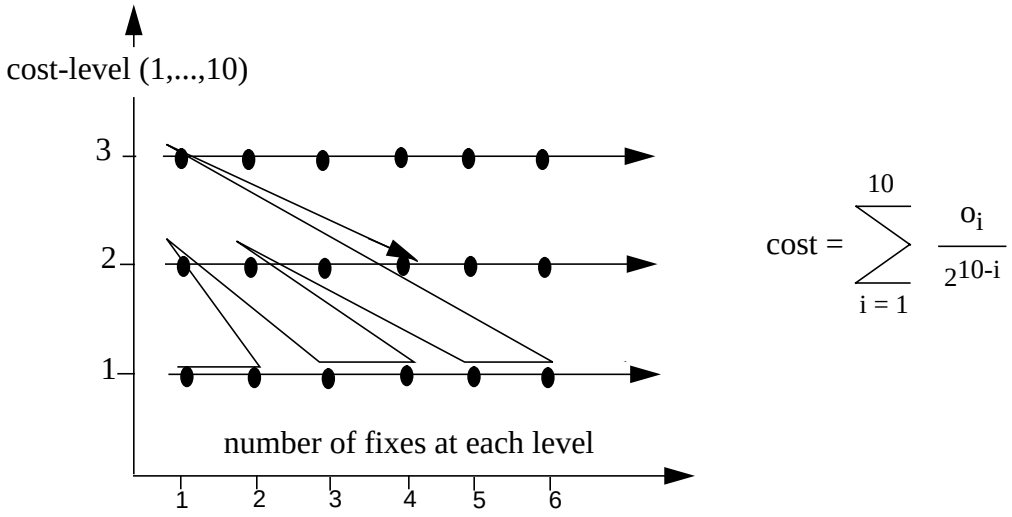

Fig. 9 An alternative cost model for the *select-a-fix-combination* step.

10. See Poeck and Gappa (1993).

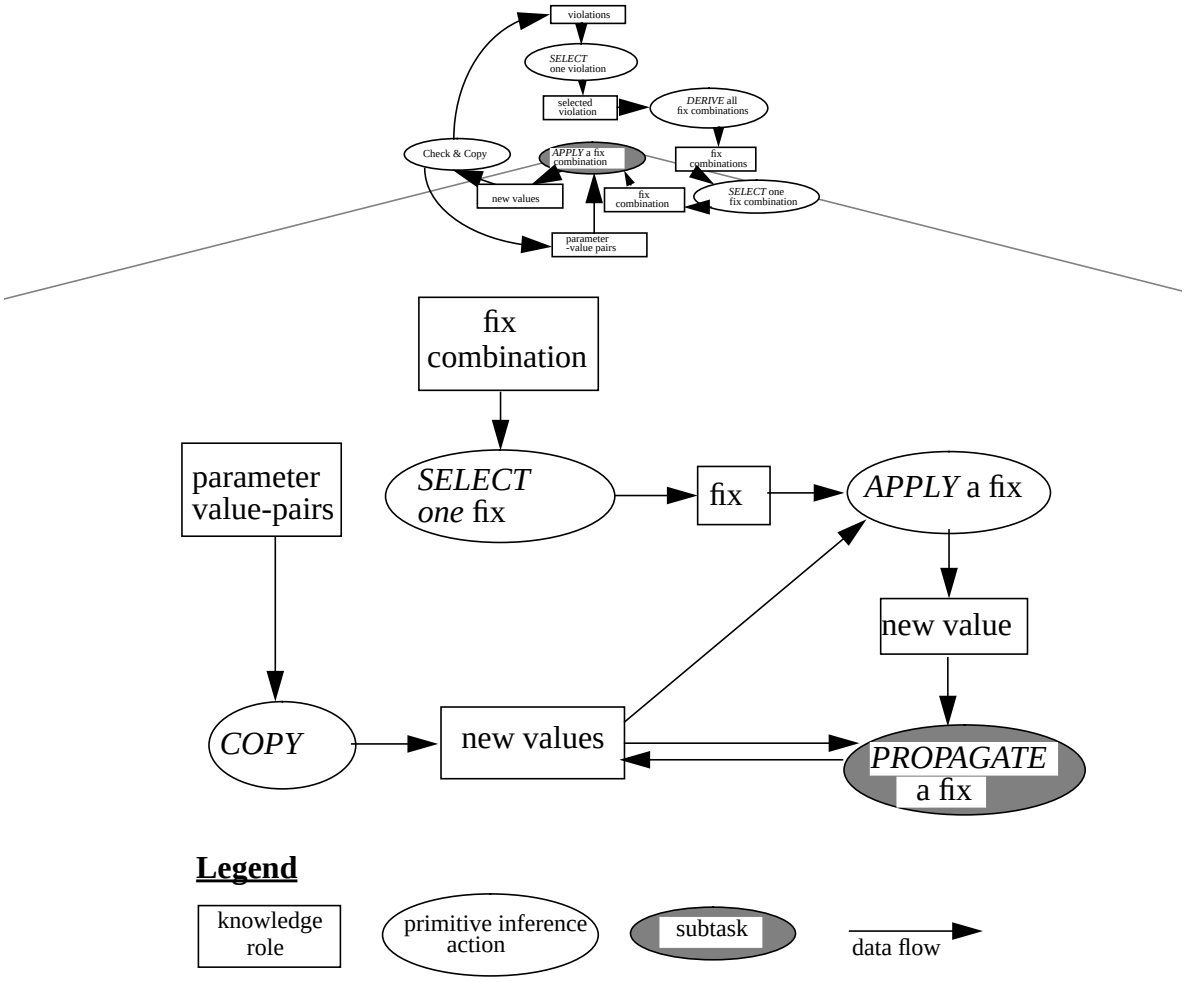

Fig. 10 Data flow of the *Apply-a-fix-combination* subtask.

knowledge acquisition step. But if we slightly modify the assumption about the available domain-specific selection heuristics, we can show that the methods not only differ in their efficiency but also in their effect. In Figure 9 we show a cost model which would use the different cost levels not as ordinar numbers but as weight factors. The alternative control flow would enable the *revise* subtask to deal with cases with infinitely many fix combinations and possible fix combinations with infinitely many fixes. The alternative method is able to do so as it does not try to derive all fix combinations but it only derives one fix combination at a time until a correct one is found. Suddenly, the difference between the two control flows changes from a question of efficiency to a question of effect in general. Naturally, questions arise like:

- under which assumption about the available domain knowledge are two methods equal in effect (or efficiency);

- does domain knowledge fulfil specific assumptions or not?

5.4 Assumptions of Apply-a-Fix-Combination Knowledge

This subtask consists of four primitive inferences. First, we must create an internal copy of the partial assignment which is done by *copy*. Then we select one fix, apply the fix to a parameter value, and, finally, propagate the consequences of the fix application. The data flow diagram of this subtask is given in Figure 10. A fix can look like:

> *Upgrade the machine groove model* or *increase the car supplement weight in 100 pound steps*.

The control flow of this subtask is defined in Figure 11. Again, we will see how specific assumptions are related to the individual steps and their control.

The selection of a fix is done non-deterministically. Again, we make no assumption about available domain knowledge which could guide the selection process, but assume that the sequence in which we choose fixes influences neither the effect nor efficiency of the method. In fact, we assume that the sequence in which fixes are applied does not matter. This includes the assumption that fixes neither directly nor indirectly interact. Direct interaction of two fixes would mean that they refer to the same parameter. In that case, the application of the latter would probably destroy or at least counteract the effect of the former. Indirect

```
Copy;
while ¬empty-fix-combination? do
    Select one fix;
    Apply a fix;
    Propagate a fix;
od
```

Fig. 11 Control flow of the *Apply-a-fix-combination* subtask.

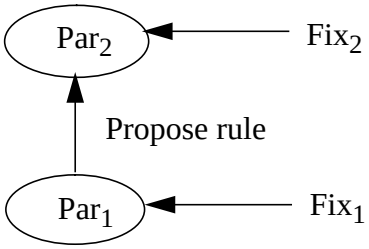

Fig. 12 An example for indirectly interacting fixes.

interaction could take place in the *propagate-a-fix* step. In this step, a modification by a fix is forward-propagated through the network formed by the propose rules. If we modify a parameter value we modify all parameter values for which it is used as a premise of a propose rule. Therefore, fixes can easily interact in this propagate step. In fact, they do so in the VT-domain. This implicit assumption of the psm is therefore violated and the "non-deterministic" order in which we choose fixes influences the outcome of the method. Figure 12 gives an illustration for indirect fix interaction and how this influences—together with the order of their application—the outcome. If fix_1 is applied and propagated first, then its effect is deleted when fix_2 is applied. If fix_2 is applied first, its effect is deleted when fix_1 is applied and propagated. This indirect interaction of fixes can also lead to the violation of constraints which were used to define range restrictions for parameter values to prevent the infinite application of decrement and increment fixes. In the *Derive-all-fix-combination* step, we include an increment-fix for a parameter p at most n times if the current value of p is m and $m+n$ is the upper bound as defined by the according constraint. If a second fix directly or indirectly influences the same parameter p we can obtain a violation of the range restriction even if it was assumed to be legal.

The assumption of the *Apply-a-fix-combination* step that fixes do not interact is very hard to prove. The violation of the assumption can be shown by an example where two fixes interact.

Based on the no-interaction-assumption, an alternative control flow can be defined as shown in Figure 13, but it should be clear that the two control flows differ significantly in their effect

if this assumption is violated.

The *Propagate-a-fix* inference is again a composed inference which is refined further by elementary inferences, but due to limited space we will not deal with this refinement. We want to point out just one property of this step. The changed value obtained by applying a fix is only forward-propagated through the network of propose rules. That is, if a propose rule for other parameters exists which uses the parameter as a premise (i.e., input value) then the propose rule is applied again with the changed parameter value. In fact, this is only done for parameters which already have a value. Parameters which will processed in a further iteration of the whole psm are not regarded. This strategy of forward-propagation (i.e., *without backward-propagation*) of changes makes a strong assumptions about the nature of the knowledge which is encoded by the propose rules. The propose relationship between parameters has to be a one-way functional relation where one parameter functionally depends on another but not vice versa. When we have two parameters p_1, p_2 so that:

$p_2 := 2 * p_1 * \pi$ (i.e., p_1 is the radius and p_2 the circumference of a circle)

we cannot apply a fix to p_2. Parameters to which fixes are applied are implicitly assumed not to be in a two-way functional dependency to the parameters which are used to calculate their initial values by means of propose rules.

```
Copy;
while ¬empty-fix-combination? do
    Select one fix;
    Apply a fix;
od
refill fix-combinations;
while ¬empty-fix-combination? do
    Select one fix;
    Propagate a fix;
od
```

Fig. 13 Alternative control flow of the *Apply-a-fix-combination* subtask.

- One could argue that it would be possible to capture a two-way functional relationship between two parameters using two propose rules. But this would lead to a cycle in the graph of propose rules and would therefore violate an assumption about the propose knowledge.

- One could also argue that we can model one functional dependency using a propose rule and the other using a constraint. But applying the fix to one parameter would then lead to a new constraint violation and the fix application would be redone by the problem-solving process.

5.5 General Assumptions and Limitations of the Entire Revise Step

It still remains for us to look at two further assumptions or limitations of the entire *revise* subtask. First, it treats user input and propose rules as defaults and, second, it processes a very restricted search strategy.

Propose Rules and User Inputs are Defaults

Fixes can change user input values as well as values which were derived by applying propose rules. The assumption is that user inputs and proposed values are only defaults. The psm uses all initial values given by the user only as a starting point for the search process. As it tries to apply the cheapest fixes first, it tries to remain as close to them as possible. But there is no explicit way for the user to indicate which rewrites he accepts and which he does not want. The same holds for the values derived by propose rules. They are just treated as defaults which can be modified by fix application.

Only One-step Look-ahead Search

The application of a fix combination is withdrawn if it leads to the violation of new constraints. Therefore, it is not possible to apply a sequence of fix combination which would bypass a possible dead end of the search process. A sequence like

constraint c_1 is violated; F_1 repairs c_1 but violates c_2, F_2 repairs c_2.

is not possible with our limited search approach. In addition to the fact that it is not possible to backtrack over the selection of violated constraints it also heavily restricts the search space.

6 Outlook

In the paper, we analyzed a formal specified knowledge level model of a variant of the psm *propose-and-revise* to determine its underlying assumptions and limitations. We believe that such an activity is a necessity for *reusing* psm´s. Only if their assumptions are made explicit can we prove whether a psm can be used to solve a given task in a given application domain. That is, a method can only be applied to a task and domain if its assumptions hold for the given application. Conversely, these assumptions can be used to guide the process of *selecting* and *combining* psm`s out of a library. They can be used as features which describe them. Additionally, these assumptions can be used to derive the variants of a psm which best fit a given application (i.e., to *modify* or *differentiate* them according to domain and task circumstances). To examine the reuse of psm`s and the degree of differentiation we will examine our solutions (Angele, Fensel, Landes & Studer, 1992; Angele, Fensel and Landes, 1992) to the Sisyphus-I task (the so-called office assignment problem), our solution to the ECAI-workshop task (Landes, Fensel & Angele, 1994), and our specification of the Board-game method (see Fensel et al., 1993).

The described work defines a strong link to the validation and verification effort of knowledge-based systems. Two different types of tasks generally arise in this context. First, we have to examine the formal specifications of psm`s for their assumptions. In addition, we can vary them, their control flow, for example, and study how this influences their assumptions and limitations. As part of this effort, a language for formally defining such properties has to be developed. A promising step is achieved by Aben (1995). He describes a

library of formally specified standard inference actions. Each inference is described by a body, a preconditions and a postconditions. The body is used to specify the logical inference, the preconditions define assumptions on the input, and the postconditions capture significant features of the result, which is derived by the inference. He also discuss the combination of inference actions but his concept of *compatibility* abstracts from all dynamics (i.e., from any control flow between the inferences). Therefore, he can only formulate assumptions which must hold for all possible control flows.

In addition to the conceptually oriented effort, there is a need to develop formal and automatic techniques for supporting the derivation and checking of assumptions. Support is required for showing that different variants of a psm are equal or unequal in their effect and efficiency under some assumptions. We have to determine, whether a given domain knowledge fulfils the assumptions of a psm. More precisely, we have to find conditions which ensure these assumptions. For example, they need not hold true for every possible (and meaningless) user input but only for the relevant input. So the task becomes a search for meaningful *preconditions* which, together with the given domain knowledge, entail *postconditions* which cover these assumptions.

Still an open problem is whether a list of assumptions or limitations is complete for a given psm. The list we mentioned in the paper is by no means complete. For example, assumptions and limitations which we did not mention are: The number of parameters has to be constant. Each property of the desired solution is expressible as constraints over some parameters and as user input values. The user is treated as a source of input data but not as a helpful agent for problem-solving. An alternative approach would be to view him as an active agent who solves the problem in interaction with and assisted by the system. The problem-solving process is modelled by viewing the expert system “as an autistic problem solver” (de Greef and Breuker, 1992) It is clear that these are different types of assumptions and it appears necessary to derive a typology of such assumptions/limitations in order to take an initial step

toward dealing with the question of completeness.

Acknowledgement

I would like to thank Frances Brazer, Frank van Harmelen, Dieter Landes, Remco Straatman for helpful comments and especially Karsten Poeck for delivering a rich input for the case study. Jeffrey Butler worked hard on improving the English of the paper.

References

Aben, M. (1995). Formal Methods in Knowledge Engineering, Ph D dissertation, University of Amsterdam.

Angele, J., Fensel, D. Landes, D. (1992). An Executable Model at the Knowledge Level for the Office-Assignment Task. Linster, M., Ed. Sisyphus ´92: Models of Problem Solving, Arbeitspapiere der GMD, no 663.

Angele, J., Fensel, D. Landes, D. & Studer, R. (1992). An Assignment Problem in Sisyphus - No Problem with KARL. Linster, M., Ed. Sisyphus ´91: Models of Problem Solving, Arbeitspapiere der GMD, no 630.

Akkermans, H., Wielinga, B. & Schreiber, G. (1993). Steps in Constructing Problem-Solving Methods. Aussenac, N. et al., Eds., Knowledge Acquisition for Knowledge-based Systems. Proceedings of the 7th European Workshop (EKAW´93), Toulouse and Caylus, France, September 6-10, Lecture Notes in Artificial Intelligence (LNAI), no 723, Springer-Verlag, Berlin.

Benjamins, R. (1993). Problem Solving Methods for Diagnosis, PhD dissertation, University of Amsterdam.

Breuker, J. & Van de Velde, W., Eds. (1994). CommonKADS Library for Expertise

Modelling, IOS Press, Ohmsha.

Breuker, J. A., Wielinga, B., van Someren, M., de Hoog, R., Schreiber, G., de Greef, P., Bredeweg, B., Wielemaker, J. & Billault, J.-P. (1987). Model-Driven Knowledge Acquisition: Interpretation Models. In ESPRIT project P1098, report, University of Amsterdam.

Chandrasekaran, B. & Johnson, T. R. (1993). Generic Tasks and Task Structures: History, Critique and New Directions. David, J.-M. , Krivine, J.-P. & Simmons, R., Eds. Second Generation Expert Systems, Springer-Verlag, Berlin..

Clancey, W. J. (1985). Heuristic Classification. Artificial Intelligence, vol 27, 1985.

David, J.-M., Krivine, J.-P. & Simmons, R., Eds. (1993). Second Generation Expert Systems, Springer-Verlag, Berlin.

Fensel, D. (1995). The Knowledge Acquisition and Representation Language KARL, Kluwer Academic Publ., Boston, to appear.

Fensel, D., Eriksson, H., Musen, M. A. & Studer, R. (1993). Description and Formalization of Problem-Solving Methods for Reusability: A Case Study. Complement Proceedings of the 7th European Knowledge Acquisition Workshop (EKAW´93), Toulouse, France, September 6-10.

Fensel, D. & van Harmelen, F. (1994). A Comparison of Languages which Operationalize and Formalize KADS Models of Expertise. The Knowledge Engineering Review, vol 9, no 2.

de Greef, P. & Breuker, J. A. (1992). Analysing System-User Cooperation in KADS, Knowledge Acquisition, vol 4, no 1.

Harel, D. (1984). Dynamic Logic. Gabby, D. et al., Eds., Handook of Philosophical Logic, vol. II, Extensions of Classical Logic, Publishing Company, Dordrecht (NL).

Kozen, D. (1990). Logics of Programs. Van Leeuwen, J., Ed., Handbook of Theoretical Computer Science, Elsevier Science Publ., B. V., Amsterdam.

Krueger, C. W. (1992). Software Reuse. ACM Computing Surveys, vol 24, no 2.

Landes, D. (1995). Die Entwurfsphase in MIKE. Methode und Beschreibungssprache, Ph D dissertation, University of Karlsruhe.

Landes, D. Fensel, D. & Angele, J. (1993). Formalizing and Operationalizing a Design Task with KARL. Treur, J. & Wetter, T., Eds. Formal Specification of Complex Reasoning Systems, Ellis Horwood, New York.

Linster, M., Ed. (1994). Sisyphus ´91/92: Models of Problem Solving. International Journal of Human Computer Studies, vol 40, no 3.

Marcus, S., Ed., (1988). Automating Knowledge Acquisition for Experts Systems, Kluwer, Boston.

Marcus, S., Stout, J. & McDermott, J. (1988). VT: An Expert Elevator Designer That Uses Knowledge-based Backtracking. AI Magazine, vol 9, no 1.

Marques, D., Dallamagne, Klinker, G., McDermott, J. & Tung, D. (1992). Easy Programming. Emporing People to Build Their Own Applications. IEEE Expert, vol 7, no 3.

Musen, M. A. (1989). Automated Generation of Model-Based Knowledge-Acquisition Tools, Morgan Kaufmann Publisher, San Mateo, CA.

Musen, M. A. (1992). Overcoming the Limitations of Role-Limiting Methods. Knowledge Acquisition, vol 4, no 2.

Poeck, K. & Gappa, U. (1993). Making Role-Limiting Shells More Flexible. N. Aussenac et al., Eds., Knowledge Acquisition for Knowledge-Based Systems, Proceedings of the 7th European Workshop (EKAW´93, Toulouse, France, September 6-10, 1993), Lecture Notes in AI no 723, Springer-Verlag, Berlin.

Poeck, K., Fensel, D., Landes, D. & Angele, J. (1994). Combining KARL and Configurable Role Limiting Methods for Configuring Elevator Systems. Proceedings of the 8th Banff Knowledge Acquisition for Knowledge-Based System Workshop (KAW´94), Banff, Canada, Januar 30th - February 4th.

Poeck, K. & Puppe, F. (1992). COKE: Efficient Solving of Complex Assignment Problems With the Propose-And-Exchange Method. Proceedings of the 5th International Conference on Tools with Artificial Intelligence, Arlington, Virginia, November 10-13.

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

Schreiber, G. & Birmingham, B., Eds. (1994). Proceedings of the 8th Banff Knowledge Acquisition for Knowledge-based Systems Workshop (KAW´94), vol III, Sisyphus II - VT Elevator Design Problem, Banff, Canada, January 30 - February 4.

Schreiber, G., Wielinga, B., Allermans, H., Van de Velde, W., de Hoog, R. (1994). CommonKADS. A Comprehensive Methodology for KBS Development. IEEE Expert, vol 9, no 6.

Steels, L. (1990). Components of Expertise. AI Magazine, vol 11, no 2.

Treur, J. & Wetter, Th., Eds. (1993). Formal Specification of Complex Reasoning Systems, Ellis Horwood, New York.

Wielinga, B., Akkermans, J. M. & Schreiber, A. Th. (1995). A Formal Analysis of Parametric esign Problem Solving. Proceedings of the 9th Banff Knowledge Acquisition for Knowledge-based Systems Workshop (KAW´94), Banff, Canada, February 26 - March 3.

Yost, G. R. (1992). Configuring Elevator Systems, Technical report, Digital Equipment Co., Marlboro, Massachusetts.

Zdrahal, Z. & Motta, E. (1995). An In-Depth Analysis of Propose & Revise Problem Solving Methods. Proceedings of the 9th Banff Knowledge Acquisition for Knowledge-based Systems Workshop (KAW´94), Banff, Canada, February 26 - March 3.