Construction of problem-solving methods as parametric design

Annette ten Teije [1], Frank van Harmelen [2], Guus Schreiber [1] and Bob Wielinga [1]

[1] SWI, University of Amsterdam, {annette}@swi.psy.uva.nl
[2] Dept. of Math and CS, Vrije Universiteit Amsterdam

Abstract

The knowledge-engineering literature contains a number of approaches for constructing or selecting problem solvers. Some of these approaches are based on indexing and selecting a problem solver from a library, others are based on a knowledge acquisition process, yet others are based on search-strategies. None of these approaches sees constructing a problem solver as a configuration task that could be solved with an appropriate configuration method. We introduce a representation of problem solving methods that allows us to view the construction of problem solvers as a configuration problem, and specifically as a parametric design problem. There are several methods for solving configuration tasks. Studying these methods and in particular the method of propose-critique-modify results in guidelines for arranging the automated configuration theory. Furthermore we illustrate this method by a scenario in a small car domain example. This scenario is detailed enough that it can be directly implemented in a suitable architecture, which we have described elsewhere.

1 Introduction

The literature on Knowledge Engineering has identified a number of different *problem types* (Hayes-Roth *et al.*, 1983; Clancey, 1985) (e.g. diagnosis, design, monitoring) and identified for each problem type a number of *problem solving methods* (PSMs), which are methods that can be employed to solve a problem of that particular type. For example, diagnosis problems can be solved by such diverse methods as consistency-based diagnosis, hierarchical diagnosis or abduction (see (Console *et al.*, 1992) for a survey).

A central question is then *"Which problem solving method (PSM) is optimal for a given problem type?"*. In general, the choice of an appropriate PSM will depend on the goal of problem solving, and on characteristics of the specific input (knowledge and data). As a result, PSMs must be selected or be constucted. In the former case, methods are selected from a predefined set, while in the latter case parts of existing methods or newly defined parts are combined to construct a new method. Such a selected or constructed method does not guarantee the satisfaction of all the intended goals, for example due to lack of sufficient knowledge about when to apply a PSM, or due to incompleteness of data or knowledge inherent to AI-problems. Because the intended goals are not guaranteed, we have to validate the constructed method. If this validation fails, we have to iterate the selection and construction process, using the results of the validation.

This paper proposes a novel solution for the automated construction of methods. The approach is based on the correspondence between the construction of methods and parametric design. A restriction of our proposal is that we consider a PSM as a logic program and study only the declarative properties of PSMs, and no efficiency or other algorithmic properties. Furthermore, our study of automated construction of PSMs is based on studying diagnostic methods, although we belief that it will apply in general to other classes of PSMs.

The structure of this paper is as follows. First we give a definition the problem of the automated construction of PSMs. Then we describe the generic configuration task based on existing literature. Subsequently, we interpret automated construction of PSMs as a configuration task and we discuss methods for this configuration task. Finally the body of this paper discusses a particular method for automated configuration of PSMs. This particular configuration method is illustrated through a detailed scenario in which we configure a diagnostic PSM. This scenario is detailed enough that it can be directly implemented in a suitable architecture, which we have described in (ten Teije & van Harmelen, 1996b).

2 Analysis of the Construction Problem

In general the inputs of automated construction are:

1. the input problem for which we need to construct a method (given as: data and knowledge, e.g. a particular case to diagnose);
2. the assumptions under which the method will have to operate;
3. the goals that the resulting method will have satisfy.

The outputs are:

1. the description of the constructed method;
2. the solutions computed by the method
3. the possibly slightly adjusted versions of the input problem, the goals and the assumptions.

The input/output relation of the construction process is as follows:

- the output has to be a representation of a method;
- it must not conflict with the (possibly adapted) assumptions;
- it must satisfy the (possibly adapted) goals
- the slightly adapted inputs (assumptions, goals, problem) have to be closely related to the original ones.

Examples of the inputs in the context of diagnosis are (1) the diagnostic problem containing the observed behaviour and the behaviour model, (2) the single fault assumption, and (3) a goal such as a maximal size of the diagnosis.

The goal of automated construction of methods is to construct a method that produces acceptable solutions for a given problem under particular assumptions and desired goals. Our approach is to first configure and then validate a method, and, if this validation fails, to iterate the configuration step. We call the construction before validation *static configuration* and the configuration using the validation results *dynamic configuration*. The question in static configuration is "Which PSM is optimal?" and in dynamic configuration "What should be done if the PSM does not give the desired solution?". In line with the distinction of static and dynamic configuration we distinguish static and dynamic goals. *Static goals* are requirements (of the solution or of the method) that can be guaranteed solely on the basis of the description of the method. For example, the goal that a method always produces singleton diagnoses. *Dynamic goals* are requirements of the solution that can only be validated after executing the method. For example, the goal of a maximal number of diagnoses. This distinction between static and dynamic goals is not fixed. With more knowledge a dynamic goal might be established statically. It depends on the knowledge that is available about methods, whether goals are static or dynamic.

The method description that we have to construct has to satisfy both types of goals. The construction process proceeds in two steps. The first step of the construction process concerns the configuration of a method that satisfies the static goals. If there is no such a method, the second step occurs: we adapt the problem, assumptions or goals slightly such that a method can be constructed that satisfies the static goals (possibly slightly adjusted). If this method also satisfies the dynamic goals, a suitable method has been constructed, otherwise we try to adapt the method in such a way that is does. However, when this is impossible we again adapt the problem, assumptions or goals slightly and configure a method for these new inputs. The basic idea is that we construct the method that computes the "best" possible solutions for the given problem and assumptions and desired goals. For computing these solutions, the constructed method possibly has to apply to a problem which is a slight modification of the original problem, and under possibly slightly modified assumptions and for possibly slightly modified goals.

In all this, the object of the construction is the method description. The possibly slightly adjusted assumptions, goals and problem are side effects of configuring an appropriate method for a given problem under particular circumstances.

3 The Representation of Methods

Our approach to automated configuration of problem solvers relies on exploiting the theory about problem solving methods from (ten Teije & van Harmelen, 1994) and (ten Teije & van Harmelen, 1996b). In that work we have proposed a uniform representation of (the functionality of) problem solving methods. The central idea of this representation is that the functionality of a class of problem solving methods is captured in a single schematic formula. Some of the predicates and terms from that formula are regarded as parameters that must be further instantiated to capture different members of the class of problem solving methods. Thus, given a schematic formula that defines the functionality of a whole class of problem solving methods, different members of that class correspond to different definitions for the parameters occuring in the schematic formula.

It is exactly this uniform representation of an entire class of problem solving methods that will allow us in this paper to view the construction process of problem solving methods as a parametric design task. Since we will illustrate our theory about the configuration of problem solving methods with examples from diagnostic problem solving methods, we will now give our schematic definition of these diagnostic methods.

In general, a diagnostic problem arises if there is a discrepancy between the observed behaviour of a system (e.g. an artifact) and how the system should behave, in other words, the expected behaviour does not correspond with reality. The diagnostic task is to find out the cause of this discrepancy. A diagnostic method computes the solutions for a diagnostic problem by using a model of the expected behaviour (the behaviour model, *BM*), the actually observed behaviour *OBS*, and contextual information *CXT*. The computed solutions of a diagnostic problem represent an explanation for the observed behaviour.

Our uniform representation of diagnostic problem solvers is based on the following general account of their functionality: An explanation distinguishes *two types of observations*: it covers some observations, and it does *not contradict* other observations. The explanation is restricted to a *vocabulary* of special candidates that could be causes of a behaviour discrepancy (e.g. components). Usually we are not interested in all possible explanations, but only the *most reasonable* explanations. We also want to *represent* an explanation as a solution that a user can interpret. (For example, in medical domains, users are usually interested in the disease, and not in all the current states of the parts of the patient's body).

Together, these six aspects written in italics make up the particular notion of diagnosis that is realised in a given method. We can capure these general characteristics of a diagnostic method in the following formal definition:

When given as input the behaviour model *BM*, a context *CXT* and a set of observations *OBS*, a diagnostic method computes a set of solutions *Sol* such that:

$$\begin{array}{lll}
\underline{\textit{Obs-mapping}}(OBS) = \langle Obs_{cov}, Obs_{con}\rangle & & \textit{and} \\
\overline{Es = \{E|} \quad BM \cup E \cup CXT \underline{\vdash_{cov}} Obs_{cov} & \textit{and} & \\
\qquad\qquad BM \cup E \cup CXT \underline{\not\vdash_{cov}} \bot & \textit{and} & \\
\qquad\qquad BM \cup E \cup CXT \underline{\not\vdash_{con}} \neg Obs_{con} & \textit{and} & \\
\qquad\qquad E \subseteq \underline{Vocabulary}\} & & \textit{and} \\
\underline{Selection}(Es, E') & & \textit{and} \\
\underline{Solution\text{-}form}(E', Sol) & &
\end{array} \tag{1}$$

Each of the six underlined terms is one of the parameters in our representation of diagnostic methods. Varying one or more parameters amounts to describing a different diagnostic methods. The *Obs-mapping* determines which observations must be explained (or: covered) Obs_{cov}, and which need only not be contradicted (Obs_{con}). E is an explanation for the observed behavoiur by covering some observations ($\vdash_{cov}$), and not contradicting others ($\not\vdash_{con}$). We write $\vdash_{cov}$ and $\not\vdash_{con}$ as different symbols to emphasise that one is not necessarily the negation of the other, and that neither is necessarily the same as the classical entailment $\vdash$. E is expressed in a particular *Vocabulary*. We are interested in the most reasonable explanations, determined by a *Selection* criterion. The *Solution-form* determines the representation of the final result of the method. The dependencies between all these components of a diagnostic method is shown in figure 1

In (ten Teije & van Harmelen, 1994), we show that we can formulate properties of this general schematic formula, as well as properties of instances of the schema. Such properties will be exploited in the configuration of methods. In (ten Teije & van Harmelen, 1996b) we have argued that this representation can in principle be applied to other families of methods than diagnostic methods, such as methods for monitoring, design, classification etc. As a result, we will claim that also our approach to the configurtion of methods is general,

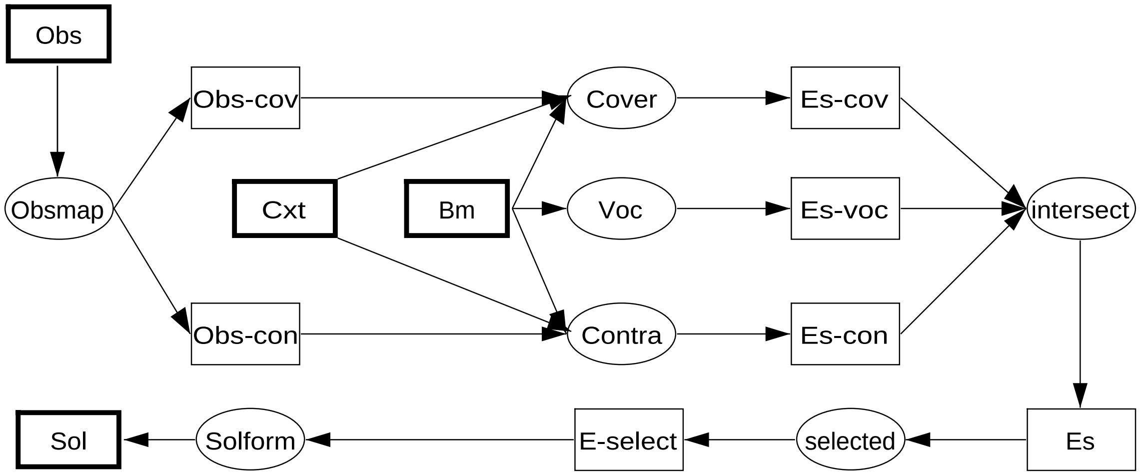

FIGURE 1: Components of diagnostic methods and their relations. Ovals are components, boxes are their inputs/outputs, thick boxes are inputs/outputs of the entire method

and could be applied to such other families of problem solving methods.

4 Configuration Task

In the literature on configuration there is a consensus about the nature of configuration tasks. Most definitions of a configuration task found in the literature are a slight variant of (Mittal & Frayman, 1989):

> "Given: (A) a fixed, pre-defined set of components, where a component is described by a set of properties, ports for connecting it to other components, constraints at each port that describe the components that can be connected at that port, and other structural constraints; (B) some description of the desired configuration; and (C) possibly some criteria for making optimal selections.
>
> Build: One or more configurations that satisfy all the requirements, where a configuration is a set of components and a description of the connection between the components in the set, or detect inconsistencies in the requirements."

The configuration task can be considered as a search problem using the above types of inputs and output (Löckenhoff & Messer, 1994) The configuration process restricts this search space in four steps using the various types of inputs (see Figure 2). The set of possible components and the possible connections between these components are fixed and given beforehand. This restricts the search space to the *possible configuration space*. The constraints restrict this possible configuration space to the *valid configuration space*. The user-requirements restrict this valid configuration space to the *suitable configuration space*. The optimality criteria can possibly restrict or divide this space further.

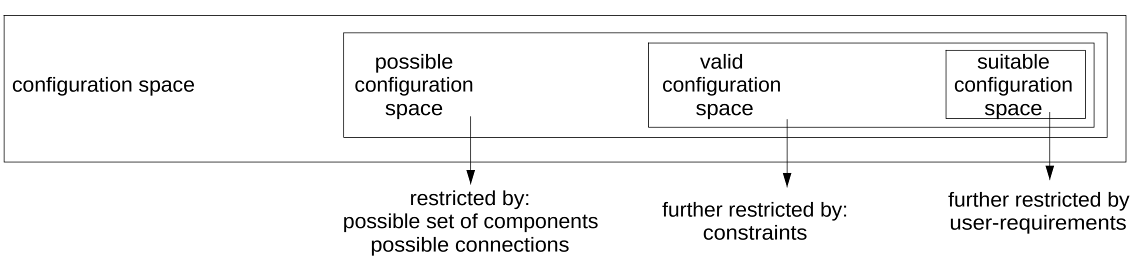

FIGURE 2: Configuration task as search problem

Parametric design is a simplification of the configuration task (class 3 problem). In parametric design are only fixed structures and fixed components. A components is a parameter which have a particular range that is given before hand. This reduces the configuration problem, because we only have to assign values to a parameter in its own range. (See (Wielinga *et al.*, 1995) for a detailed analysis of parametric design).

4.1 Automated Configuration of Problem Solvers as Parametric Design

In this section we map the automated configuration of PSMs on the configuration task. In order to make this mapping, we consider the general characteristics of the configuration task given above in the context of the construction of problem solvers and we consider the configuration of PSMs as a search problem.

We first consider the input types of the configuration task in the context of configuring PSMs. The inputs are:

- *components:* The set of possible components are the possible definitions of the components in the schematic formula from Section 3 (formula (1)). (e.g. subset minimality for the *Selection* component). These possible definitions are the building blocks of the configuration and are fixed and given beforehand.
- *compositional structures* (the connections): The representation of a method is the schema from formula (1). This schema is the only allowed structure, and is indeed fixed and given beforehand The mapping to a configuration problem is possible, exactly because we have a schema for representing diagnostic methods in a uniform way (ten Teije & van Harmelen, 1994).
- *constraints:* the constraints between the diagnostic components and constraints between underlying assumptions of the components.
- *user-requirements:* the goals (static or dynamic) that have to be fulfilled.
- *optional:* optimality criteria, transformation knowledge and heuristic knowledge for search. Although we appreciate the need for these types of knowledge, they are outside the scope of our current work.

The output of the configuration of methods consists of the six components of particular types which are structured in such a way that together they represent a diagnostic method.

The three types of configurations (possible, valid, and suitable, Figure 2) can be given a meaning in configuring methods. A possible configuration is a method that contains a definition for each component of the general method schema. A valid configuration is a method that expresses a diagnostic method and has no conflicts with the assumptions under which the method must operate. A suitable configuration is a method that satisfies the desired goals.

The mapping from the elements of a general construction problem onto our problem of method construction shows that we can indeed interpret automated configuration of diagnostic problem solvers as a configuration problem. In fact, it can even be interpreted as parametric design, because we use a fixed structure and the possible definitions of each component can be considered as the range of the parameters in formula (1). However, in our view of configuring PSMs we do not only modify the method, but possibly also the assumptions, goals, and the input problem, as already stated in Section 2.

4.2 Methods for the Configuration Task

In this section we discuss configuration methods from the literature: *generate-&-test*, *propose-critique-modify* (PCM) and a specific PCM method *propose-&-revise*. We evaluate all these possible methods for configuring problem solvers. Although our study is not exhaustive, in this section we argue that the PCM-paradigm is an appropriate paradigm for configuring PSMs.

Generate-&-test family This family of methods generates a configuration in the first step, and subsequently tesets this configuration. There is a wide range of generation and test steps, from a simple generation step with a knowledge-intensive test step to a knowledge-intensive generation step with a simple test step. Knowledge that can be used concerns the set of possible components, the possible connections between components, the constraints and the user-requirements. Characteristic of a generate-&-test method is that when a configuration does not pass the test, the configuration process continues with a completely new configuration, without taking into account the reason why the previous configuration failed the test. In our case the generate-&test method is not appropriate because our test for the dynamic goals is expensive, since it requires performing diagnosis.

Propose-critique-modify family Characteristic of a propose-critique-modify (PCM) method is that when a configuration is not a suitable configuration, the configuration process does not continue with a complete new configuration, but uses the test results for determining a new configuration instead of generating a new one from scratch. The propose-critique-modify (PCM) family (Chandrasekaran, 1990; Brown & Chandrasekaran, 1989) consists of four steps: propose, verify, critique and modify. We discuss each step in turn.

Propose: The propose step gives a partial or a complete configuration. Methods for the propose step are: solution decomposition, design proposal by case retrieval, and constraint satisfaction (Chandrasekaran, 1990). For our specific case of configuring diagnostic methods, another method (close to the one used in the VT-task (Schreiber & Birmingham, 1996)) seems more appropriate. In this propose-method, parts of the design (in our case some of the parameters in the diagnostic schema) are proposed on the basis of requirements. These partial proposals are then completed into full proposals by proposing values for the remaining parameters. (As will be explained later on, in our case this completion process is unguided in our current proposal).

Verify: The verify step involves checking that the proposed configuration satisfies the constraints and the user-requirements. (Chandrasekaran, 1990) distinguishes two verification steps. (1) "attributes of interest" that can be directly calculated or estimated by means of domain specific formulae. In our case (configuring diagnostic problem solvers) these are the constraints on the diagnostic components and on the assumptions. (2) "behaviour interest" that can be derived by simulation. In our case the simulation amounts to performing diagnosis. Based on these results the dynamic goals have to be verified. We use the term simulation-verification of (Chandrasekaran, 1990), but validation should be a more appropriate name, because we validate the method by execution.

Critique: The critique step is a diagnostic problem of mapping from undesired behaviour to the parts of the configuration which are possibly responsible for this undesired behaviour[1]. This step analyses the failure of the configuration. Therefore it needs information about how the structure of the device contributes to the desired behaviour. In our case this is knowledge of how properties of the components of the diagnostic schema relate to properties of the complete schema. In this phase one can use (meta-)diagnostic knowledge about goal violations and repairs.

Modify: The modify step uses the repair information from the critique step and executes the repair action. It changes the configuration to get closer to the specifications. In our case this is the actual adaptation of the diagnostic method.

Propose-&-revise family The propose-&-revise family is a sub-family of PCM methods. These methods are used in the VT-domain (Schreiber & Birmingham, 1996). This family of methods is a simplification of the PCM method, because the critique step is replaced by compiled knowledge. The idea behind this family of methods is that it is possible to give an initial proposal for a configuration. This configuration is constructed by selecting values for the set of components based on the user-requirements. This configuration can be "fixed" (repaired) if constraints are violated. These fixes are the compiled critique knowledge. Fixes are direct associations of a constraint violation and a repair action by changing one or more parameter values (Runkel *et al.*, 1995; Marcus *et al.*, 1988; Fensel, 1995). Propose-&-revise methods require these fixes as search control knowledge.

A propose-&-revise method is not appropriate for automated configuring of PSMs for two reasons. First in our problem we need a full critique step. The critique step is quite complex and it is not possible to code it in simple direct associations between a constraint violation and a repair action. Secondly, propose-&-revise methods are used because of the large search spaces, but our most important motivation is to prevent the expensive tests of dynamic goals (performing diagnoses). Our efficiency problem is not in the constraints but in verifying the dynamic goals.

Conclusion From this very brief discussion of the configuration of PSMs seen as a configuration task, we conclude that the family of propose-critique-modify is the most suitable one for a method for automated configuration of PSMs. The specific propose-&-revise method is not appropriate for our application, because (1) we need an explicit critique step and (2) the efficiency problem differs from constraints in propose-&-revise versus dynamic goals in our case. Furthermore, we made the PCM method more specific by allowing the possibility to adapt beside the diagnostic methods, also the assumptions, the diagnostic problem and the goals.

[1] Notice that this is a *meta*-diagnostic problem, since we are diagnosing failures in diagnostic methods.

5 A Propose-critique-modify Method for Configuring PSMs

In this section we describe a method of the PCM family for automated configuration of problem solvers. We configure complete models and verify, criticise and modify them. We discuss the four steps of a PCM method (propose, verify, critique and modify), and visualise them in diagrams: the ovals are inferences (steps and sub-steps in the method), the solid-line boxes are input/output data of the inferences, and the dotted boxes represent knowledge that is specific for a particular type of PSMs. In our case the dotted boxes contain knowledge about diagnostic methods.

5.1 Propose

The propose step proposes a configuration. It has to propose an instance of the general schema that we use for representing PSMs. In our study such a proposed configuration is an instantiation of the six components of the diagnostic schema. We describe a method by a term[2]

ds(Obs-mapping,Vocabulary,Cover,NotContra ,Selection,Solform)

where each argument of *ds* (for: diagnostic system) represents a definition of the particular component (e.g. *Obs-mapping*, ie. one of the underlined terms from formula (1)). Such a definition is a definition taken from the possible set of instances of a component. The proposed components definitions are not structured, but are only a definition from a fixed set that is given beforehand. The *Selection* component is the sole component that can be structured. However in the propose step only "basic" selection criteria are proposed, which can be adapted to more complex ones later in the modify step the *Slection* component. We will illustrate this in the scenario in Section 6.

The propose step (see Figure 3) results in a configuration (i.e. a method description) from the possible configuration space, by selecting a definition for each of the six components. This selection is controlled by the required static goals. An example of a static goal would be that the configured method has to result in a small set of the solutions, which would result in proposing a strong *Selection* component.

If the static goals do not determine a definition for each component (or when there are no static goals), the proposed method is completed with an arbitrarily chosen definition from the set of possible definitions for these components. When different static goals require different definitions of the same component, one of these definitions is chosen arbitrarily, and the goals that are not guaranteed by the method become dynamic goals. Satisfying static goals might depend on the diagnostic problem or on the given assumptions. For this reason, the given input assumption and problem are input for the propose step.

Characteristic of this propose step is that it always gives a proposal, and that the static goals control the search space in this phase of the configuration process. The specific (diagnostic) knowledge that is used in the propose step is (1) the knowledge for fulfilling a static goal, (2) the number of components (the arity of the schema of *ds(...)*) (3) a set of definitions for each component. The propose step enables us to generate possible methods using the definitions for the diagnostic components in the system. However, at this moment we do not say anything about the sequence of choices of diagnostic component and about the sequence of the proposed configurations.

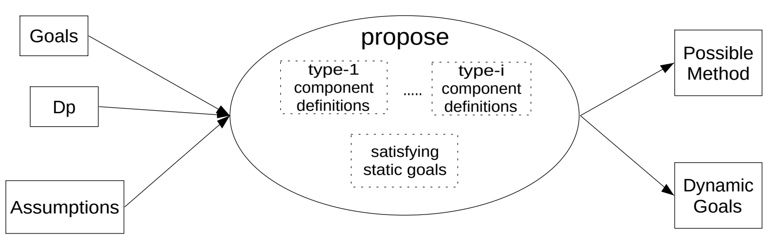

FIGURE 3: Propose Step: The *Possible Method* is a complete definition of a method, where the *Goals* are fulfilled as much as possible. The *Dynamic Goals* are those goals which are part of the *Goals*, but which are not guaranteed by the proposed method.

[2]Terms beginning with a capital letter will denote variables.

5.2 Verify

The verify step checks whether the proposed method satisfies the constraints and the user-requirements (goals). The verify step is divided into two (sub) steps: knowledge-verification and simulation-verification (these names are taken from (Chandrasekaran, 1990)). In the context of problem solving methods we might better call them the static-verification (verifying before execution of the method) and the dynamic-verification (verifying after execution of the method) respectively. We discuss both verification steps in turn.

Knowledge-verification In flexible problem solving the knowledge-verification consists of two types of problem-type specific knowledge (e.g. diagnosis specific) (1) constraints between components and (2) constraints following from assumptions. The knowledge-verification step (see Figure 4) uses the component-constraints and the assumption-constraints for testing whether a method is valid. Both type of constraints might depend on the given assumptions and the input problem. For example, the compatibility of some diagnostic components depends on the kind of behaviour model (which is part of the diagnostic input problem).

An example of an assumption-conflict is the following: Suppose that the assumption is given that the causes in our behaviour model are not necessarily indepent, but are possibly correlated. This would cause an assumption-conflict if we would ever use number-minimality as a *Selection* component. Number-minimality selects the explanation with the lowest number of causes (since a small number of faults is more likely than a high number of faults). This minimality-criterion only makes sense if the the causes are assumed to be uncorrelated. After all, if the causes are correlated, a single unmodelled cause might underly a large number of correlated causes in our explanation, and we would incorrectly rule out such an explanation with our selection criterion.

In the configuration literature the term *valid configuration* is used. A method is valid if it is both component-valid method and assumption-valid. A method is *component-valid* if and only if all the component constraints hold and a method is *assumption-valid* if no assumption conflict occurs.

If verification fails, a new propose step will be performed. However, the distinction between the propose step and the verify step is relative. We can make the propose step gradually more knowledge intensive by including more knowledge of the knowledge-verification step in the propose step. We can only propose component-valid methods, or only assumption-valid methods, or even only valid methods. We can make the propose step less knowledge intensive by generating arbitrary methods, without using the static goals for guiding the proposal of a method. The knowledge about the particular problem type (in our case diagnosis) determines which type of knowledge (static goals, assumption conflicts or component constraints) must be part of the propose or knowledge-verification steps. In our case the knowledge about diagnostic methods enables us to guide the propose step using the static goals. This makes the propose step a kind of nested generate-&-test, which generates proposals which are tested using the static goals. This saves us generating proposals which can be easily determined as inappropriate.

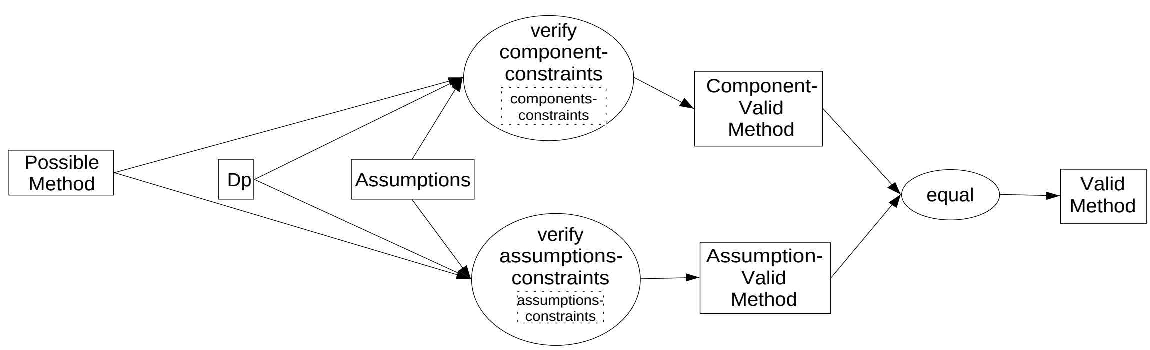

FIGURE 4: Knowledge-verify step: a *Valid Method* is a *Possible Method* which causes no assumption conflicts and no component-constraint conflicts. If verification failed a new propose step will be performed.

Simulation-verification Simulation-verification consists of performing diagnosis followed by tests whether the dynamic goals are met. Diagnosis is performed using the valid method of the knowledge-verification step. The computed diagnoses are used for testing the dynamic goals. (See Figure 5).

The verification of the dynamic goals requires the computed diagnoses. Computing these diagnoses is expensive, and therefore the simulation-verification is expensive. An examples of a dynamic goals is a requirement on the size of the diagnoses. Sometimes these dynamic goals can be guaranteed by a particular choice of a

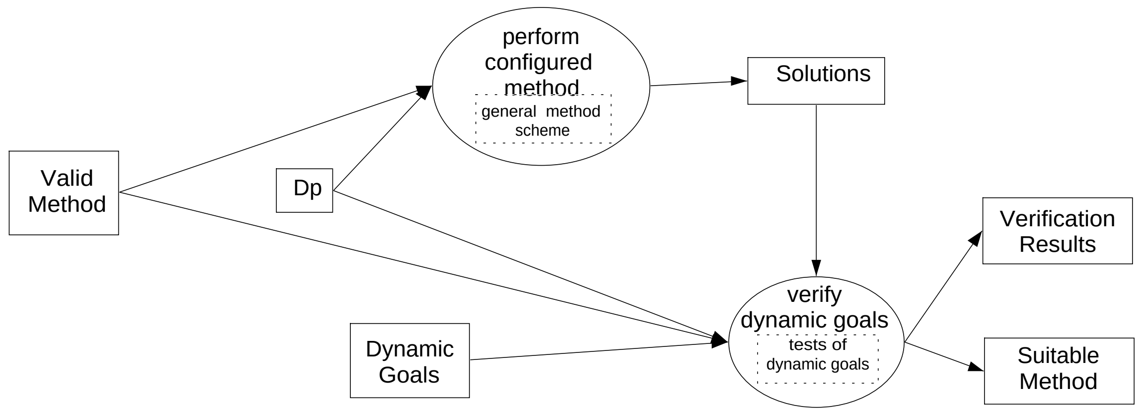

FIGURE 5: Simulation-verify Step: If the *Verification-Results* contains "success" then the method is a *Suitable Method*. The *Verification-Result* is "success" if the *Valid Method* meets all the *Dynamic Goals*, otherwise the result consist of the violated goals, as well as the used method.

component (ie. statically determined), but if this component is not appropriate for other reasons (e.g. an assumption conflict) then we might chose another component. In such a case we have to verify this goal dynamically. In the case that not all goals are met, the results of verification contain the failing goals, as well as the method that failed to meet these goals.

5.3 Critique

The critique step is an analysis of why the verification failed, in other words why the method is not an appropriate method. In our propose-critique-modify method we verify and criticise complete methods. The result of the step is the identification of one of the six components that is held responsible for the failure of the verification step. Notice that we do not yet identify a possible repair action that must be taken to fix this component. That is the purpose of the modify step. The blame-assignment is done based on domain specific knowledge (i.e. diagnosis knowledge). Unfortunately, we do have only limimted concrete examples of such knowledge. Another open issue is what to do if there are multiple possible components that can be responsible for the verification failure.

An example would be a violation of the goal "maximum number of diagnoses is one". The system might contain the knowledge that the existence of too many solution can be blamed on the selection criterion. A possible subsequent repair action in the modify step would then be to use a definition for the *Selection* component that filters more explanations.

We need a critique step, because the verification (especially the simulation-verification) is very expensive (because of performing diagnosis). Such a critique step enables us to control the search instead of generating arbitrary methods and testing these methods until we find a correct one. This is our main motivation for using a propose-critique-modify method. Normally the large search space is the main motivation to use PCM methods. In our case this holds too, but even more important is the motivation of the expensive simulation-verify step. Therefore controlling (reducing) the search space is necessary.

5.4 Modify

The modify step uses the result of the critique step to find an appropriate modification. Given a component that must be modified, finding the appropriate repair action is not immediately obvious. Like every step of our PCM-method the modification uses problem-type specific knowledge, such as the properties of components. For example, the repair-action of strengthening the *Selection* component results in checking for which possible *Selection* components this holds (for example: "number-minimal" is stronger then "subset-minimal")

Another example of knowledge that is useful for modifications of methods is whether configurations (methods) give the same solutions. This enables us to exclude modifications before verifying, and therefore to avoid the expensive simulation verify.

For example, in diagnosis we have the knowledge that when the computed sets of explanations are equal, we know that using the same values for the *Selection* and *Solform* components will result in the same solutions for these two methods. We can use this in avoiding a useless repair-action:

$$\begin{array}{l} \textit{same-Es}(\ ds(Obsmap_1, Voc_1, Cover_1, NotContra_1, Selection, Solform), \\ \qquad\qquad ds(Obsmap_2, Voc_2, Cover_2, NotContra_2, Selection, Solform)) \\ \rightarrow \textit{same-Sols}(\ ds(Obsmap_1, Voc_1, Cover_1, NotContra_1, Selection, Solform), \\ \qquad\qquad\qquad ds(Obsmap_2, Voc_2, Cover_2, NotContra_2, Selection, Solform)) \end{array}$$

(here *Es* and *Sols* refer to the variables of the same name in formula (1)). A way to establish that *same-Es* holds is to use knowledge about properties of the components that are used.

A modification action can consist of modifying an individual component so that it has a desired property, modifying an entire method so that it has a desired property, or tuning components so that they become more compatible. We have mainly studied modification of methods. Finding the appropriate modification step can be a complex process that might consist of generating possible repairs, and preferring those that are "closest" to the original component. In Section 6.2, we illustrate such a complex repair-action.

6 A Scenario of the proposed PCM-method

In this section we illustrate our PSM-method for configuration of methods. We start with an initial configuration problem: a diagnostic case to be solved, plus assumptions and goals to be satisfied by the diagnostic method that we will configure. We then pass through the various steps of our method, each indicated with a $\triangleright$. The entire succession of steps is graphically depicted in Figure 9.

The amount of detail in which we have described the scenario might seem somewhat excessive. The reason for this amount of detail is that we can now ensure that each of the steps in our scenario is implementable. In fact, in (ten Teije & van Harmelen, 1996b) we have described an architecture which we have implemented using logic-programming and meta-reasoning techniques, and which is powerful enough to directly implement each of the steps that occur in the scenario of this section.

6.1 The Input-problem

The input of automated flexible diagnostic problem solving is the diagnostic problem, the assumptions that must be respected, and the desired goals. The diagnostic problem contains domain knowledge of the system under diagnosis (the behaviour model, *BM*), the observed behaviour and the context. Our diagnosis problem is in a car domain, and we use the domain model of Figure 6. The case contains two observations: *lights(yes)* and *engine-starting(no)* and there is no context information. The desired goals are: "use a standard notion of explanation" (*explanation-notion(standard)*), and "at most two alternative diagnoses are allowed" (*max-number-diagnoses(2)*). The given assumption is that "the causes are different in likelyhood".

$$\begin{array}{lcl} BM & = & \text{Figure 6} \\ OBS & = & \{lights(yes), \textit{engine-starting}(no)\} \\ Goals & = & \{\textit{explanation-notion}(standard), \textit{max-number-diagnoses}(2)\} \\ Assumptions & = & \text{the causes are different in likelyhood} \end{array} \tag{2}$$

The scenario described in the next section will show the steps for computing the outputs of this flexible diagnostic solving problem.

6.2 The steps in the PCM method

$\triangleright_1$ Propose

We have to propose a method with definitions for each of the six components. The goal that guides the choice for the *Cover* and *NotContra* components is *explanation-notion(standard)*, since the system contains knowledge that standard entailment is most frequently used in diagnostic methods (as opposed to the use

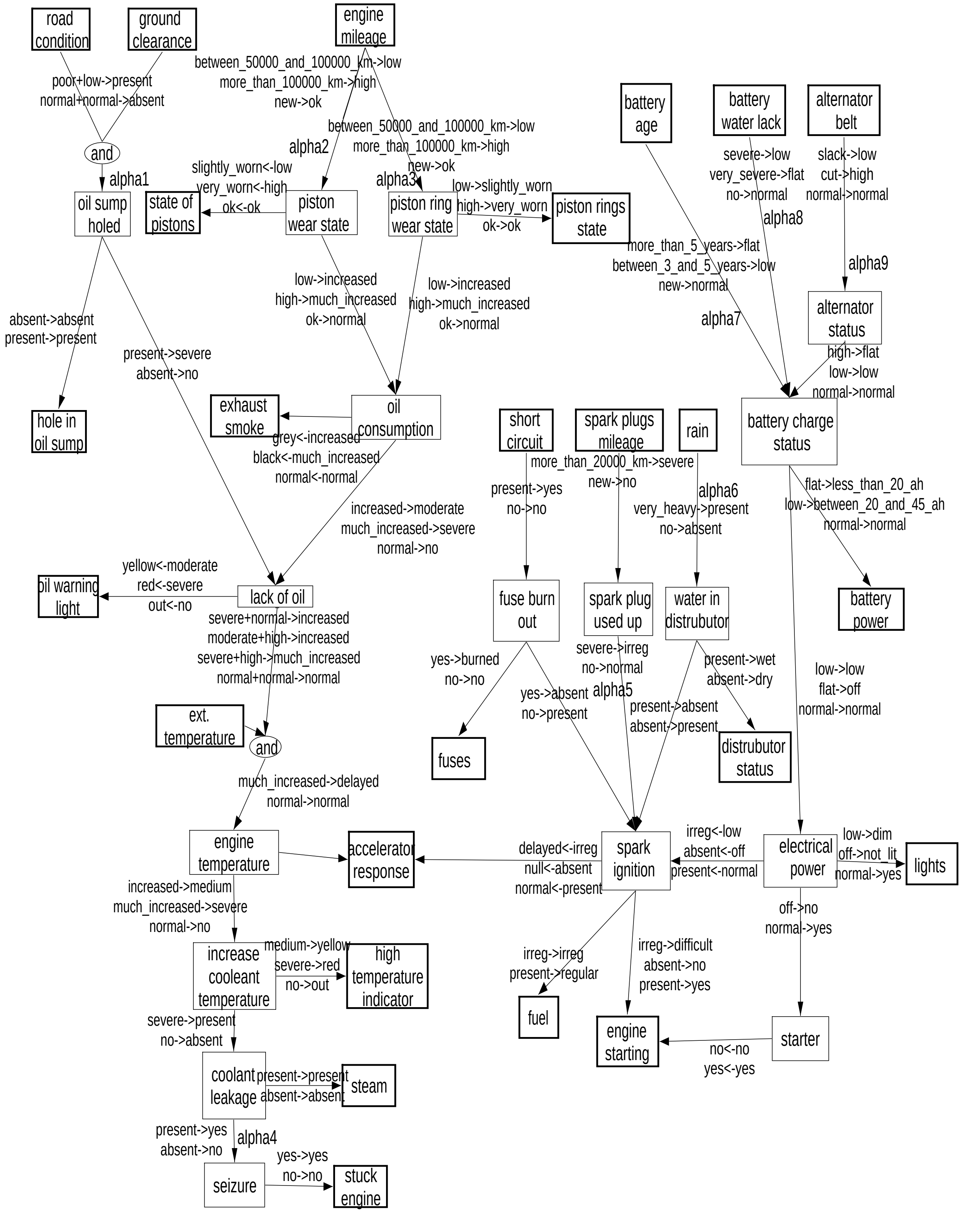

FIGURE 6: Behaviour model (BM) of a car from (Dupré, 1994). The bold lined boxes are initial causes, conditions, and observables.

type component	name	description
Obs-mapping	abd-obs	all observations have to be covered: Obs_{cov} contains all observations, and Obs_{con} is empty
Vocabulary	initial-fault-nodes	the vocabulary contains all initial fault nodes and incompleteness assumptions (α's)
Cover	$\vdash$	use of standard entailment
NotContra	$\not\vdash$	use of standard entailment
Selection	*#-min*	an explanation is selected if it contains the smallest number of causes
Solform	$=$	no effective solform (thus: a minimal set of explanations is also the diagnosis set)

FIGURE 7: The components of the proposed method.

of non-standard variatons of entailment proposed in (ten Teije & van Harmelen, 1996a)). As a result, we choose for both explanation relations (*Cover* and *NotContra*) the classical entailment relations ($\vdash$ and $\not\vdash$ respectively). The other four components are chosen blindly.

The proposed method is[3]:

$$ds(abd\text{-}mapping, initial\text{-}fault\text{-}nodes, \vdash, \not\vdash, \#\text{-}min, =) \tag{3}$$

In *Table 7* the definitions of the components are briefly described. In (ten Teije & van Harmelen, 1994) the definitions of some of these components are given more formally.

The dynamic goals now become all the goals that have not already been statically determined. In this case the only dynamic goal is *max-number-diagnoses(2)*.

$\rhd_1$ Knowledge-verification

One of the usual constraints on diagnostic methods is to demand that the *Cover* component is at least as strong as the *NotContra* component. After all, if an observable is entailed by a consistent theory, then that observable is also consistent with this theory.

The method described by term (3) does not violate this constraint. However, there is another assumption conflict, because *#-min* assumes that every cause has equal likelyhood. This means that the knowledge-verification step has failed, and therefore a new propose step is started.

$\rhd_2$ Propose

We now propose another method. Because this step is still guided by the same static goal as before (*explanation-notion(standard)*), the method still contains $\vdash$ as *Cover* definition and $\not\vdash$ as *NotContra* definition. The other components are again chosen blindly. The propose method is now[4]:

$$ds(abd\text{-}mapping, initial\text{-}fault\text{-}nodes, \vdash, \not\vdash, \subset\text{-}min, =) \tag{4}$$

$\rhd_2$ Knowledge-verification

As in " $\rhd_1$ knowledge-verification" there is no violation of the constraint concering the *Cover* and *NotContra* components. The other assumption-confllict also disappears because $\subset$ *-min* does not assume equal likelyhood of causes. As a result, in this case assumptions conflicts no longer occur.

$\rhd_2$ Simulation-Verify

In this step the system performs diagnosis using the valid method of term (4). Based on the computed diagnoses it tests the dynamic goals.

Using the method of the term (4) results in the following $Obs_{cov} = \{engine\text{-}starting(no), lights(yes)\}$ and $Obs_{con} = \emptyset$. The vocabulary defined by *initial-fault-nodes* contains all the initial nodes of Figure 6 that correspond to fault-modes, plus all the assumption-symbols α_i. Performing diagnosis results in "no diagnosis", which becomes the verification result.

$\rhd_2$ Critique

[3]We write *#-min* for number-minimality, and $=$ for the identity mapping

[4]We write $\subset$ *min* for subset-minimality.

name	Obs_{cov}, Obs_{con}	intuition
complete-mapping:	$Obs_{cov} = OBS$ $Obs_{con} = \{\neg o_i : o_i \in \mathcal{O} \setminus OBS\}$	all observations must be covered the absence of all other observables must be consistent
abd-mapping:	$Obs_{cov} = OBS$ $Obs_{con} = \emptyset$	all observations must be covered with no further consistency check (this is abductive diagnosis)
cbd-mapping:	$Obs_{cov} = \emptyset$ $Obs_{con} = OBS$	we demand no cover and only that all observations are consistent with *BM* (this is consistency-based diagnosis)
abnormality-mapping:	$Obs_{cov} = \{o \in OBS \vert abnormal(o)\}$ $Obs_{con} = \{o \in OBS \vert normal(o)\}$	all abnormal o must be covered and all normal o need only be consistent
polarity-mapping:	$Obs_{cov} = \{o_i \in OBS\}$ $Obs_{con} = \{\neg o_i \in OBS\}$	all positive o must be covered and all negative o must be consistent

FIGURE 8: $\mathcal{O}$ denotes the possible observable values, and OBS denotes the currently given observations

The reason for not finding any diagnoses is that there is no explanation for *lights(yes)*: only incompleteness assumptions (α's) and faults are part of the vocabulary (*initial-fault-nodes*), and a fault cannot explain the correct behaviour of *lights(yes)* when we use $\vdash$ and $\not\vdash$ for *Cover* and *NotContra* respectively. This step determines that a possible suitable repair action is adapting the *Obs-mapping*. This is so because a different *Obs-mapping* might require only incorrect behaviour to be explained (as apposed to all behaviour, including correct behaviour, as is the case withe the current definition, namely *Obs-mapping=abd-mapping*).

$\rhd_2$ **Modify**

The modify step must now repair the component specified by the critique step. The repair action is determined by first generating a set of variants of the *Obs-mapping*, and then applying two filters on this generated set of *Obs-mapping* definitions.

• **generate**: generate variants of the *Obs-mapping* component.

We require that any solution of the method using the original *Obs-mapping* is also a solution for the adapted method with the new *Obs-mapping* (after all, we want to increase the set of solutions). This relation is expressed in the predicate *subset-Es(Old,New)*. It denotes that the explanations generated by a method with *Obs-mapping*-component *Old* are also generated by a method with *Obs-mapping*-component *New*, provided all the other components remain the same. In this generation step we generate those *Obs-mapping* definitions which satisfy *subset-Es(abd-mapping,New)*:

$$\begin{array}{l} \{New \vert subset\text{-}Es(Old, New)\} = NewObsmapSet \\ \rightarrow \\ generate(ds(Old, Vocabulary, Cover, NotContra, Selection, Solform), NewObsmapSet) \end{array}$$

For our problem, the system generates the following set based on its factual knowledge of *subset-Es*:

$$\begin{array}{l} subset\text{-}Es(complete\text{-}mapping, abd\text{-}mapping) \\ subset\text{-}Es(abd\text{-}mapping, cbd\text{-}mapping) \\ subset\text{-}Es(abd\text{-}mapping, abnormality\text{-}mapping) \\ subset\text{-}Es(abd\text{-}mapping, polarity\text{-}mapping) \end{array} \tag{5}$$

The complete definitions of these *Obs-mapping* components are in (ten Teije & van Harmelen, 1994), but in sloppy notation these definitions are given in table 8.

The generated set of *Obs-mapping* definitions is now:

$$\{cbd\text{-}mapping, nor\text{-}abnorm\text{-}mapping, polarity\text{-}mapping\}$$

because *subset-Es(abd-mapping,New)* holds for these elements.

• *filter*$_1$**:** of all the possible candidate repairs, we prefer the variants that are the "closest" to the original *Obs-mapping* component:

$$\{New | New \in ObsmapSet \wedge closest\text{-}Obs\text{-}mapping(Old, PossibleSet, New)\} = FilteredSet$$
$$\rightarrow$$
$$filter_1(ds(Old, Vocabulary, Cover, NotContra, Selection, Solform), ObsmapSet, FilteredSet)$$

We define "closest" as those *Obs-mapping* definitions whose Obs_{cov} set is (1) in any case no superset of the original Obs_{cov} set (since we do not want to explain more observable values strongly) and (2) is not a subset of another possible Obs_{cov} set (since we want to delete as few observable values as possible).

The predicate *closest-Obs-mapping* is therefore defined as follows, whereby $subset\text{-}Obs_{Cov}(X, Y)$ denotes that the *Obs-mapping* X gives an Obs_{cov} set that is a subset of the Obs_{cov} set computed by the *Obs-mapping* Y.

$$\begin{array}{ll}
\multicolumn{2}{l}{ObsmapSelect \in PossibleObsmaps \wedge} \\
\multicolumn{2}{l}{subset\text{-}Obs_{cov}(ObsmapSelect, ObsmapOld) \wedge} \\
\{Obsmap: & subset\text{-}Obs_{cov}(ObsmapSelect, Obsmap) \wedge \\
 & Obsmap \in PossibleObsmap \wedge \\
 & Obsmap \neq ObsmapSelect\} = \emptyset \\
\multicolumn{2}{l}{\rightarrow} \\
\multicolumn{2}{l}{closest\text{-}Obs\text{-}mapping(ObsmapOld, PossibleObsmaps, ObsmapSelect)}
\end{array}$$

We filter the set based on the following factual knowledge of $subset\text{-}Obs_{cov}$:

$$\begin{array}{l}
subset\text{-}Obs_{cov}(complete\text{-}mapping, abd\text{-}mapping) \\
subset\text{-}Obs_{cov}(abd\text{-}mapping, complete\text{-}mapping) \\
subset\text{-}Obs_{cov}(abnormality\text{-}mapping, abd\text{-}mapping) \\
subset\text{-}Obs_{cov}(polarity\text{-}mapping, abd\text{-}mapping) \\
subset\text{-}Obs_{cov}(cbd\text{-}mapping, abnormality\text{-}mapping) \\
subset\text{-}Obs_{cov}(cbd\text{-}mapping, polarity\text{-}mapping)
\end{array}$$

This factual knowledge, just as the knowledge in (5), is stored as given facts in our system. However, given sufficiently powerful theorem-proving techniques, it would be possible for the system to automatically derive these facts from the definitions in table 8. From these definitions, it follows that *closest-Obs-mapping* holds for the *Obs-mapping* definitions *polarity-mapping* and *abnormality-mapping*. The FilteredSet is therefore:

$$\{abnormality\text{-}mapping, polarity\text{-}mapping\} \tag{6}$$

- $filter_2$**:** We now filter those variants which result in the same solutions as the original method in the current case. In this filter the system executes a part of the diagnosis, namely the *Obs-mapping* definition. The results of the possible *Obs-mapping* definitions have to be computed and compared with the outputs of the original *Obs-mapping*. Those which give the same Obs_{cov} and Obs_{con} will be deleted from the set. In contrast with $filter_1$, this filter is specific for the current problem on hand, whereas the $filter_1$ was independent of the problem.

Applying the *Obs-mapping* definitions from (6) to *OBS*=\{*light(yes),engine-starting(no)*\} gives the following values for Obs_{cov} and Obs_{con}:

Obs-mapping	Obs_{cov}	Obs_{con}
abd-mapping	\{*light(yes),engine-starting(no)*\}	$\emptyset$
abnormality-mapping	\{*light(yes)*\}	\{*engine-starting(no)*\}
polarity-mapping	\{*light(yes),engine-starting(no)*\}	$\emptyset$

We see that the *Obs-mapping* with value *polarity-mapping* gives the same sets as the original *Obs-mapping* (which had value *abd-mapping*). The *Obs-mapping* with value *abnormality-mapping* gives other sets. This results in a FilteredSet where the only *Obs-mapping* is *abnormality-mapping*.

The modify therefore results in the method:

$$ds(abnormality\text{-}mapping, initial\text{-}fault\text{-}nodes, \vdash, \not\vdash, \subset\text{-}min, =) \tag{7}$$

The originally proposed method of term (4) could not handle observed behaviour that was correct behaviour. The above critique-&-modify step tried to recover from this shortcoming by adapting the *Obs-mapping* component, resulting in the method from term (7). The next step is to verify the adapted method.

$\triangleright_3$ **Knowledge-verification**

The knowledge verification still succeeds, since the *Cover*-, *NotContra*- and *Selection*-components and the assumptions have not changed. (see " $\triangleright_1$ Knowledge-verification")

$\triangleright_3$ **Simulation-verification**

Again we perform diagnosis, but now using the modified method of term (7). Performing diagnosis results in the following diagnoses:

$$\begin{array}{l} \{short\text{-}circuit(present)\} \\ \{battery\text{-}age(more\text{-}than\text{-}5\text{-}years), \alpha_7\} \\ \{battery\text{-}water\text{-}lack(very\text{-}severe), \alpha_8\} \\ \{alternator\text{-}belt(cut), \alpha_9\} \end{array} \tag{8}$$

Unfortunately, the test whether the dynamic goal *max-number-diagnoses(2)* is satisfied fails. This means we have to perform another critique step.

$\triangleright_3$ **Critique**

In the verification step the problem of too many solutions was recognized. A repair action for this problem is a modification of the *Selection* component. If the new *Selection* component is a stronger filter, then less diagnoses will be left. The system uses the knowledge that constructing the conjunction of the current *Selection*-component with an additional selection criterion will have this effect.

$\triangleright_3$ **Modify**

The repair action of configuring the new *Selection* criterion is executed in this step. In our case the *Vocabulary* (*initial-fault-nodes*) contains faults and incompleteness-assumptions. We can therefore eomply a selection-criterial that prefers explanations which are subset-minimal in the incompleteness assumptions ($\subset$ *-min-in-*α). The proposed *Selection* criterion then becomes "$\subset$ *-min* **and** $\subset$ *-min-in-*α".

The adapted method is:

$$ds(abnormality\text{-}mapping, initial\text{-}fault\text{-}nodes, \vdash, \not\vdash, \subset\text{-}min \textbf{ and } \subset\text{-}min\text{-}in\text{-}\alpha, =) \tag{9}$$

The proposed method from (7) resulted in too many diagnoses. The above critique and modify steps tried to recover from "too many diagnosis" and have modified the method. This modified method now has to be verified.

$\triangleright_4$ **Knowledge-verification**

The knowledge verification still satisfies, as before ($\subset$ *-min-in-*α, also does not violate the unequal-likelyhood assumption).

$\triangleright_4$ **Simulation-verification**

Again we perform diagnosis using the modified method of term (9). Performing diagnosis results in the following diagnosis:

$$\{short\text{-}circuit(present)\} \tag{10}$$

Checking this against the dynamic goal shows that we have now also satisfied *max-number-diagnoses(2)*.

We have now (finally!) solved the original diagnostic problem specified in (2). The method of term (9) has explained the observations {*engine-starting(no),lights(yes)*} under the assumption "the causes are different in likelyhood" for the desired goals "use a standard notion of explanation" and "at most two alternative diagnoses are allowed". The sole computed diagnosis is (10).

During this diagnostic problem solving process the configuration system has had to recover from the initial inability to deal with correct behaviour (by modifing the *Obs-mapping* component) and it had to recover from "too many solutions" caused by too weak a selection filter (by modifying the *Selection* component).

6.3 Alternatives for Critique & Modify Steps

In this section we give alternatives of the critique and modify steps of the above scenario. How to choice between these alternative actions. is still subject of study. The search space which is generated by the trace described above and the alternatives describe below is depicted in Figure 9.

6.3.1 Alternatives for " $\rhd_2$ Critique & Modify"

We propose two alternatives for recovering from the impossibility to handle correct behaviour.

$\rhd_{2'}$ Critique

An alternative for the critique step is to better tune the *Obs-mapping* component to the *Vocabulary* component *initial-fault-nodes*. In general, the combination of *abd-mapping* and *initial-fault-nodes* is not an obvious choice, because using the *initial-fault-nodes* assumes that only abnormal behaviour is observed. However, the given problem contains also normal behaviour *light(yes)*.

A more obvious choice of *Obs-mapping* component can be determined by checking whether we have observed both normal and abnormal behaviour. This is a case specific repair action, because we use the current observed behaviour in the choice of *Obs-mapping*. The abnormality or normality of the observed behaviour is checked using the *abnormality-mapping Obsmap* component. If execution of *abnormality-mapping* results in a non-empty set of Obs_{cov} then we use the knowledge that the combination of *initial-fault-nodes* and *abd-mapping* is a bad combination, and *abnormality-mapping* is probably a better one.

$\rhd_{2'}$ Modify

The previous critique step results in the same method as the " $\rhd_2$ Modify" step, namely term (7).

$\rhd_{2''}$ Critique

The other alternative for " $\rhd_2$ Critique & Modify" is to adapt the *Vocabulary* component. If the *Obs-mapping* component *abnormality-mapping* does not result in an empty Obs_{cov} set, then a better choice of *Vocabulary* is possibly *all-initial-nodes*. This vocabulary contains all initial causes (including correct states) and the incompleteness assumptions, and is therefore better tuned to *Obs-mapping=abd-mapping* .

$\rhd_{2''}$ Modify

We would now come to another method then before, namely:

$$ds(\textit{abd-mapping}, \textit{all-initial-nodes}, \vdash, \not\vdash, \subset\textit{-min}, =) \tag{11}$$

$\rhd_{3''}$ Knowledge-verification

The knowledge verification still satisfies, as before.

$\rhd_{3''}$ Simulation-verification

Performing diagnosis results in the following diagnosis part

$$\begin{array}{ll}
\text{for } \textit{light(yes)}: & \{\textit{battery-age}(\textit{new}), \alpha_7\}, \\
& \{\textit{battery-water-lack}(\textit{no}), \alpha_8\}, \\
& \{\textit{alternator-belt}(\textit{normal}), \alpha_9\}. \\
\text{for } \textit{engine-starting(no)}: & \{\textit{short-circuit}(\textit{present})\}, \\
& \{\textit{rain}(\textit{very-heavy}), \alpha_6\}, \\
& \{\textit{battery-age}(\textit{more-than-}5\textit{-years}), \alpha_7\}, \\
& \{\textit{battery-water-lack}(\textit{very-severe}), \alpha_8\}, \\
& \{\textit{alternator-belt}(\textit{cut}), \alpha_9\}
\end{array} \tag{12}$$

This yields 3 x 5 = 15 diagnoses, so we have too many possible diagnoses. We end up with these other diagnoses because the critique and modify steps are based on the observation that the vocabulary was too small, whereas before the devision of the observations was considered as wrong. After verification we establish that "too many solutions" are computed. A repair action for solving "too many diagnoses" is needed. We do not describe this trace further.

6.3.2 Alternative for $\rhd_3$Critique & Modify

Finally, we give an alternative for the critique and modify steps that tries to recover from "too many diagnoses".

$\triangleright_{3'}$ **Critique**

In analysing the failure of the verification step, the system uses the knowledge that if the number of observations four or less and there are too many diagnoses, then the repair action becomes "ask to the user the relevant observables for the computed diagnosis". This repair action changes the input problem (since additional observations are requested). In contrast, the previous repair actions only changed the method.

$\triangleright_{3'}$ **Modify**

New observables need to be asked from the user. The relevant observables are those which are connected to a cause of the computed set of diagnoses, but that are not already part of the observed behaviour. Our set of observables for asking to the user is therefore based on the causes: *short-circuit*, *battery-age*, *battery-water-lack* and *alternator-belt*.

In our problem the following observables are asked:

fuses, distributor-status, accelerator-response, fuel, battery-power.

The user gives only a value for *distributor-status*, namely *wet*. The new observation theory contains therefore:

$$engine\text{-}starting(no) \wedge light(yes) \wedge distributor\text{-}status(wet) \tag{13}$$

The diagnostic problem has now been adapted by adding new information.

$\triangleright_{4'}$ **Knowledge-verification**

The knowledge verification still satisfies as before.

$\triangleright_{4'}$ **Simulation-verification**

We perform diagnosis using the adapted problem and the method of term (7). This results in just one diagnosis:

$$\{rain(very\text{-}heavy), \alpha_6\} \tag{14}$$

We continue with the test of the dynamic goals, namely *max-number-diagnose(2)*, which succeeds. The diagnosis problem is now solved. Notice that we end up with another diagnosis then in the previous scenario, where it was. *short-circuit*). This is because we recover from "too many solutions" by asking new observables, whereas in the first scenario we made the *Selection* component stronger.

6.4 Scenario Conclusions

This scenario has illustrated our proposed PCM method. It shows the method proposal (guided by the static goals), the knowledge-verification (of constraints and assumptions), the simulation-verification (by computing diagnoses and testing the dynamic goals), and the critique and modify (for recovering from violations of dynamic goals). We have show three possibilities for recovering from the inability to handle normal observed behaviour:

- by adapting the *Obs-mapping* as a way to get more explanations;
- by adapting the *Vocabulary*, to better tune the *Cover*, *Obsmap* and *Vocabulary* components to each other;
- by choosing an *Obs-mapping* that depends on the current observations.

Furthermore we have shown three possibilities of recovering from "too many solutions":

- by strengthening the *Selection* component, because the used *Selection* is a to weak filter.
- by adding more data, because more data will exclude diagnoses;

The scenario and the alternative paths are summarised in Figure 9. The choice on the brancing nodes of this search space (ie. which repair action has to be taken) is difficult, and needs further study.

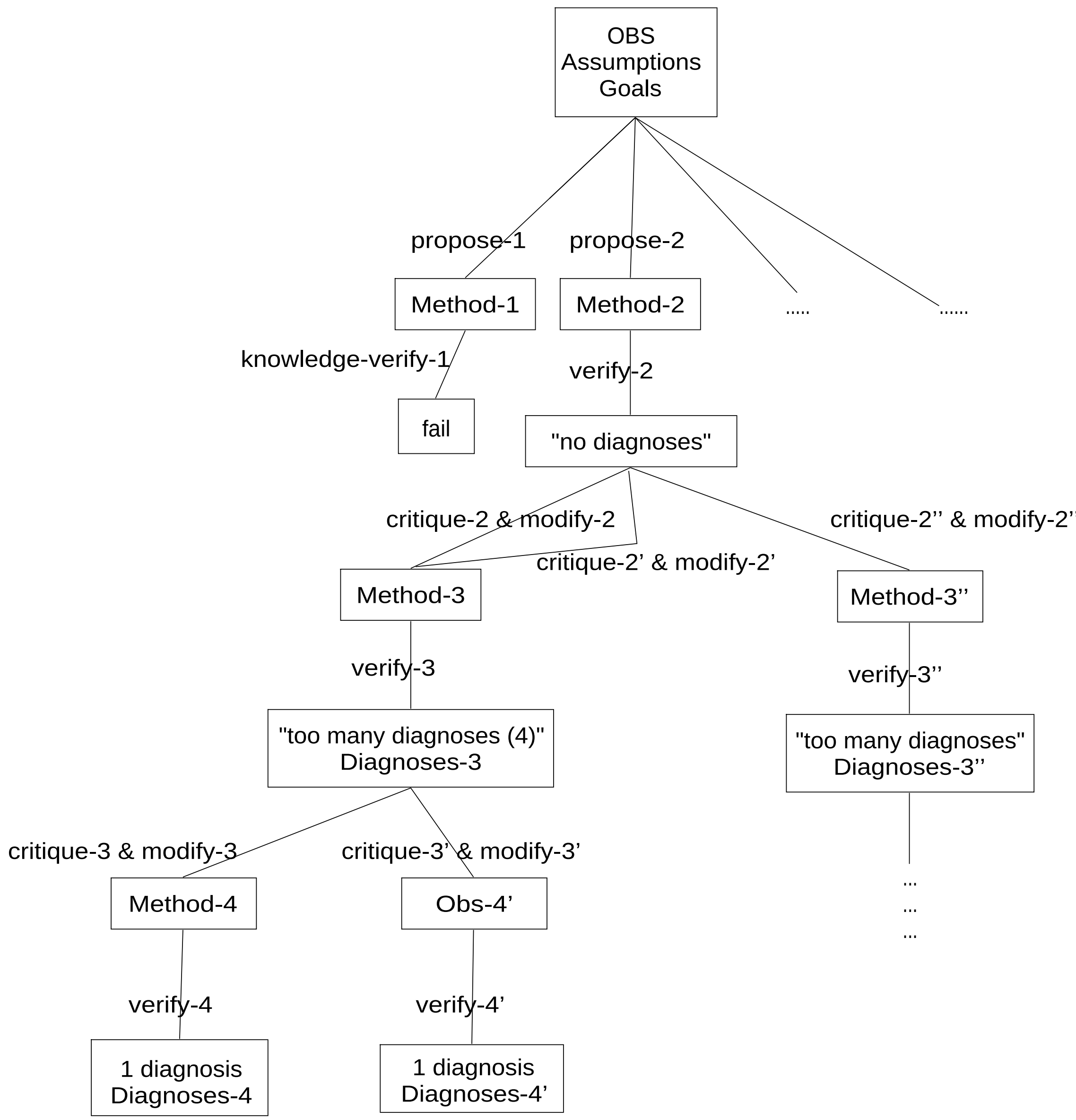

FIGURE 9: The steps of the PCM method in the scenario and the alternative steps. The *Method-i* and the *Diagnoses-i* are explained in table 1.

Method-i Diagnoses-i	value	equation
OBS	{lights(yes),engine-starting(no)}	(2)
Goals	{explanation-notion(standard), max-number-diagnoses(2)}	(2)
Assumptions	the causes are different in likelyhood	(2)
Method-1	ds(abd-mapping, initial-fault-nodes, $\vdash$,$\not\vdash$, #-min,=)	(3)
Method-2	ds(abd-mapping, initial-fault-nodes, $\vdash$, $\not\vdash$, $\subset$-min, =)	(4)
Method-3	ds(abnormality-mapping, initial-fault-nodes, $\vdash$, $\not\vdash$, $\subset$-min, =)	(7)
Method-3”	ds(abd-mapping, all-initial-nodes, $\vdash$, $\not\vdash$, $\subset$-min, =)	(11)
Method-4	ds(abnormality-mapping, initial-fault-nodes, $\vdash$, $\not\vdash$, $\subset$-min **and** $\subset$-min-in-α, =)	(9)
Obs-4’	engine-starting(no), light(yes), distributor-status(wet)	(13)
Diagnoses-3	{short-circuit(present)} {battery-age(more-than-5-years),α_7}	
	{battery-water-lack(very-severe),α_8} {alternator-belt(cut),α_9}	(8)
Diagnoses-3”	...	(12)
Diagnoses-4	{short-circuit(present)}	(10)
Diagnoses-4’	{rain(very-heavy),α_6}	(14)

TABLE 1: Explanation of the terms in figure 9

7 Conclusion & Related Work

In this paper we have given a proposal for the automated configuration of problem solvers for an arbitrary problem-type. Because we use a parameterised schema for describing a problem solver, we are able to regard configuration of problem solvers as a parametric design problem. Our propose-critique-modify method for configuration of (the functionality of) PSMs uses several knowledge types. Our basic assumption is that we exploit the much knowledge of the problem type (in our case diagnosis) for the configuration of problem solvers.

Although parametric desing is classified as “routine design” this does not imply that it is an easy problem to solve. This is in line with our experience. It is difficult to instantiate various knowledge types for the case of diagnosis. However, these knowledge types enable us to come to grips with the complex problem of the automated configuration of (diagnostic) PSMs.

Related Work In the knowledge-engineering literature we find approaches for selecting and configuring problem solving methods (e.g. (Istenes *et al.*, 1996; Stroulia & Goel, 1994; Benjamins, 1993)). None of these approaches validate the selected or configured method or are able to execute this method. These systems are only intended for selecting or configuring a method, and checking whether the method does indeed work effectively is outside their scope. All these systems use a method-decomposition tree for describing a method. In (Istenes *et al.*, 1996) the kind of operations on methods are: select a method, identify a possible method, choose the most favourable method. The approach in (Stroulia & Goel, 1994) comes from the field of knowledge acquisition. The system make choices, advises and interacts with the user during the configuration of a PSM. Examples of feedback of the system are: there is an error in the decomposition tree (e.g. input/output do not match), places which need more specific application conditions of methods, suggestions for changes for data or knowledge. In (Benjamins, 1993) the configuration of methods is based on a decomposition tree of tasks and methods. The decisions for the choice of a method are taken locally at each node, without referring to descendants, ancestors and siblings. The necessary and suitability application criteria determine the choice of the method. The primitive methods of such a tree are labels, which refer to a semi-formal description of the method. The contents of these “labels” does not influence the choices during method configuration.

For all these three systems, selecting a method means selecting an informal or semi-formal description of the method. This is a description that is oriented on algorithmic aspects of the method. However, one would expect that the functionality of the method also plays a role in method selection.

Remarkable is that all these theories of selecting and configuring problem solving methods are abstract and very high-level. This is a consequence of the desire to generalise the description of methods across very different families of methods, and therefore to avoid the use of specific knowledge, for instance knowledge of design methods. In our view, we have to exploit domain specific knowledge for strengthening of theories of problem solving methods.

Acknowledgment This work has benefited from many discussions with Dieter Fensel. We also thank Richard Benjamins and Remco Straatman for their comments on an earlier version.

References

Benjamins, V. R. (1993). *Problem Solving Methods for Diagnosis.* PhD thesis, University of Amsterdam, Amsterdam, The Netherlands.

Brown, D. C. & Chandrasekaran, B. (1989). *Design Problem Solving: Knowledge Structures and Control Strategies.* Research Notes in Artificial Intelligence. London, Pitman.

Chandrasekaran, B. (1990). Design problem solving: A task analysis. *AI Magazine*, 11:59–71.

Clancey, W. J. (1985). Heuristic classification. *Artificial Intelligence*, 27:289–350.

Console, L., de Kleer, J., & Hamscher, W. (1992). *Readings in Model-based Diagnosis.* Morgan Kaufmann.

Dupré, D. (1994). *Characterizing and Mechanizing Abductive Reasoning.* PhD thesis, Universita di Torino.

Fensel, D. (1995). A case study on assumptions and limitations of a problem solving method. In *Proceedings of the 9th Banff Knowledge Acquisition for Knowledge-Based System Workshop (KAW'95)*, Banff, Canada.

Hayes-Roth, F., Waterman, D. A., & Lenat, D. B. (1983). *Building Expert Systems.* New York, Addison-Wesley.

Istenes, Z., Tchounikine, P., & Trichet, F. (1996). Using zola to model dynamic selection of tasks and methods as a knowledge level reflective activity. In *Europeen Knowledge Acquisition Workshop (Position Papers*, pages 42–53, Nottingham.

Löckenhoff, C. & Messer, T. (1994). Configuration. In Breuker, J. A. & Van de Velde, W., (Eds.), *The CommonKADS Library for Expertise Modelling*, chapter 9, pages 197–212. Amsterdam, The Netherlands, IOS Press.

Marcus, S., Stout, J., & McDermott, J. (1988). VT: An expert elevator designer that uses knowledge-based backtracking. *AI Magazine*, Spring:95–111.

Mittal, S. & Frayman, F. (1989). Towards a generic model of configuration tasks. In *Proceedings of IJCAI'89*, pages 1395–1401.

Runkel, J., Birmingham, W., & Balkany, A. (1995). Solving vt by reuse. *Int. J. Human-Computer Studies*, (43).

Schreiber, A. T. & Birmingham, W. P. (1996). The Sisyphus-VT initiative. *International Journal of Human-Computer Studies.* Editorial special issue.

Stroulia, E. & Goel, A. (1994). Reflective, self-adapative problem solvers. In Steels, L., Schreiber, G., & van de Velde, W., (Eds.), *Proceedings of the 8th European Knowledge Acquisition Workshop (EKAW'94)*, volume 867 of *Lecture Notes in Artificial Intelligence*, pages 394–413, Hoegaarden, Belgium. Springer Verlang.

ten Teije, A. & van Harmelen, F. (1994). An extended spectrum of logical definitions for diagnostic systems. In *Proceedings of DX-94 Fifth International Workshop on Principles of Diagnosis*, New Paltz, New York.

ten Teije, A. & van Harmelen, F. (1996a). Computing approximate diagnoses by using approximate entailment. In *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR'96)*, Boston, Massachusetts.

ten Teije, A. & van Harmelen, F. (1996b). Using reflection techniques for flexible problem solving. *Future Generation Computer Systems*, special issue Reflection and Meta-level AI Architectures. To appear in 1996.

Wielinga, B. J., Akkermans, J. M., & Schreiber, A. T. (1995). A formal analysis of parametric design problem solving. In Gaines, B. R. & Musen, M. A., (Eds.), *Proceedings of the 8th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop*, volume II, pages 37.1–37.15, Alberta, Canada. SRDG Publications, University of Calgary.