# Structured Development of Problem Solving Methods

Dieter Fensel and Enrico Motta

**Abstract**—Problem solving methods (PSMs) describe the reasoning components of knowledge-based systems as patterns of behavior that can be reused across applications. While the availability of extensive problem solving method libraries and the emerging consensus on problem solving method specification languages indicate the maturity of the field, a number of important research issues are still open. In particular, very little progress has been achieved on foundational and methodological issues. Hence, despite the number of libraries which have been developed, it is still not clear what organization principles should be adopted to construct truly comprehensive libraries, covering large numbers of applications and encompassing both task-specific and task-independent problem solving methods. In this paper, we address these "fundamental" issues and present a comprehensive and detailed framework for characterizing problem solving methods and their development process. In particular, we suggest that PSM development consists of introducing assumptions and commitments along a three-dimensional space defined in terms of *problem-solving strategy*, *task commitments*, and *domain (knowledge) assumptions*. Individual moves through this space can be formally described by means of *adapters*. In the paper, we illustrate our approach and argue that our architecture provides answers to three fundamental problems related to research in problem solving methods: 1) what is the epistemological structure and what are the modeling primitives of PSMs? 2) how can we model the PSM development process? and 3) how can we develop and organize truly comprehensive and manageable libraries of problem solving methods?

**Index Terms**—Knowledge modeling, problem-solving methods, ontologies, knowledge engineering, software engineering, formal languages.

## 1 Introduction

*Problem solving methods* (PSMs) describe the reasoning components of knowledge-based systems as patterns of behavior that can be reused across applications. For instance, the problem solving method *Propose & Revise* ([56], [92]) provides a generic reasoning pattern, characterized by iterative sequences of model "extension" and "revision," which can be reused when solving—for instance—scheduling [81] or design [56] problems. Problem solving methods define an important technology for supporting structured development approaches in knowledge engineering: they 1) provide strong model-based frameworks in which to carry out knowledge acquisition ([55], [87]) and 2) support the rapid development of robust and maintainable applications through component reuse ([19], [72], [59]). More in general, the study of problem solving methods can be seen as a way to move beyond the notion of knowledge engineering as an "art" [26], to formulate a task-oriented systematization of the field, which will make it possible to produce rigorous handbooks similar to those available for other engineering fields. A number of papers describing the state of the art in problem-solving method research can be found in [8].

So far, most of the research effort has focused on identifying and defining specific classes of problem solving methods. As a result, several problem solving method libraries are now available ([11], [55], [19], [71], [6], [12], [67], [64], [59], [76]) and a number of problem-solving method specification languages have been proposed, ranging from informal notations (e.g., CML [74]) to formal modeling languages—see [41] and [28] for comprehensive surveys. Some of these libraries provide executable reasoning components (for example, [59]), others (e.g., the CommonKADS library [12]) provide only conceptual models of such components similar to design patterns [43] in OO-design.

Researchers in this area have also (partially) addressed "foundational" issues, concerning 1) the nature of problem solving methods, 2) the relation between problem solving methods on one side and *task* and *domain* knowledge on the other,[1] and 3) the principles underlying the method development process ([90], [29], [59], [61], [30]). Nevertheless, a number of fundamental problems are still open. For instance, while both [90] and [61] characterize PSM development as a task-centered process mediated by the selection of a problem solving strategy, they only provide limited, coarse-grained insights on the type of development steps carried out

- *D. Fensel is with the Division of Mathematics and Computer Science, Vrije Universiteit Amsterdam, The Netherlands. E-mail: dieter@cs.vu.nl.*
- *E. Motta is with the Knowledge Media Institute, The Open University, UK. E-mail: e.motta@open.ac.uk.*

*Manuscript received 1 Oct. 1998; revised 5 May 2000; accepted 8 June 2000; posted to Digital Library 6 Apr. 2001.*
*For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 107480.*

1. Throughout this paper, we will follow the established naming convention in knowledge engineering and use the term "task" to refer to the goal that must be achieved by a problem solver [19]. Thus, task knowledge refers to the knowledge associated with the task specification which has to be achieved by the problem solver.

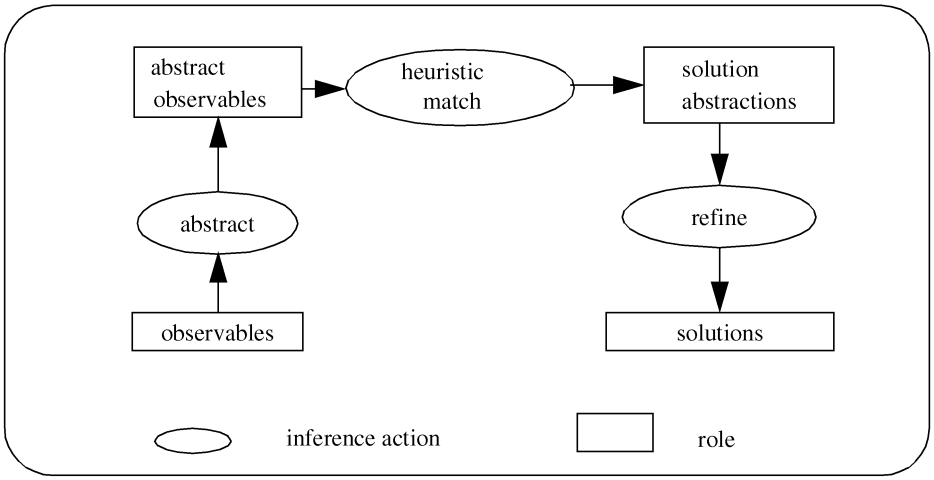


Fig. 1. The heuristic classification inference scheme.

during the process. Moreover, despite the number of libraries which have been developed, it is still not clear what organization principles should be adopted to construct truly comprehensive libraries, covering large numbers of applications and encompassing both task-specific and task-independent problem solving methods.

In this paper, we address these "fundamental" issues and present a comprehensive and detailed framework for characterizing problem solving methods and their development process. In particular, we suggest that PSM development consists of introducing assumptions and commitments along a three-dimensional space defined in terms of a *problem-solving strategy*, a number of task commitments and a number of *domain knowledge assumptions*.

Individual moves through this space can be formally described by means of *adapters* ([37], [29]). In the paper, we illustrate our approach and argue that our architecture provides answers to three fundamental problems related to research in problem solving methods: 1) what is the epistemological structure of PSMs (i.e., what are the generic building blocks and how are they interrelated)? 2) how can we model the PSM development process? and 3) how can we develop and organize truly comprehensive and manageable libraries of problem solving methods?

The paper is organized as follows: In Section 2, we present the research issues we tackle in this paper. In Section 3, we present a three-dimensional framework for structuring the problem-solving method development process and we illustrate a sample method development process. In Section 4, we discuss a typology of adapters, which are the modeling device we use for representing method development steps. Finally, we discuss related work and reiterate the main contributions of our approach.

## 2 Foundational Problems in PSM Research

In what follows, we discuss the three main problems which we are addressing in the paper: 1) the problem of situating problem solving methods with respect to domain and task knowledge; 2) the problem of characterizing and representing the method development process; and 3) the problem of constructing practically usable libraries with sufficient cover. In our view, these problems are foundational, in the sense that no comprehensive theory and practice of method development and use can ignore them. In this section, we will also provide an initial sketch of our proposed solutions to these problems, which will then be described in more detail in later sections.

### 2.1 Problem Solving Methods as Reusable Reasoning Patterns

Problem-solving methods describe reusable reasoning patterns. This reusability is achieved by abstracting from different sources of "noise": *implementation* aspects, *domain* aspects, and (in the case of task-independent methods) *task* aspects. To clarify this fundamental feature of problem solving methods let's recall a well-known case study carried out by [20], who analyzed the problem solving behavior of a set of first generation expert systems. Though these systems were realized using different representation formalisms (e.g., production rules, frames, LISP) and were concerned with different domains and tasks, Clancey's analysis showed that they all subscribed to a common problem solving behavior, which could be described by means of a *generic inference pattern* called *heuristic classification* (see Fig. 1). This inference pattern comprises three basic *inference actions—abstract*, *heuristic match*, and *refine*—and four *knowledge roles—observables*, *abstract observables*, *solution abstractions*, and *solutions*. It is important to emphasize that such a description is given in a generic way. For instance, a solution abstraction could be a literary genre in a book advisory system and a medical disease in a medical diagnosis system. Thus, it is possible to reuse such a problem-solving method for different domains (books or medicine) and tasks (selection or diagnosis).

Unfortunately, an in-depth analysis of the relation between problem-solving methods on one side and domains and tasks on the other shows that this relation is not as straightforward as it might look at first impression and that different trade-offs and modeling approaches are possible within the resulting space. In particular, two important issues arise: the *interaction problem* and the *usability-reusability trade-off*. These are discussed in the next two sections.

#### 2.1.1 The Interaction Problem: How to Relate PSMs and Domains

The *interaction* problem [14] states that domain knowledge cannot be represented independently of how it will be used in reasoning. Vice versa, a problem-solving method and its specific variants cannot be constructed independently of assumptions about the available domain knowledge. In other words, developing a reusable problem-solving method requires the explicit representation of the assumptions the method introduces about the available domain knowledge. We illustrate this aspect with a simple example taken from the analysis of Propose & Revise carried out by [28]. During the propose stage, a model of an artifact is extended. This extension is performed by applying the relevant domain knowledge. If a method can assume that at each step of the process there is always at most one applicable extension operator, then no additional domain knowledge is required. Otherwise, operator selection knowledge is needed to discriminate between multiple applicable operators. Further assumptions on the completeness, correctness, and utility of the model extension knowledge influence the competence of the method (see [28] for more details).

To recap, abstracting from the application domain is necessary for enabling reuse of reasoning patterns. However, different methods introduce different assumptions over the available domain knowledge, which are not necessarily satisfied by all potential application domains, e.g., consider the single-fault assumption required by some diagnostic methods. Hence, the application of a problem solving method to a domain may require an *adaptation* process, whose purpose is to bridge the gap between a method's assumptions and the functionalities provided by the knowledge base of the application domain in hand. For this reason, in contrast with existing libraries of problem solving methods, our approach explicitly integrates support for the adaptation process in the specification of problem solving methods.

#### 2.1.2 The Usability-Reusability Trade-Off: How to Relate PSMs and Tasks

Problem-solving methods were introduced in the literature as specialized task-specific reasoning mechanisms (cf. [16]). They were called *strong methods* by [57] because they make stronger assumptions on the nature of the available domain knowledge and the structure of the target task, than those required by the so-called *weak methods*, e.g., generic search algorithms such as depth-first search, breadth-search, etc., see [13]. Hence, strong methods arose as a reaction to uniform approaches to problem solving, in line with a *functional view of intelligent behavior* [17].

This view of problem solving methods has in recent years been criticized by a number of researchers. In particular, two aspects of the "strong methods" approach have come under fire: 1) the characterization of problem solving methods as task-specific reasoning patterns and 2) the dichotomy between strong and weak methods. This second aspect has been discussed in a number of papers by Motta and Zdrahal ([60], [93], [61], [59]), who show not only that it is possible to reformulate strong methods such as Propose & Revise as specializations of search algorithms, but also that these reformulations help to clarify the competence of these problem solving methods. Hence, as already pointed out in an earlier paper of ours [34], it is not really the case that there is a dichotomy between weak and strong methods: On the contrary, the latter can be constructed by differentiating and specializing the former. The library developed by Motta and Zdrahal [61], [59] provides evidence for this thesis by showing that a class of problem solving methods for parametric design can be constructed by specializing a task-specific formulation of a generic search paradigm.

The characterization of problem solving methods as task-specific reasoning patterns has also been undermined by much recent research. For example, [53] reports on a simple assignment task that was used as a common benchmark to compare and contrast alternative methodologies and tools for knowledge engineering. It turned out that a variety of different problem-solving methods could be applied to this task and, in particular, [44] showed that the problem did not have to be necessarily tackled by means of constructive (i.e., design) methods, but could also be characterized as a classification problem. Thus, there is clearly a $n : m$ relationship between problem solving methods and tasks (different methods can be applied to a task and different tasks can be solved by the same method). However, this reuse across task boundaries is hampered by the task-specific commitments embedded in a method specification. Hence, [9] suggested that problem-solving methods should be specified in a *task-neutral* style, to allow reuse across classes of tasks. Unfortunately, following this approach means that the strong support for knowledge acquisition provided by task-specific formulations is lost.

This situation is an example of what [51] called the *usability-reusability trade-off* of problem-solving methods. The more task-specific a method, the more support for developing an application it can provide, the less reusable it is. Vice versa, the more task-independent a method, the more it is reusable, the less support it provides for a specific application. In this paper, we show that this trade-off can be overcome by grounding problem solving method specifications on a rich framework, which makes it possible to separate the different parts of a problem solving method. Thus, our framework provides three means to solve this problem:

- Algorithmic schemes that describe the *problem-solving strategy* of a problem-solving method: These schemes are free of task-specific commitments. An algorithmic schema represents the essential structure of a certain class of algorithms (cf. [78]).
- Modeling support to allow the refinement of these schemes according to task and domain-specific circumstances: We use the term "externalizing" to indicate that we explicitly provide modeling primitives to support the configuration of library components (e.g., configuring PSM components for other tasks or domains). Because the configuration process is explicitly modeled, it is therefore available in an external form to the users of the library. The importance of recording the knowledge engineering experience in a library has been

recognized for a number of years ([82]; [86]), but until now researchers have focused on recording informal experiences—see Section 5 for a detailed discussion. In our approach, development and adaptation steps are formally modeled and included in the library. Externalization also enables reuse at different levels of refinement. For instance, we can allow a task-neutral method to be used for different tasks, by providing the method plus the appropriate task-specific adaptors. In addition, the externalized adaptors can themselves be reused to specialize different methods. For instance, the same mechanisms used in [25] to refine chronological backtracking can also be used to refine breadth-first search. In this paper, we will use *adapters* [37], [30] to model the external specifications of refinements—see Section 3 for a detailed example.

### 2.2 Developing and Adapting Problem-Solving Methods

Despite the vast literature on problem-solving methods, relatively little work has focused on techniques and models for characterizing and supporting the problem-solving method development process. Early knowledge engineering frameworks (e.g., Role-limiting methods [55] and KADS-1 [11]) only considered complete problem solving methods and provided little insights on the method development process. Later approaches [84], [70], [71], [6], [19], [79], [12], [4] introduced some degree of flexibility, by organizing libraries of problem solving methods in terms of *task-method structures*. In this organization, a problem-solving method $P$ decomposes a task $T$ into subtasks $T_1, \ldots, T_n$ that are recursively solved by problem-solving methods $P_1, \ldots, P_n$. While this organization improves over libraries of monolithic problem solving methods, it is based on a relatively unstructured principle associating a method to the task it solves. Hence, it does not address either the problem of how to develop new components (it only says where to put them), nor how to adapt components for different tasks and domains.

The approach taken in the *VITAL* project [75] is based on the *GDM methodology* ([87], [67]) and uses homogeneous task-subtask trees, in which every node is a task. Therefore, it can be seen as a simplified form of the before mentioned task-method approach and the criticisms we have already made about task-method structures apply here, too. Actually, the situation is worse in the VITAL-GDM approach, given that the choice of a method is implicit in the selection of the subtree, rather than explicit in the decomposition structure.

A principled approach to method development has been proposed in the papers by Akkermans et al. [2] and Wielinga et al. [90], who characterize this process as an *assumption-driven* activity. In their approach, a formal specification of a task is derived from informal requirements by introducing assumptions about the problem and the problem space. This task specification is then refined into a functional specification of the problem-solving method by making assumptions about the problem-solving strategy and the available domain theory.

Our approach builds on these ideas and tries to make them more rigorous and comprehensive by introducing 1) a precise definition of the different dimensions of this process, 2) a typology of the elementary steps required by the PSM development process, and 3) a notation for representing them. In particular, in contrast with the rather generic framework proposed by Akkermans and Wielinga, our approach stresses the view of problem-solving method development as *adaptation* along three dimensions: 1) the algorithmic scheme or problem-solving strategy a problem-solving method subscribes to, 2) the (data) structures a problem-solving method uses to describe its input, intermediate states, and output, and 3) the assumptions on domain knowledge that influence the definition of its elementary inferences and competence.

### 2.3 Managing Libraries of Problem-Solving Methods with Broad Horizontal Cover

Developing manageable libraries of problem-solving methods that provide large horizontal cover is still an open issue. Here, we use the term "horizontal cover" to refer to the range of problems that are covered by a library (i.e., the classes of applications which can be successfully tackled by reusing methods from the library). Early libraries of problem solving methods were very limited. For instance, the original KADS library had only a $1:1$ mapping between methods and problem types (in total there are less than twenty problem-solving methods listed in [11]). Seven years later, the CommonKADS library of [12] provides hundreds of problem-solving methods. However, its coverage is still very limited with respect to the large range of methods which exist in the literature and (more importantly) with respect to the huge number of possible customizations which are possible. For instance, let's consider the already mentioned Propose & Revise method and its application to the configuration of a vertical transportation system (VT domain, cf. [56], [73]). This domain was chosen as a test case in the Sisyphus-II benchmarking initiative and a number of solutions to the problem were published in the special issue of the *International Journal of Human-Computer Studies* reporting on the initiative [73]. One of us [27] analyzed in detail one of the published solutions to the VT problem, in order to derive a formal specification of Propose & Revise. As it turns out, Fensel encountered a number of difficulties. First of all, Propose & Revise makes a number of assumptions about the given configuration problem. Different variants of the method can be identified according to the precise definition of these assumptions. These determine the precise definition of the elementary inferences of the method, as well as its control structure and overall competence—see also [92]. None of these variants can be taken to be the "gold standard" for this class of methods: Different variants are relevant in different application domains. Therefore, one cannot choose a particular variant when building a library: This approach

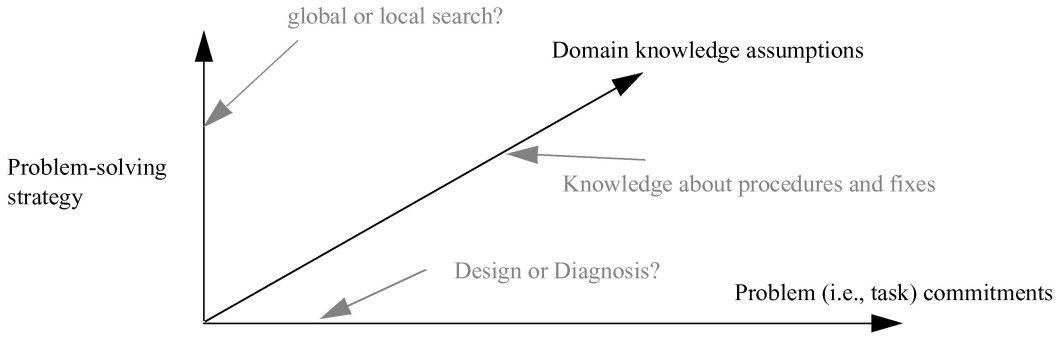


Fig. 2. The three dimensions of PSM description and development.

leads to nonreusable problem-solving methods.[2] But, adding all possible variants of Propose & Revise into the library does not look very promising either, given the large number of different variants. Moreover, Propose & Revise is only one problem solving method. Applying this strategy to all problem-solving methods would result in an infinitely large library.

Hence, there seems to be only one reasonable strategy: 1) Identifying generic patterns from which the numerous variants can be derived by adaptation and 2) providing modeling support to facilitate the necessary adaptation process. Thus, we will able to produce manageable libraries with broad horizontal cover. In particular, we believe that *externalizing* adaptation is the key factor in component-based development of problem-solving methods. This can be seen by a simple example. Let's assume $n$ search strategies and $m$ problem types. Without adapters $n * m$ components would be necessary to cover all these combinations. The use of adapters means that only $n + m$ components are needed. To make this example more concrete, let's consider local search and design tasks:

- A local search algorithm has four main parameters that determine its search character [48]: the selection of start nodes, the generation of successors nodes, the selection of "more promising" nodes, and the definition of the preference relation. Different values for these parameters distinguish between, e.g., best-first search, hill-climbing, and beam search. Keeping the precise definitions of these parameters external of the core definition of the method makes it possible to provide a large variety of search methods using only a small number of components.
- A *configuration design* task is a design task in which all components are known at the beginning of the design process. Hence, the specification of the class of configuration design tasks can be seen as a *refinement* of the class of design tasks. In turn, the specification of the class of *parametric design* tasks (these are tasks in which the skeletal structure of the target design is given at the beginning of the design process) can be obtained by specializing the class of configuration design tasks.

2. Not surprisingly, this was the experience encountered by many developers of problem-solving method shells.

Defining a component for each variation of a search method and for each possible task-specific (and domain-specific) refinement is clearly an intractable problem. A tractable and structured approach for defining usable and reusable components can only be achieved by separating the adaption process from the specification of the "key generic components."

In the rest of the paper, we will illustrate this approach in detail.

## 3 The Method-Specification Space

We have already pointed out that we view problem solving method development as a process taking place in a three-dimensional space, defined by problem-solving strategies, domain assumptions, and task commitments (see Fig. 2).[3] These three dimensions are described below.

- **Problem-solving strategy.** This is a high-level description which specifies a type of problem solving rather than an actual algorithm, i.e., we describe a entire class of algorithm. A problem-solving strategy fixes some basic data structures, provides an initial task-subtask decomposition and a generic control regime. This generic control regime is meant to be shared by all problem-solving methods which subscribe to the same problem solving strategy. Examples of problem solving strategies are: *Generate & Test*, *Local Search*, and *Problem Reduction* ([78]).
- **Domain knowledge assumptions.** These are assumptions on the domain knowledge that is required to instantiate a problem-solving method in a particular application. These assumptions specify the types and the properties of the knowledge structures which need to be provided by a domain model, in addition to those required to fulfil task-specific commitments. For instance, when solving a design problem by means of Propose & Revise, a domain needs to provide the knowledge required to express *procedures* and *fixes*, in addition to the task-related knowledge needed to formulate the specific design

3. Of course, like all analogies, the analogy with 3D space must not be taken to the extreme. Clearly, not all possible points in the space defined in Fig. 2 are neccessarily meaningful or reachable. Nevertheless, the figure still accurately describes the space of possibilities introduced by our framework.

problem, e.g., parts and constraints. Domain assumptions are necessary to enable efficient problem solving for complex and intractable problems ([40], [31]). More in general, the reliance on such domain-specific knowledge is the defining feature of knowledge-intensive approaches to problem solving.

- **Problem (i.e., task) commitments.** These specify ontological commitments to the type of problem that is solved by the problem-solving method. These commitments are expressed by subscribing to a particular *task ontology*. For instance, a parametric design task ontology provides definitions for terms such as design model, parameter, and constraint—see [59] for a detailed specification of a task ontology. The ontological commitments introduced by a task can be used to refine the competence of a problem-solving method, the structure of its computational state, and the nature of the state transitions it can execute (cf. [25], [33], [34]). For instance, a generic search method can thus be transformed into a specialized method for model-based diagnosis or parametric design. Such a task-specific refinement still produces a reusable method specification given that this is formulated independently of a particular application domain. A diagnostic problem solving method may be formulated in terms which are specific to diagnostic problem solving, but it can be reused in different technical or medical diagnostic applications. The advantage of refining problem-solving methods in a task-specific way is that the resulting model provides much stronger support for knowledge acquisition and application development than a task-independent one, i.e., the method becomes more *usable*.

Fig. 2 visualizes the three dimensions of our problem solving method space by means of arrows. Although this representation may be taken to imply that each dimension is characterized by a total order, this is not actually the case. Different tasks, such as diagnosis or design, and different problem solving schemes, such as local search or search by pruning (e.g., branch and bound), may not be derivable from each other. However, they can be derived from more abstract definitions. Hence, each dimension is defined by an acyclic graph. The graph is defined by the refinement relationship between the elements of the design space and reflects the partial order defined by refinements. Having said so, in this paper, we will focus only on one type of tasks (design) and one type of problem-solving scheme (local search) and therefore this graph collapses into a total ordered one.

A clear identification and separation of problem-solving strategy, problem commitments, and domain assumptions enables a principled way to developing a problem-solving method and structuring libraries of problem-solving methods. Current approaches usually merge these different aspects, thus limiting the possibilities for reuse, obscuring the nature of the methods, and making it difficult to reconstruct the process which led to a particular specification. In our approach, the development and adaptation of problem-solving methods is characterized as a navigation process in this three-dimensional space. Moves through the three-dimensional space are represented by means of adapters. In what follows, we will present a detailed example illustrating our view of method development through adapter-mediated navigation in a three-dimensional space. In Section 4, we will then generalize from the particular example shown here and we will provide a generic typology of adapters.

Specifically, in the next sections, we will illustrate the following method-development process—see Fig. 3.

**First**, we introduce a definition of a problem solving strategy. We start with defining a generic search scheme doing a step into the problem solving strategy dimension and specialize the generic search scheme to a local search one (i.e., we refine along the problem solving strategy dimension).

**Second**, we introduce a problem (i.e., task definition). We define a generic optimization problem taking a step into the problem commitments dimension and we refine our definition of a generic optimization problem in two steps to produce a design problem and a parametric design problem (i.e., we add problem commitments).

**Third**, we link problem definitions and problem solving strategies in three steps:

1. First, we refine local search so that it guarantees to find *locally optimal* states.
2. Then, we introduce a domain assumption to ensure that the method is able to find *globally optimal* states a). In the discussion, we will illustrate the importance of introducing domain assumptions by showing that failing to do this results in an incorrect specification. We also discuss why a transition without assumptions on domain knowledge b) may miss the gist of problem-solving methods.
3. We specialize the globally optimal search-based problem-solving model scheme for optimal parametric design problems.

**Fourth**, we refine the optimal parametric design problem solver and produce a Propose & Revise problem solving method, which is configured for parametric design tasks. This step is carried out by differentiating between alternative successor relationships.[4]

Finally, we show how the proposed method development process can be used to perform a "rational reconstruction" of a comprehensive library of problem-solving methods [59], [61].

4. The refinement to Propose & Revise does not add any new principled design path to our example. However, it links it to a well-known method.

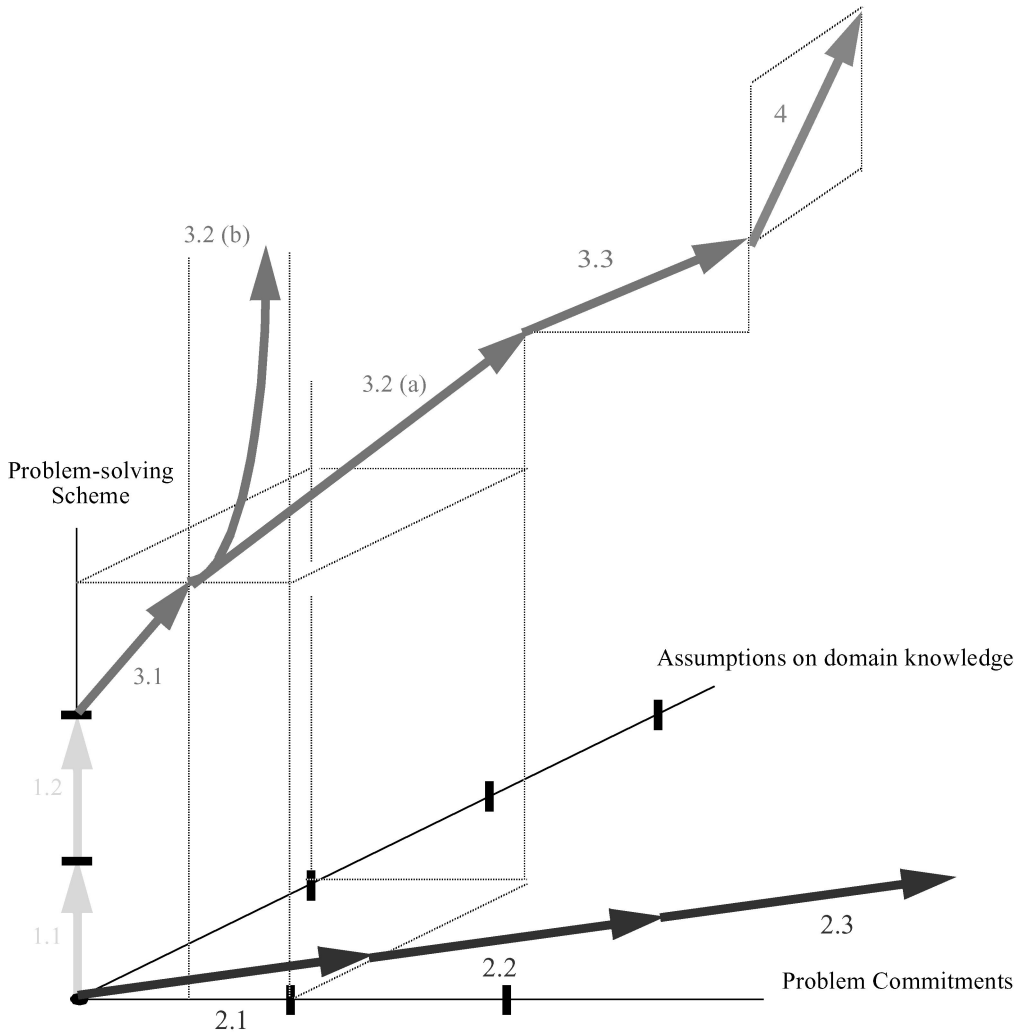


Fig. 3. Illustrative moves in the method specification space.

### 3.1 Local Search

Fig. 4 provides a formal definition of a generic search scheme.[5] Our formalization combines pseudocode to express state transitions with first-order logic to define the functionality of subcomponents and properties of states. This is done in the style of dynamic logic (cf. [49])—see [37], [35] for more details. The generic search scheme consists of four elementary state transitions, i.e., inferences. *Initialize* starts the search process and the sequence of *Select Node*, *Derive Successor Nodes*, and *Update Nodes* recursively navigates a search space until a solution has been found. The signature of the method is based on the notion of object and includes the definition of three constants,[6] *Node*, *Nodes*, and *Successor Nodes*, and a predicate, *Stop criterion*. The latter specifies a halting condition; the constants respectively define the currently selected node (*Node*), the pool of currently available nodes (*Nodes*), and the nodes generated at each cycle of the search process (*Successor Nodes*). The scheme in Fig. 4 defines a *parametrized problem solving template*, which can be instantiated in several different ways to provide a specific search method. The template comprises six parameters: the four elementary inferences, the predicate *Stop criterion*, and the sort *Object*.

Alternative search methods can be generated by refining the inference *Update Nodes* in the generic search scheme shown in Fig. 4. For instance, if the output is obtained by merging *Successor Nodes* with the original input *Nodes*, a variant of *best-first search* is performed, where all expanded nodes are available when selecting the next node. This may be the basis for defining an A* search strategy [66]. If only the newly generated nodes are returned, then a variant of *hill-climbing* is performed—i.e., we derive a model of a *local search scheme*. This is shown in Fig. 5.

### 3.2 Parametric Design Task

Fig. 6 provides a generic definition of a global optimization problem. Its goal statement defines a solution as an object which is optimal with respect to a preference relation. The latter is required to define a partial order and introduces assumptions on domain knowledge. A domain must provide such a preference relation to ensure that the task is well-defined for it.

5. The schema is typical of search strategies which expand the search space. Other types of search, such as brand & bound, which try to *restrict* the search space, would exhibit different schemas.

6. Constants may have different values in different states according to the multiple-world semantics of our specification appraoch. Hence, they are not logical constants. They rather correspond to a constant "address" (i.e., a name) of a storage cell with changing content according to the current state of computation. For more details, see [37], [35].

```
scheme generic search
    control flow
        Search()
            /* Initializes and starts the search. */
            Nodes:= Initialize();
            Output := RecursiveSearch(Nodes)
        RecursiveSearch(Nodes)
            Node := Select Node(Nodes);
            Successor Nodes := Derive Successor Nodes(Node);
            Nodes := Update Nodes(Nodes, Successor Nodes);
            IF Stop criterion(Node, Nodes)
                THEN RETURN Node
                ELSE RecursiveSearch(Nodes)
            ENDIF
    terminology
        sorts    functions
            Object, Node : Object;
            Objects : set of Object;Nodes, Output,
        predicates Successor Nodes : Objects;
            Stop criterion : Object x Objects,
     elementary inferences
          Derive Successor Nodes;
          Initialize
                Initialize() ≠ ∅;
          Select Node
                Select Node(x) ∈ x;
          Update Nodes
                ∅ ≠ Update Nodes(x,y) ⊆ x ∪ y
endscheme
```

Fig. 4. A generic search schema.

Design can be characterized in generic terms as the process of constructing artifacts. Usually, an artifact has to fulfill certain requirements, should not violate certain constraints, and should follow the principle of economy, i.e., should have minimal cost, (cf. [58], [18], [60]). Fig. 7 shows a simplified characterization of the class of design problems, which ignores the distinction between requirements and constraints.

The definition in Fig. 7 refines three aspects of task *global optimum*. First, a *Solution* is now required to be valid, as well as optimal. Second, the generic name *Object* has been replaced with the specific name *Design Model*, in accordance with the terminology used in design tasks.[7] Third, the preference relation is now defined in terms of the cost function; that is, optimal designs are those which minimize the design cost.

The definition of task design given in Fig. 7 can be specialized for parametric design problems by introducing the notions of *parameters* and *value ranges*, as shown in Fig. 8. Parametric design problems reduce the complexity of the design task by assuming the existence of a parametrized solution template for the target artifact. Hence, as shown in Fig. 8, a design model is now defined as a partial function from parameters to value ranges and a solution is defined as a valid and *complete* design model. This is a model in which all parameters are bound and no constraint is violated. The already mentioned VT elevator design problem ([56], [73]) provides a well-known example of a parametric design task. Here, the problem is to configure an elevator in accordance with the given requirements specification and the applicable constraints. The parametrized solution template consists of 199 design parameters which specify the various structural and functional aspects of an elevator, e.g., number of doors, speed, load, etc.

7. When refining the definition given in Fig. 7 to specify parametric design tasks, we will substantiate this terminological change by providing axioms which describe the logical structure of a design model.

```
adapter local search
    import generic search
    export local search
    axioms
        Successor Nodes = Update Nodes(Nodes, Successor Nodes)
endadapter
```

Fig. 5. The refinement to local search.

```
task global optimum
    sorts Object;
    predicates
        Solution : Object,
        optimal : Object,
        < : Object x Object;
    goal
        Solution(x) ↔ optimal(x)
        optimal(x) ↔ ¬ ∃ y (x < y)
    requirements
        /* The relation < defines a partial order*/
        ¬(x < x)[1]
        x < y ∧ y < z →x < z
endtask
```

Fig. 6. The task *global optimum*.

The definition given in Fig. 8 provides two main refinements of our specification of task design: 1) it introduces the additional requirement that a solution should be complete and 2) fleshes out the notion of design model which had been introduced purely as lexicon in Fig. 7.

### 3.3 Applying Problem-Solving Methods to Tasks

In this section, we show how the local search method defined in Section 3.1 can be adapted for the class of optimization problems defined in Section 3.2. This adaption will be carried out in three steps. First, we will ensure that our method finds local optima. That is, we formally establish a competence for the method. Up to now, we had only characterized its operational behavior, in terms of states, elementary inferences, and control flow. Second, we will show how we can bridge the gap between a method which can only guarantee local optimality and a task specification which requires global optimality. In particular, we will show that this adaption can be carried out according to two alternative strategies. The final adaptation to parametric design can be achieved via simple adaptation.

#### 3.3.1 A Locally Optimal Method

The local search method defined in Fig. 5 does not consider any preference relation. Hence, it cannot reason about

**adapter** Design
**import** *global optimum*
**export** *design*
**sorts**
*Design Model*, *Constraint*, *Cost*, *Constraints* : set of Constraint;
**functions**
*violated* : *Design Model* → *Constraints*;
*cost* : *Design Model* → *Cost*;
**predicates**
*valid*, *solution*: *Design Model*;
**rename**
*Object* : *Design Model*;
**axioms**
$Solution_{\text{Export}}(x) :\leftrightarrow Solution_{\text{Import}}(x) \wedge valid(x)$
$valid(x) :\leftrightarrow violated(x) = \emptyset$
$x < y :\leftrightarrow (cost(y) < cost(x) \wedge valid(y))$
**endadapter**

Fig. 7. A definition of the generic design task.

**adapter** Parametric Design
**import** *design*
**export** *parametric design*
**sorts**
$Parameter = \{p_1,..., p_n\}$, *Parameters* : set of *Parameter*,
$ValueRange_1, ..., ValueRange_n$;
**functions**
*Parametric Design Model* : *Parameter* → $ValueRange_1 \cup ... \cup ValueRange_n$
where *Parametric Design Model*(*pi*) ∈ $ValueRange_i$ for all $i$=1,..,$n$;
*assigned* : *Parametric Design Model* → *Parameters*;
**predicates**
*complete* : *Parametric Design Model*;
**rename** *Design Model* : *Parametric Design Model*;
**axioms**
$SolutionExport(x) \leftrightarrow SolutionImport(x) \wedge complete(x)$;
$complete(x) \leftrightarrow \forall\ y\ (y \in assigned(x))$;
$x <_{\text{Export}} y :\leftrightarrow (x <_{\text{Import}} y \wedge complete(y))$
**endadapter**

Fig. 8. The task parametric design.

optimality, let alone guarantee it. Fig. 9 adds the necessary requirements for such a proof. First, we import the definition of task *global optimum*, thus acquiring the conceptual machinery needed to talk about optimality. Second, we introduce three requirements (in the form of axioms): 1) the method stops if and only if none of the newly generated nodes is better than the currently selected one—i.e., if the current node is *locally optimal*, 2) *Select* always chooses the best node from its input set, and 3) the quality of the selected node improves monotonically. The latter requirement is specified by stating that at least one element of *Successor Nodes* is better or equal to the best element in *Nodes*, where "better" is determined in terms of the partial order specified by the preference relation. Having introduced these three axioms, we can then prove that our search procedure behaves like hill-climbing, i.e., it monotonically converges to a local optimum.[8]

### 3.3.2 Establish the Competence to Find Global Optima

There is still a significant gap between a locally optimal method and the task requirements for global optimality. In principle, there are two different strategies that can be taken to bridge such a gap. We could modify the control regime of the method, so that it navigates the entire

8. Of course, such local optimum is not guarenteed to be global because the hill-climbing process might get stuck into local maxima, Notice also that *Select Node* and *Update Nodes* have to solve a global optimum problem. They must provide the best node and the best successor. Therefore, the problem of finding a global optimum reappears as subproblem, however, reduced for a subset of all nodes. This divide & conquer stragety is common for most local search strategies. Again, its success depends on further assumptions on the successor relationship.

**adapter local search finding local optima**
**import** *local search, global optimum*
**export** *local search finding local optima*
**competence**
$\neg \exists x\ (x \in Nodes \cup Successor\ Nodes) \wedge Output < x)$
**axioms**
(1) $Stop\ criterion(x, Y) :\leftrightarrow \neg \exists y\ (y \in Y \wedge x < y\ )$
(2) $\neg \exists x,y\ (x \in y \wedge Select\ Node(y) < x)$
(3) $\neg \exists x,y\ (x \in Update\ Nodes(Nodes,Successor\ Nodes) \wedge y \in Nodes \wedge x< y)$
**endadapter**

Fig. 9. A locally optimal method.

search space. If it is finite, then this approach is guaranteed to find a global optimum. Such strategy is represented by arrow 3.2 (b) in Fig. 3. Unfortunately, this approach is normally unfeasible. AI problems are typically very complex and not amenable to "brute force" approaches (cf. [15]).

An alternative strategy is to introduce *assumptions* on domain knowledge that make it possible to bridge the gap between the competence of a method and the goal of the relevant task—see arrow 3.2 (a) in Fig. 3. That is, parts of the problem-solving process can be delegated from the method to the domain knowledge. The problem-solving method defines a reasoning strategy on top of the domain knowledge to ensure that this knowledge is used in a proper and efficient way to support problem solving. An assumption can be introduced either to strengthen the competence of the combined problem solver method plus domain knowledge, or to weaken the task that can be performed by it. In the latter case, it describes the restrictions under which dependable problem solving can be guaranteed (cf. [31]).

For our example, we have to establish a necessary property of the domain knowledge used by *Successor Nodes*. The successor relationship must always provide a better successor in the case that a node is not already a global optimum (see Fig. 10). Such an assumption prevents our local search procedure from getting stuck in local optima.

This kind of assumptions can be verified by means of a technique called *inverse verification, which is* described in [38] and [39].

Of course, the given formulation of the *better-successor* assumption is very generic. However, it is easy to see that concrete refinements of this assumption underlie some well known methods. For instance,

**adapter local search for global optima**
**import** *local search finding local optima, global optimum*
**export** *local search for global optima*
**axioms**
*Solution*(*Output*)
*better-successor assumption*:
$\exists y\ (successor(x,y) \wedge x< y) \vee \neg \exists z\ (x < z)$
**endadapter**

Fig. 10. The assumption for finding global optima.

- the A* search method (cf. [13]) guarantees that it will find an optimal solution to a problem if the application domain supports the specification of an admissible heuristic function. The assumption on the existence of such a heuristic defines a refined version of the *better-successor* assumption.
- Bylander et al. [15] and [23] characterize variants of assumptions that ensure efficient problem-solving for diagnostic tasks. The assumptions they use require that a superset of hypotheses must also explain a superset of observations. These assumptions can be seen as task-specific refinements of the generic *better-successor assumption* (cf. [39] for more details). They define the conditions under which a local search method is able to find an optimal (i.e., parsimonious) diagnosis.

#### 3.3.3 Applying Local but Optimal Search to Design Tasks

Applying the methods to the design tasks defined in Fig. 7 and Fig. 8 only requires importing the relevant logical theories. Hence, we can define two simple adapters, as shown in Fig. 11.

### 3.4 Refining Local but Optimal Search for Parametric Design to Propose & Revise

We will now conclude our sample method development process by showing how our search method can be further refined to derive a generic specification of a Propose & Revise problem solving method for parametric design.

Propose & Revise introduces *differentiation* in the search procedure by distinguishing between two different behaviors of *Derive Successor Nodes* [92], [60], [61], [59]. If the current state is incomplete, then a *design extension step* is carried out, to extend the current design model. If the current state is not valid (i.e., some constraint is violated), then a *revision step* is carried out, to restore consistency. The adapter shown in Fig. 12 defines Propose and Revise inferences in terms of the more generic *Derive Successor Nodes.* It is important to note that the specification given in Fig. 12 is neutral with respect to the sequence of propose and revise steps and allows for alternative variants of Propose & Revise to be defined by introducing different control regimes. For instance, one may apply propose steps until a model is complete, and then revise. Alternatively, one may revise a model as soon as an inconsistency arises (cf. [27], [92]). These two control regimes are formalized in

**adapter** Local Search for Design
**import** *local search for global optima, design*
**export** *local search for design*
**endadapter**

**adapter** Local Search for Parametric Design
**import** *local search for design, parametric design*
**export** *local search for parametric design*
**endadapter**

Fig. 11. Local but optimal search for design and parametric design.

**adapter** Propose&Revise for Parametric Design
**import** *local search for parametric design*
**export** *Propose&Revise for parametric design*
**functions**
*Partial completeness* : *Design Model* → *Parameters*;
**axioms**

/* Propose steps extend incomplete states. */
($y \in$ Propose (x)) →
$\neg$ *Complete*($x$) $\wedge$ $y \in$ Derive Successor Nodes (x) $\wedge$
Partial completeness(x) < Partial completeness(y)

/* Revise steps correct inconsistent states but do not change the degree of completeness. */
($y \in$ Revise (x)) →
$\neg$ *valid*($x$) $\wedge$ $y \in$ Derive Successor Nodes (x) $\wedge$ *valid*($y$) $\wedge$
Partial completeness(x) = Partial completeness(y))
**endadapter**

Fig. 12. A generic characterization of Propose & Revise.

/* Control regime #1: revise only after completing the model */ ($y \in$ Revise (x)) $\leftrightarrow$ *Complete*($x$)
**while not** complete
**do**
*propose*
**end do;**
**if not** valid
**then**
revise
**end if**

/* Control regime #2: revise as soon as an inconsistency arises*/ ($y \in$ Propose (x)) $\leftrightarrow$ *valid*($x$)
**while not** complete **or not** valid
**do**
**if not** valid
**then**
revise
**end if**
**if not** complete **and** valid
**then**
propose
**end if**
**end do**

Fig. 13. Alternative control regimes for Propose & Revise.

Fig. 13. Each control regime is defined by introducing the relevant axiom and the associated control flow.

### 3.5 Summary

Fig. 14 summarizes the method development process described in the previous sections. On the left side, we refine the problem definition and on the right side we refine the problem-solving strategy that guides the problem-solving process. The virtual elements do not require explicit specifications because these follow from the combination of an existing specification and an adapter. However, for convenience, a library may also directly provide these derived specifications.

The approach described in this paper has been used to perform a "rational reconstruction" of the library of problem solving components described in [59], [61]. This library provides a comprehensive set of components for parametric design problem solving and has been used in several real-world applications, including sliding bearing design, initial vehicle design, and the design of casting technology for manufacturing mechanical parts [85].

All the problem solving methods in the library by Motta and Zdrahal are defined as specializations of a generic search-based model of parametric design problem solving. This model specifies a rather complex three step procedure for carrying out *Derive Successor Nodes*. First, a *design context* is abstracted from the current node, then a *design focus* is derived within the context and, finally, a *transformation operator* is derived from the design focus—see Fig. 15. The use of the three-step selection process is motivated by the structure of parametric design problems:

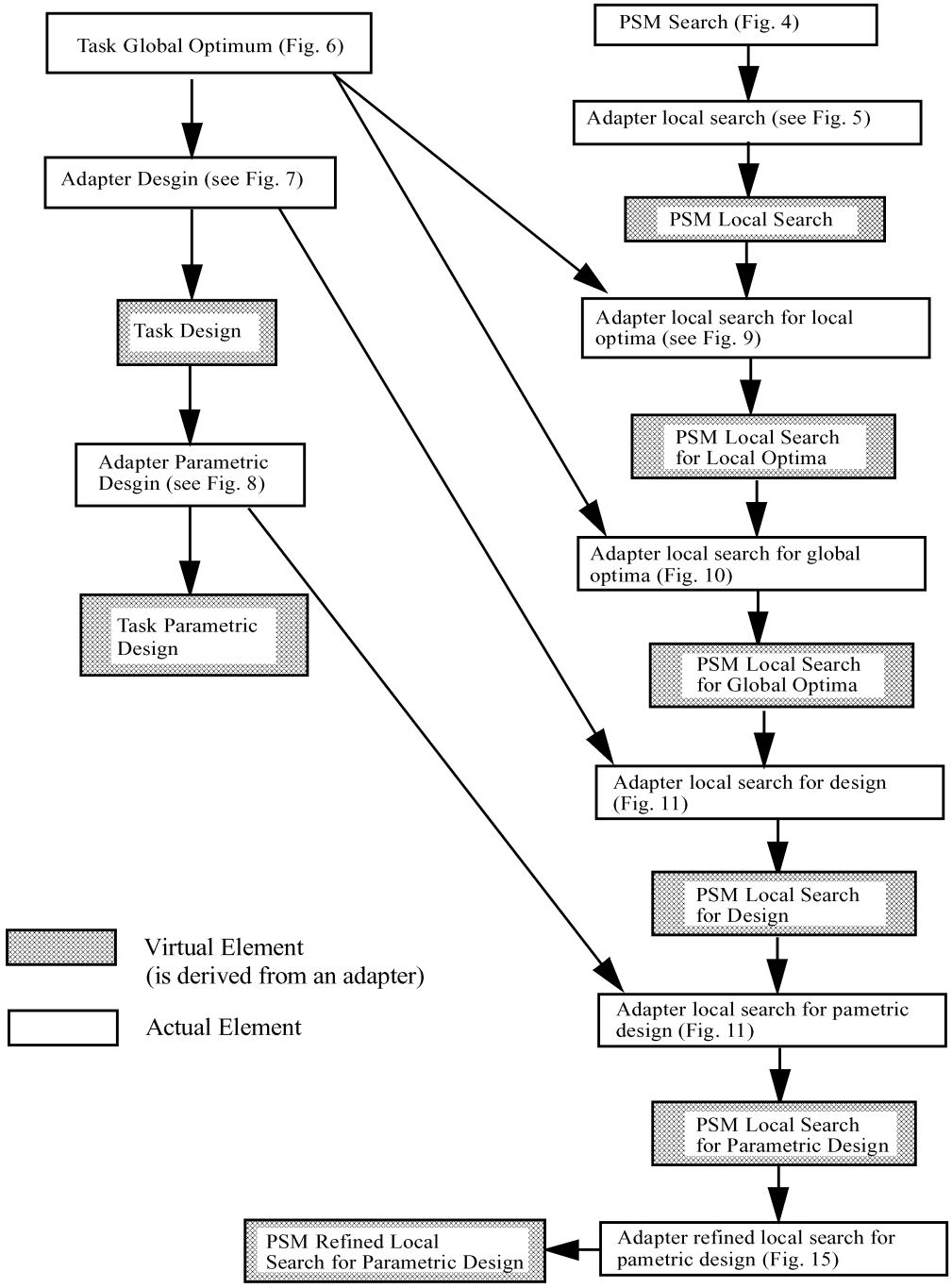


Fig. 14. The development graph of a problem-solving method for parametric design.

- The quality information associated with a parametric design model distinguishes four criteria: violation of constraints, fulfillment of requirements, completeness of value assignments, and costs. Four different *contexts* can be immediately identified according to these criteria: trying to repair constraint violations, trying to improve fulfillment, trying to improve completeness, and trying to reduce costs.[9]

9. As the reader may have realized, a more pedantic organization would have already introduced three of these context decisions when specifying (generic) design problems, given that these decisions are not specific to parametriic design. For reasons of simplicity, we have skipped this intermediate step.

- Within a context, we can decide about the focus of the design activity by using the object information, in this case, the structure of the parametric design model. Specifically, a parametric design model can be functionally characterized in terms of the relevant violated constraints, nonfulfilled requirements, and unassigned parameters. Thus, choosing a focus consists of selecting one of these elements, in accordance with the current context—e.g., choosing the appropriate constraint violation when the current context is one of design revision.
- Design is about applying transformations/extensions to a design model. In general, several

```
adapter refined local search for parametric design
    import local search for parametric design;
    export refined local search for parametric design;
    sorts
        Context, Focus, Transformation,
        History : set of (Context x Focus x Transformation);
    Program
        Derive Successor Nodes(Design Model)
            Context := Decide Over Context(Design Model);
            Focus := Decide Over Focus(Design Model, Context);
            Transformation := Decide Over Transformation(Design Model, Context, Focus);
            Bookmark Context, Focus, and Transformation in History of Design Model;
            Design Models := Apply Transformation(Design Model, Transformation);
            RETURN Design Models
endadapter
```

Fig. 15. Refining *Derive Successor Nodes*.

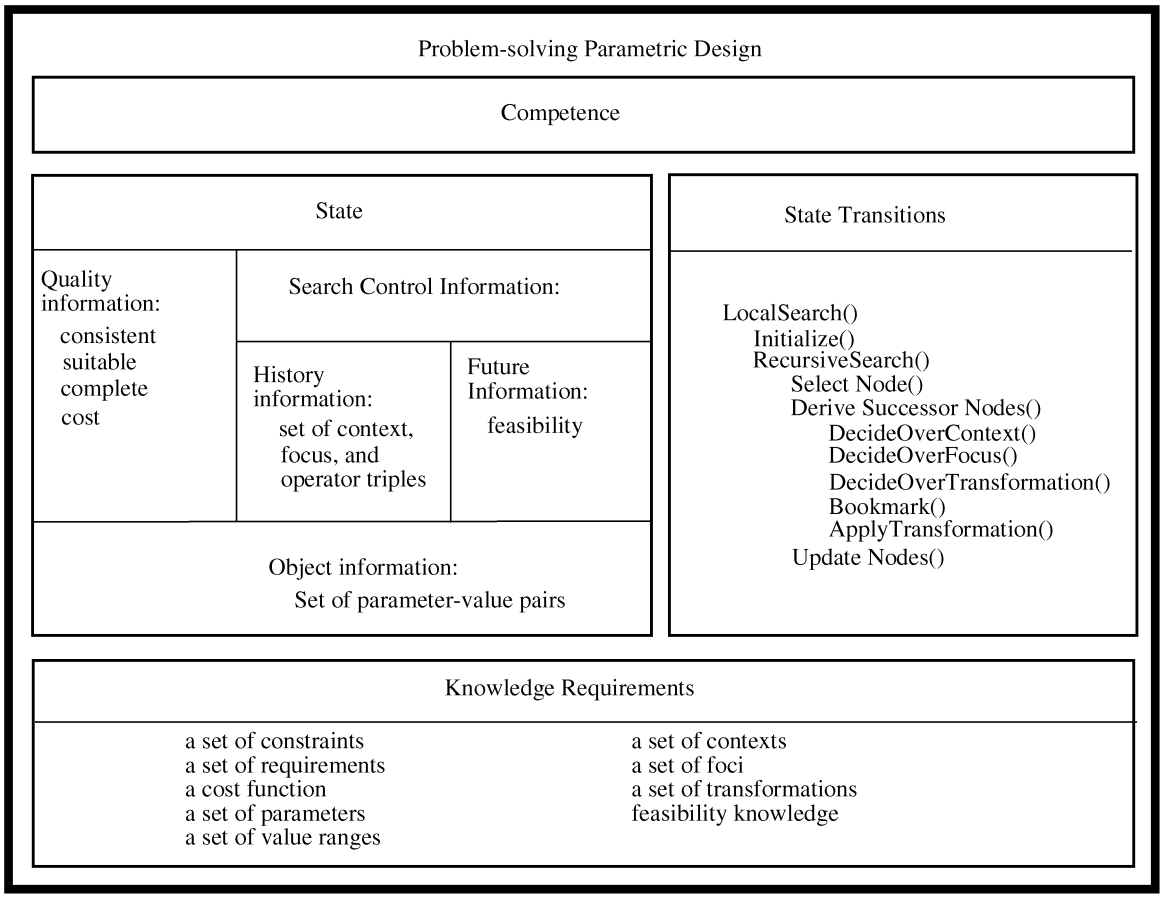


Fig. 16. A survey of parametric design problem solving.

transformations/extensions may be applicable for a given context and focus and, therefore, additional operator selection knowledge may be required.

A synoptic view of the generic model of parametric design problem solving underlying the library by Motta and Zdrahal is presented in Fig. 16. This style of specification, which abstracts from several dozens of definitions in the original library by Motta and Zdrahal, results from the approach adopted here, which consists of choosing an algorithmic scheme and applying several adapters that refine its state descriptions and state transitions. Such a specification is of course more abstract than what is usually called a problem-solving method in the literature. Its advantage is that it concisely captures a family of specifications. For instance, a Propose & Revise method for parametric design can be specified by distinguishing two contexts when deriving successor design models (cf. Section 3.4): a *propose* context, in which design models are extended, and a *revise* context, in which design models are repaired [61], [59]. In other words, this problem-solving method can be derived from our framework by simply refining some of its parameters.

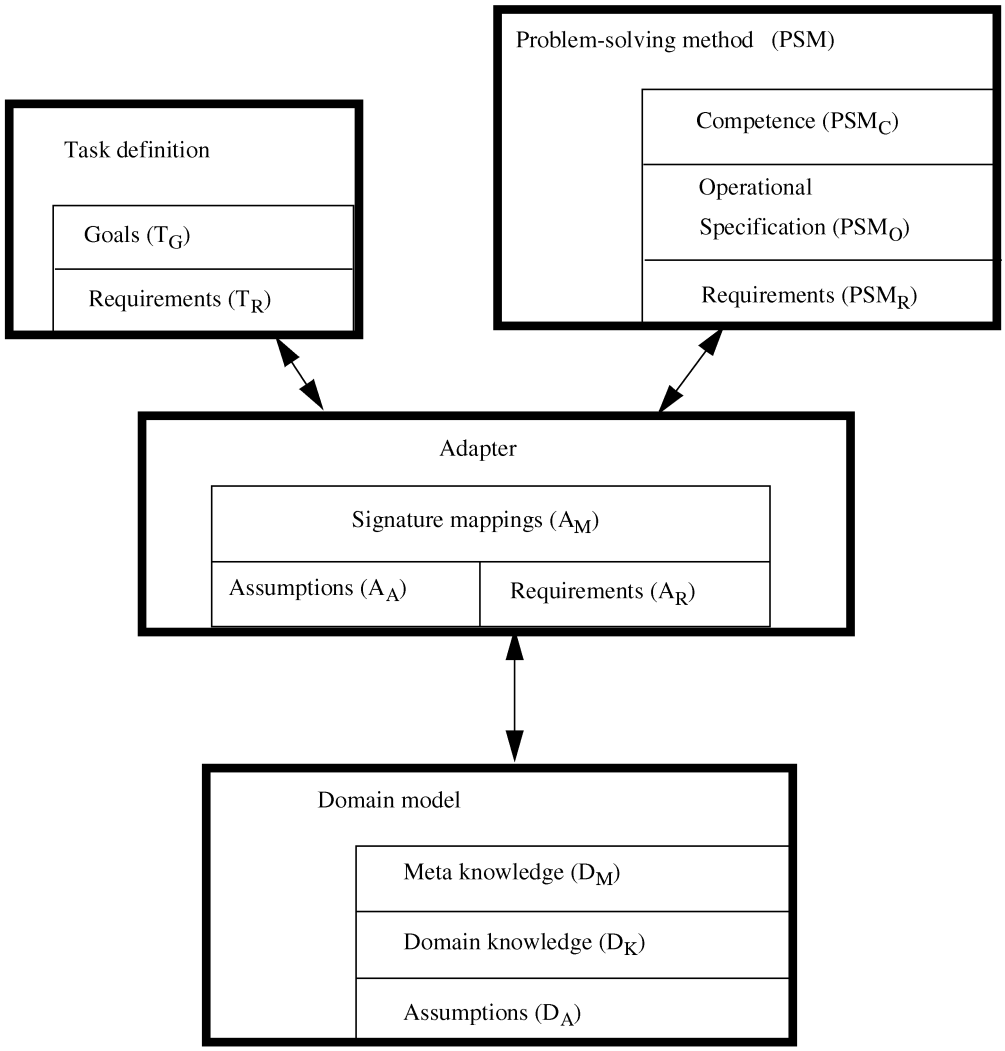


Fig. 17. The four component architectures for knowledge-based systems.

## 4 Transitions in the Method Specification Space

The integration of preexisting components into larger systems is an important problem in computer science. This issue reflects the fact that as systems become larger and more complex, they are no longer built from scratch, but are instead configured out of preexisting building blocks. As a result, there is currently much interest in software mechanisms which can be used to interconnect components and adapt them for specific applications. The *adapter* pattern is one of several *design patterns* in the textbook by [43], which deals with object-oriented integration. Adapters are also present in most approaches to software architectures ([77], [91]), where they are often called *connectors*. Their main purpose is to integrate components which exhibit different interaction styles. Another type of adapters, called *wrapper* and *mediators*, have been proposed to enable sharing and reuse of heterogeneous and distributed information and knowledge sources ([88], [89]).

While these mechanisms have been designed in different subfields of computer science and differ in a number of details, they all play the same role: they make it possible to adapt a component to a new context, thus enabling its reuse and *externalizing* the adaptation mechanism.

Fensel and Groenboom [37] introduced *adapters* into the CommonKADS model of expertise as a generalization of the mappings between domain and inference layers. Adapters make it possible to specify tasks, problem-solving methods, and domain knowledge independently of each other, thus enabling their reuse (cf. Fig. 17).[10] Fensel [29] generalized the use of adapters to stepwise adaptation of problem-solving methods, tasks, and assumptions via a *pile* of adapters. In this approach, the adapter itself becomes reusable, given that it is used to adapt components which are themselves generic and reusable (e.g., a generic task). This situation is specific for adapters used in knowledge engineering and reflect the fact that only this community has been developing generic description of problem types ([12], [10]).

Still, our current adapter concept may be too general and a typology of adapters may significantly facilitate its usability. So far, we have identified one basic viewpoint for organizing adapters: in terms of their purpose (*teleological* aspect). This

10. Adapters correspond conceptually to the *transformation operators* of [86].

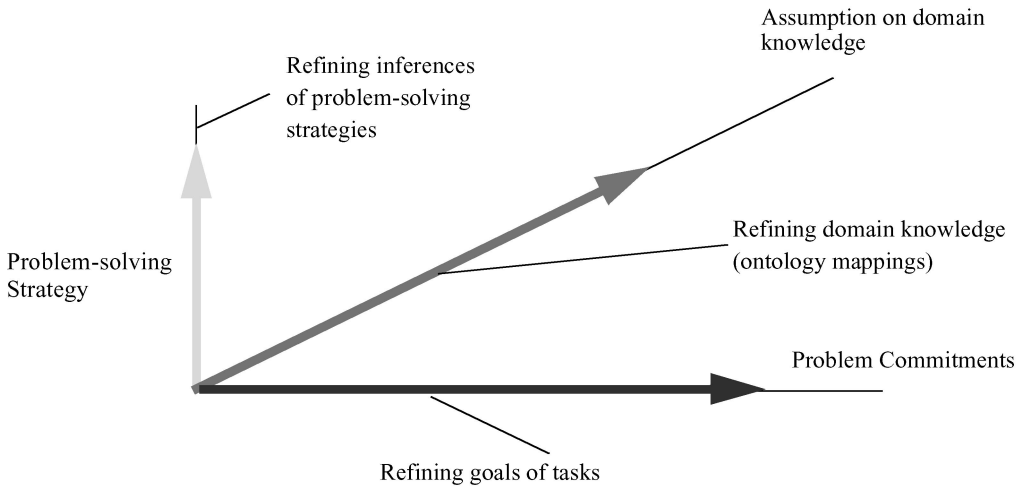


Fig. 18. One-dimensional moves in the problem space.

viewpoint indicates whether an adapter is used to refine a problem type, a domain dependency via assumptions, or a problem-solving strategy. In the following, we will work out this idea in more detail and show how we can transform the development process of problem-solving methods into a more structured engineering activity.

In Section 3, we introduced a three-dimensional method-specification space (see Fig. 2) which we navigate by means of adapters. The dimensions of the space are the problem-solving strategy, the problem (i.e., task) commitments, and the domain knowledge assumptions. Here, the idea is to use these three dimensions as the basis for producing a typology of adaptors. Specifically, moves in this space can be distinguished in terms of their *orientation*:

- A movement may occur within only one of the three main axes. This can be either a refinement of the problem-solving strategy, or a refinement of a domain assumption or a refinement of a problem commitment.
- A movement may occur along the plane specified by any two dimensions. That is, a move may refine a problem-solving strategy by introducing domain assumptions, may connect a problem-solving strategy to a problem type, or refine a problem type in terms of domain assumptions.
- A movement may affect all three dimensions. This move exhibits maximal complexity and, therefore, we will show that it can be decomposed in terms of two-dimensional moves.

In the following section, we will briefly sketch the different adapter types.

## 4.1 One-Dimensional Moves

In the following sections, we investigate adapters that provide moves in a one-dimensional subspace; i.e., adapters which either refine elementary inferences of problem-solving strategies, or introduce/refine assumptions on domain knowledge, or introduce/refine goals of problem definitions (see Fig. 18).

### 4.1.1 Refining a Problem-Solving Paradigm

Problem-solving methods can be viewed as components with an *active* interface. Each inference defines a parameter that can be further refined externally. Usually, a number of axioms are added which strengthen the inference. Examples are provided in Section 3.1. This type of adaptation occurs within one dimension, i.e., it follows the problem-solving scheme line. However, this is the case only if the refinement of inferences is not accompanied by the introduction of new requirements and assumptions on domain knowledge. In the latter case, the move in question occurs within a two-dimensional subspace (cf. Section 4.2.1).

### 4.1.2 Refining Domain Knowledge

The situation in which domain knowledge is refined independently of its use for problem-solving schemes and task definitions does not occur in our framework, where adaption is driven by problem-solving method development and reuse. Nevertheless, these types of refinements can occur in situations where it is necessary to integrate different domain ontologies and can be handled by means of mediator-type approaches [88], [89]. An example for such a move is the refinement of a generic upper-level ontology in a certain domain; for example, the domain-specific instantiation of a generic definition of the part-of relation.

### 4.1.3 Refining the Goal of a Task

All examples of task refinement shown in Section 3 also introduce new requirements on domain knowledge. An optimization task requires a preference function, a design task requires constraints, a cost function, etc. Hence, these refinements affect both the domain and the task dimensions. Nevertheless, it is possible to imagine task adaption moves which only affect this dimension. A simple example is the case in which we ask for all solutions to a problem, rather than only one.

## 4.2 Two-Dimensional Moves

In this section, we discuss adapters that provide moves within two-dimensional subspaces: refining domain

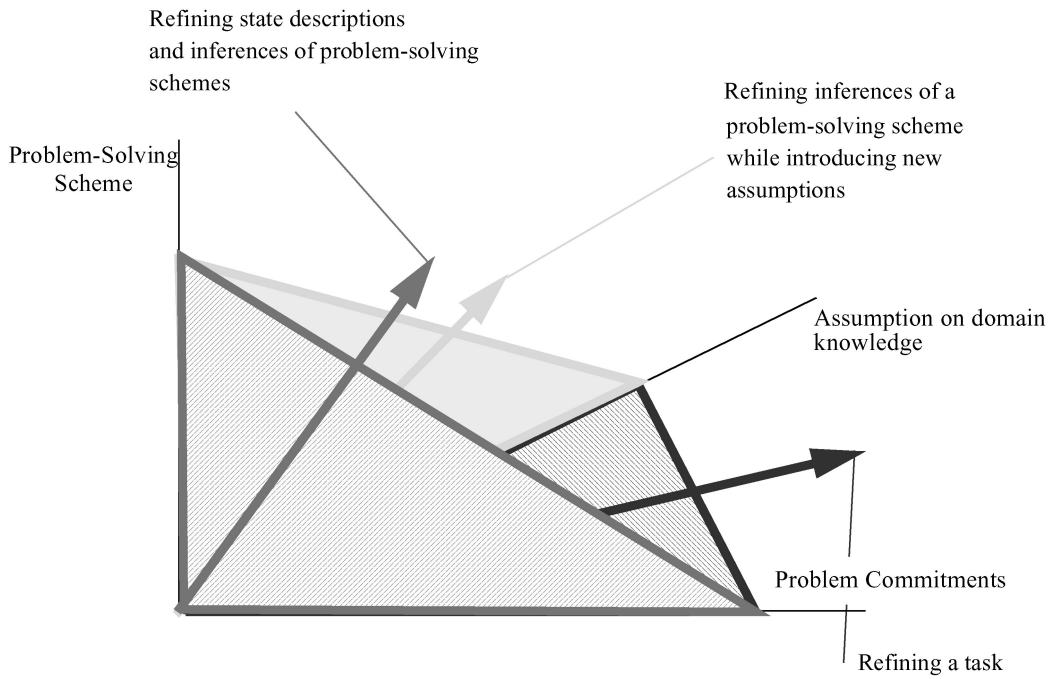


Fig. 19. Two-dimensional moves in the problem space.

assumptions of problem-solving strategies or tasks, or applying problem solving schemes to tasks (see Fig. 19).

#### 4.2.1 Refining Problem-Solving Inferences while Introducing Domain Assumptions

Section 3.4 discusses the refinement of local search for parametric design to Propose & Revise for parametric design. The inference *Derive Successor Nodes* is specialized by distinguishing between two types of successor derivation steps: those which extend completeness (propose) and those which fix inconsistencies (revise). Hence, the adapter shown in Fig. 12 provides an example of refining a problem solving inference. At the same time, because additional types of knowledge are required by the resulting Propose & Revise method (knowledge about design extension and revision and a relative completeness measurement), then this adapter also denotes a move along the domain assumption axis. However, no change occurs along the task dimension: The newly generated method is designed to solve exactly the same class of problems as the source method.

#### 4.2.2 Adapting Problem-Solving Strategies to Problem Types

Sections 3.3.1 and 3.3.3 describe simple adaptations of problem-solving schemes to different tasks, which do not introduce new assumptions on domain knowledge.

#### 4.2.3 Refining a Task

A task is refined by strengthening its goal and knowledge requirements. Section 3.2 provides examples of these refinements (from a generic optimization problem, through design, to parametric design). In these steps, we refined the definition of the goal and introduced new requirements on domain knowledge. A solution to an optimization problem is defined in terms of an optimality criterion; a solution to a design problem must also be valid; a solution to a parametric design problem must be valid and complete. Because these refinements introduce new requirements on domain knowledge, the resulting moves occur in a two-dimensional subspace.

### 4.3 Three-Dimensional Moves

Assumptions on domain knowledge are introduced by problem-solving schemes to improve the competence of the method without changing their algorithmic strategy. An example of this approach was shown in Section 3.3.2, where the competence of a local search method was strengthened without actually modifying its incomplete search strategy. This strengthening was achieved by formulating assumptions on the structure of the search graph defined by the domain knowledge. The relevant adapter relates methods to tasks and closes the gap by introducing assumptions on domain knowledge. Therefore, it is necessarily a move affecting all three dimensions of the method specification space. However, we split the move into two steps. First, we have a two-dimensional move into the problem-solving strategy and problem commitment directions. With Section 3.3.1, we refine our search method to a method that search for (local) optima. We enrich the competence of the method with task-specific commitments. Second, we have a two-dimensional move into the problem-solving strategy and domain knowledge assumptions directions. With 3.3.2a we add assumptions on domain knowledge that enrich the competence of the PSMs to finding global optima. *Breaking down three-dimensional moves into a sequence of two two-dimensional moves significantly reduce the complexity of the development process of PSMs.*

We also mentioned an alternative strategy (3.3.2b), which tries to close the gap between task requirement and method competence by modifying the algorithmic scheme (in particular, by searching a larger part of the search space) instead of introducing assumptions on domain knowledge. In this second scenario, we would again perform adaptation only in the same two-dimensional projection of the entire problem space as done with 3.3.2. However, as a consequence, we would have an inefficient problem-solver.

- 3.3.1 and 3.3.2a decompose a three-dimensional move into two two-dimensional ones. Efficiency is achieved by assuming appropriate domain knowledge.
- 3.3.1 and 3.3.2b are a sequence of two two-dimensional moves in the same directions. They do not define an "appropriate" decomposition of a three dimensional move, given that they lead to a problem solver that is intractable in most cases. Enforcing completeness by performing a complete search only works for very small search spaces.

In the case of knowledge-based systems, it is usually necessary to delegate parts of the problem solving to the domain knowledge and to weaken the task specification in order to enable efficient problem solving. Therefore, three-dimensional moves are often unavoidable. However, it turns out that these can always be realized by a sequence of two-dimensional moves. That is, we can first map a method to a task and then introduce assumptions on domain knowledge that close the relevant gaps.

## 5 Related Work, Conclusions, and Future Work

The landmark papers of [20] and [16] introduced the concepts of PSMs and libraries of PSMs into the development process of knowledge-based systems. Since then, many PSMs have been developed but hardly any work exists, which provides guidance on the development process. Exceptions are [2] and [90] which have put forward some general ideas on how such a development process can be guided by engineering principles. In this paper, we have presented a systematic approach, which enables the structured development, adaptation, and reuse of problem-solving methods. We have shown that problem-solving methods can be described by means of 1) a structured set of generic tasks, problem-solving patterns, and generic domain assumptions and 2) adapters formalizing the relevant refinement steps. These are explicitly modeled, thus allowing their reuse for new problem types and different problem-solving schemes. Adapters introduce a new modeling element into existing modeling approaches, which makes the method development process *explicit* and *external* to the model, i.e., separated from problem solving components. In most other approaches, adaptation is treated as a side issue while adaptation via several levels is not considered at all. In our view, the approach proposed here provides a clear foundation for both problem-solving method specification and configuration, and enables the development of well-formed, comprehensive, and reusable libraries. Making adaptation external and explicit formally records the design decisions that have led to the constructed problem solver, thus providing the key to reusability and maintainability. The proposed typology of adapters provides both a theoretical basis and practical guidelines for the method development process. Our approach supports knowledge engineers who want to develop maintainable PSM libraries with a large cover. We provide the means to reduce the number of components to a manageable size and we define a structured approach for specifying which can help others in using the library (i.e., in navigating through the space of possible component combinations). The explicit characterization of moves through the problem-solving method space is important both to formalize the method development process and to ensure that not only problem solving components but also the process of creating them becomes part of a library.

The importance of recording the knowledge engineering experience in a library has been recognized for a number of years ([82], [86]). However, approaches to capturing design expertise in knowledge engineering have mostly centered on recording informal guidelines. For example, while [52] use hyperlinks expressed in natural language to adapt model fragments, they do not provide an explicit representation of the adaptation component itself. The CommonKADS Library [12] has already an embryonic type of adapters, which are called *features*. However, these are used in CommonKADS more as annotations (metacomponents) than as integral parts of a method specification. In other words, they are rather ad-hoc and considered as a side aspect, rather than as a structuring mechanism for a library.

In contrast with these approaches, our proposal makes it possible to record explicitly and formally the complete method development process, thus providing the basis for automatic method configuration. In addition, our approach is based on a theoretically sound framework, which makes it possible to tease out the various types of commitments embedded in a problem solving method, thus clarifying the nature of specific methods and maximizing the possibilities for reuse.

The work presented here can be situated in the general context of program derivation from specifications. However, the existing approaches rely on specific representation and specification languages (functional programming languages [78] or logic-programming [42]) and focus on algorithmic refinement as means to improve the efficiency of programs. Therefore, they are rather complementary to our approach, which relies on a specific *architecture* for specifying knowledge-based systems and focuses on introducing problem commitments and domain knowledge assumptions to improve efficiency and to economize the adaptation of reusable components to new applications. Refining the algorithmic structure of a problem-solving method can be viewed as a design time activity, needed to optimize the runtime efficiency of the target system.

The notion of adapters in our context plays a role similar to that played by *mediators* in heterogeneous information systems [89], *connectors* in software architectures [77], and *adapters* in design patterns [43]. The idea underlying all of these approaches is essentially the same: some kind of "external kit" is required in order to

allow the interaction of reusable components and their configuration for different computational scenarios. The *externalization* of this adaptation process has the advantage that the original components remain unchanged, while they become usable in the new situation. Our use of adapters extends these approaches in three directions: 1) we use adapters as a means to stepwise refine problem-solving methods according to task and domain-specific circumstances, 2) we identify a typology of adapters based on our three-dimensional method specification space, and 3) we characterize adapters so that they can themselves be reused. In particular, an adapter that refines a method for a generic problem definition can also be applicable to other methods. For instance, an adapter that introduces the notion of design problem can be used to refine a local search method as well as a global search method for design problem solving. Thus, having been originally introduced to enable reuse of other components, adapters become reusable too. The UPML language ([36]), which has been developed in the course of the IBROW project,[11] Fensel and Benkamins [32] distinguish two types of adapters: *refiners* that correspond to one-dimensional moves and *bridges* that correspond to two-dimensional moves.

Description of libraries of PSMs in UPML are used by a brokering service for semiautomatic selection and configuration of distributed reasoning services (cf. [7]).

Finally, we should mention the approach of [83], who uses metalevel reasoning to automatically configure problem-solving methods for diagnostic problems. The problem with this approach is its complexity. Automatically configuring an optimal problem solver may have an order of magnitude higher complexity than the problem that is meant to be solved by the problem solver. Therefore, further heuristic restrictions of her approach are required to provide reasonable semiautomatic support as envisaged by the IBROW project. It is clear that decision procedures for limited-rationality problem-solver (problem-solving methods are heuristic problem solvers) can only have themselves limited rationality (cf. [54]).

In conclusion, in this paper, we have presented an approach to the specification of problem solving methods, which addresses the fundamental research issues in this area. In particular, we have characterized problem solving methods in terms of a three-dimensional space defined in terms of domain assumptions, problem-solving strategies, and task commitments. This space provides both a theoretical foundation, by clarifying the epistemology of problem solving methods, and an engineering foundation, by supporting a method development process and a library organization schema. These 1) subsume both task-independent and task-dependent approaches and 2) allow the development of manageable libraries with broad horizontal cover.

11. IBROW started with a prephase under the Fourth European Framework and is now a fully fledged Information Society Technologies (IST) project under the Fifth European Program. This second phase of the project started in Feb. 2000 and is scheduled to finish by Jan. 2003—see URL http://www.swi.psy.uva.nl/projects/IBROW3/home.html for more information.

Of course, this paper is only concerned with specifying our framework and there is still much work to do in order to validate this approach in the knowledge engineering practice. Clearly, the ultimate value of an approach to reuse is whether it works in practice. In the recently started IBROW project, we'll try to answer this question. In particular, we will be looking at nontechnical issues [46] and at simplified models of competence specification to try and facilitate both the library development process and the process of identifying the "right components" in the library. We will also need to develop the appropriate navigational support, as well as an infrastructure to allow users and developers to record informal experiences about the use of the library. We plan to support navigation in a library of PSM components by means of a specialized version of the WebOnto ontology editor/browser [24] and to support library-centered knowledge sharing by customising our existing knowledge management infrastructure [63].

Our preliminary application of this framework to structure multiple libraries of problem solving methods [62] has produced encouraging results, as it appears not only that the proposed framework provides an effective organization for constructing libraries with large horizontal cover, but also that the resulting libraries exhibit the degree of flexibility required to support semiautomated method configuration, i.e. they are not as brittle as traditional, monolithic libraries. However, more work needs to be carried out to show that the resulting libraries do indeed provide better reuse support than current proposals. In the course of the IBROW project, we are collaborating with some of the main groups involved in PSM research, including the KADS group in Amsterdam and the Protege group at Stanford, to further develop and test the framework proposed in this paper. Our future research will tell us whether the combination of a clear theoretical basis and powerful tools can indeed enable widespread reuse of knowledge engineering technology.

## ACKNOWLEDGMENTS

The authors would like to thank Richard Benjamins, Enric Plaza, Rudi Studer, Bob Wielinga, and the anonymous reviewers for helpful discussions and comments.

## REFERENCES

[1] M. Aben, "Formal Methods in Knowledge Engineering," PhD dissertation, Univ. of Amsterdam, 1995.
[2] J.M. Akkermans, B. Wielinga, and A.Th. Schreiber, "Steps in Constructing Problem-Solving Methods," *Knowledge-Acquisition for Knowledge-Based Systems,* N. Aussenac et al., eds., 1993.
[3] D. Allemang and T.E. Rothenfluh, "Acquiring Knowledge of Knowledge Acquisition: A Self-Study of Generic Tasks," *Current Trends in Knowledge Acquisition,* Th. Wetter et al., eds., 1992.
[4] J. Angele, D. Fensel, and R. Studer, "Developing Knowledge-Based Systems with MIKE," *J. Automated Software Eng.,* vol. 5, no. 4, pp. 389-418, 1998.
[5] "Interpretation Models for KADS," *Proc. Second KADS User Meeting (KUM '92),* C. Bauer and W. Karbach, eds., 1992.
[6] V.R. Benjamins, "Problem Solving Methods for Diagnosis and Their Role in Knowledge Acquisition," *Int'l J. Expert Systems: Research and Application,* vol. 8, no. 2, pp. 93–120, 1995.
[7] V.R. Benjamins, B. Wielinga, J. Wielemaker, and D. Fensel, "Brokering Problem-Solving Knowledge at the Internet," *Proc. European Knowledge Acquisition Workshop (EKAW-99),* D. Fensel et al., eds., May 1999.

[8] V.R. Benjamins and D. Fensel, Special issue on problem-solving methods of the *Int'l J. Human-Computer Studies (IJHCS),* vol. 49, no. 4, 1998.
[9] P. Beys, R. Benjamins, and G. van Heijst, "Remedying the Reusability-Usability Trade-Off for Problem-Solving Methods," *Proc. 10th Banff Knowledge Acquisition for Knowledge-Based System Workshop (KAW '96),* 1996.
[10] J. Breuker, "Problems in Indexing Problem Solving Methods," *Proc. 15th Int'l Joint Conf. Artificial Intelligence (IJCAI-97),* 1997.
[11] J. Breuker, B. Wielinga, M. van Someren, R. de Hoog, G. Schreiber, P. de Greef, B. Bredeweg, J. Wielemaker, and J.-P. Billault, *Model-Driven Knowledge Acquisition: Interpretation Models,* Esprit Project 1098, 1987.
[12] *The CommonKADS Library for Expertise Modeling,* J. Breuker and W. Van de Velde, eds. Amsterdam, The Netherlands: IOS Press, 1994.
[13] *Catalogue of Artificial Intelligence Techniques,* A. Bundy, ed., third ed. Berlin: Springer-Verlag, 1990.
[14] T. Bylander and B. Chandrasekaran, "Generic Tasks in Knowledge-Based Reasoning: The Right Level of Abstraction for Knowledge Acquisition," *Knowledge Acquisition for Knowledge-Based Systems,* B. Gaines et al., eds., vol I, pp. 65–77, 1988.
[15] T. Bylander, D. Allemang, M.C. Tanner, and J.R. Josephson, "The Computational Complexity of Abduction," *Artificial Intelligence,* vol. 49, pp. 25–60, 1991.
[16] B. Chandrasekaran, "Generic Tasks in Knowledge-Based Reasoning: High-level Building Blocks for Expert System Design," *IEEE Expert,* vol. 1, no. 3, pp. 23–30, 1986.
[17] B. Chandrasekaran, "Towards a Functional Architecture for Intelligence Based on Generic Information Processing Tasks," *Proc. 10th Int'l Joint Conf. Artificial Intelligence (IJCAI-87),* vol. 2, pp. 1183-1192, 1987.
[18] B. Chandrasekaran, "Design Problem Solving: A Task Analysis," *AI Magazine,* pp. 59-71, 1990.
[19] B. Chandrasekaran, T.R. Johnson, and J.W. Smith, "Task Structure Analysis for Knowledge Modeling," *Comm. ACM,* vol. 35, no. 9, pp. 124–137, 1992.
[20] W.J. Clancey, "Heuristic Classification," *Artificial Intelligence,* vol. 27, pp. 289–350, 1985.
[21] W.J. Clancey, "Model Construction Operator," *Artificial Intelligence,* vol. 53, pp. 1–111, 1992.
[22] R. Dechter, "Constraint Processing Incorporating Backjumping, Learning and Cutset-Decomposition," *Proc. Fourth Conf. Artificial Intelligence Applications—CAIA '88,* pp. 312–319, 1988.
[23] J. de Kleer and B.C. Williams, "Diagnosing Multiple Faults," *Artificial Intelligence,* vol. 32, pp. 97–130, 1987.
[24] J. Domingue, "Tadzebao and WebOnto: Discussing, Browsing, and Editing Ontologies on the Web," *Proc. 11th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop (Univ. of Calgary),* B.R. Gaines and M. Musen, eds., 1998.
[25] H. Eriksson, Y. Shahar, S.W. Tu, A.R. Puerta, and M.A. Musen, "Task Modeling with Reusable Problem-Solving Methods," *Artificial Intelligence,* vol. 79, no. 2, pp. 293–326, 1995.
[26] E.A. Feigenbaum, "The Art of Artificial Intelligence: Themes and Case Studies of Knowledge Engineering," *Proc. Fifth Int'l Joint Conf. Artificial Intelligence (AAA-77),* 1977.
[27] D. Fensel, "Assumptions and Limitations of a Problem-Solving Method: A Case Study," *Proc. Ninth Banff Knowledge Acquisition for Knowledge-Based System Workshop (KAW '95),* 1995.
[28] D. Fensel, "Formal Specification Languages in Knowledge and Software Engineering," *The Knowledge Eng. Rev.,* vol. 10, no. 4, pp. 361–404, 1995.
[29] D. Fensel, "The Tower-of-Adapter Method for Developing and Reusing Problem-Solving Methods," *Knowledge Acquisition, Modeling and Management,* E. Plaza et al., eds., Springer-Verlag, 1997.
[30] D. Fensel, "Problem-Solving Methods: Understanding, Development, Description, and Reuse," *Lecture Notes of Artificial Intelligence (LNAI) 179,* 2000.
[31] D. Fensel and R. Benjamins, "The Role of Assumptions in Knowledge Engineering," *Int'l J. Intelligent Systems (IJIS),* vol. 13, no. 8, pp. 715–748, 1998.
[32] D. Fensel and V.R. Benjamins, "Key Issues for Problem-Solving Methods Reuse," *Proc. 13th European Conf. Artificial Intelligence (ECAI-98),* pp. 63-67, 1998.
[33] D. Fensel, H. Eriksson, M.A. Musen, and R. Studer, "Conceptual and Formal Specification of Problem-Solving Methods," *Int'l J. Expert Systems,* vol. 9, no. 4, 1996.
[34] D. Fensel, E. Motta, S. Decker, and Z. Zdrahal, "Using Ontologies For Defining Tasks, Problem-Solving Methods and Their Mappings," *Knowledge Acquisition, Modeling and Management,* E. Plaza et al., eds., Springer-Verlag, 1997.
[35] D. Fensel, R. Groenboom, and G.R. Renardel de Lavalette, "MCL: Specifying the Reasoning of Knowledge-Based Systems," *Data and Knowledge Eng. (DKE),* vol. 26, no. 3, pp. 243-269, 1998.
[36] D. Fensel, V.R. Benjamins, E. Motta, and B. Wielinga, "UPML: A Framework for Knowledge System Reuse," *Proc. Int'l Joint Conf. AI (IJCAI-99),* 1999.
[37] D. Fensel and R. Groenboom, "An Architecture for Knowledge-Based Systems," *The Knowledge Eng. Rev. (KER),* vol. 14, no. 3, 1999.
[38] D. Fensel and A. Schönegge, "Using KIV to Specify and Verify Architectures of Knowledge-Based Systems," *Proc. 12th IEEE Int'l Conf. Automated Software Eng. (ASEC-97),* 1997.
[39] D. Fensel and A. Schönegge, "Inverse Verification of Problem-Solving Methods," *Int'l J. Human-Computer Studies,* vol. 49, 1998.
[40] D. Fensel and R. Straatman, "The Essence of Problem-Solving Methods: Making Assumptions to Gain Efficiency," *The Int'l J. Human Computer Studies (IJHCS),* vol. 48, no. 2, pp. 181-215, 1998.
[41] D. Fensel and F. van Harmelen, "A Comparison of Languages which Operationalize and Formalize KADS Models of Expertise," *The Knowledge Eng. Rev.,* vol. 9, no. 2, pp. 105–146, 1994.
[42] N.E. Fuchs and M.P. Fromherz, "Schema-Based Transformations of Logic Programs," *Proc. LOPSTER 91, Int'l Workshop Logic Program Synthesis and Transformation,* 1991.
[43] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns,* Addison-Wesley, 1995.
[44] B.R. Gaines, "A Situated Classification Solution of a Resource Allocation Task Represented in a Visual Language," *Int'l J. Humna-Computer Studies,* pp. 243–271, 1994.
[45] J. Gaschnig, "Experimental Case Studies of Backtrack vs. Waltz-type vs. New Algorithms for Satisfying Assignment Problems," *Proc. Second Biennial Conf. Canadian Soc. for Computational Studies of Intelligence,* July 1978.
[46] J.H. Gennari and M. Ackerman, "Extra-Technical Information for Method Libraries," *Proc. 12th Workshop Knowledge Acquisition, Modeling and Management (KAW '99),* 1999.
[47] J.H. Gennari, A.R. Stein, and M.A. Musen, "Reuse for Knowledge-Based Systems and CORBA Components," *Proc. 10th Banff Knowledge Acquisition for Knowledge-Based System Workshop (KAW '96),* 1996.
[48] R.P. Graham Jr. and P.D. Bailor, "Synthesis of Local Search Algorithms by Algebraic Means," *Proc. 11th Knowledge-Based Software Eng. Conf. (KBSE-96),* 1996.
[49] D. Harel, "Dynamic Logic," *Handbook of Philosophical Logic, vol. II, Extensions of Classical Logic,* D. Gabby et al., eds., 1984.
[50] *High Performance Knowledge Bases (HPKB).* Darpa Project, available from http://www.teknowledge.com:80/HPKB/, 1997.
[51] G. Klinker, C. Bhola, G. Dallemagne, D. Marques, and J. McDermott, "Usable and Reusable Programming Constructs," *Knowledge Acquisition,* vol. 3, pp. 117–136, 1991.
[52] D. Landes and R. Studer, "The Treatment of Non-Functional Requirements in MIKE," *Proc. Fifth European Software Eng. Conf. (ESEC '95),* 1995.
[53] "Sisyphus '91/92: Models of Problem Solving," *Int'l J. Human-Computer Studies (IJHCS),* M. Linster, ed., vol. 40, no. 3, 1994.
[54] B.L. Lipman, "How to Decide How to Decide Hot to ...: Modeling Limited Rationality," *Econometria,* vol. 59, no. 4, pp. 1105–1125, 1991.
[55] *Automating Knowledge Acquisition for Experts Systems,* S. Marcus, ed. Boston: Kluwer Academic, 1988.
[56] S. Marcus, J. Stout, and J. McDermott, "VT: An Expert Elevator Designer That Uses Knowledge-Based Backtracking," *AI Magazine,* vol. 9, no. 1, pp. 95–111, 1988.
[57] J. McDermott, "Preliminary Steps Toward a Taxonomy of Problem-Solving Methods," *Automating Knowledge Acquisition for Expert Systems,* S. Marcus, ed., 1988.
[58] S. Mittal and F. Frayman, "Towards a Generic Model of Configuration Tasks," *Proc. 11th Int'l Joint Conf. Artificial Intelligence—(IJCAI '89),* 1989.
[59] E. Motta, *Reusable Components for Knowledge Modeling.* Amsterdam: IOS Press, 1999.
[60] E. Motta and Z. Zdrahal, "Parametric Design Problem Solving," *Proc. 10th Banff Knowledge Acquisition for Knowledge-Based System Workshop (KAW '96),* 1996.

[61] E. Motta and Z. Zdrahal, "An Approach to the Organization of a Library of Problem Solving Methods which Integrates the Search Paradigm with Task and Method Ontologies," *Int'l J. Human-Computer Studies (IJHCS),* vol. 49, 1998.
[62] E. Motta, D. Fensel, M. Gaspari, and R. Benjamins, "Specifications of Knowledge Components for Reuse," *Proc. 11th Int'l Conf. Software Eng. and Knowledge Eng. (SEKE '99),* pp. 36-43, 1999.
[63] E. Motta, S. Buckingham-Shum, and J. Domingue, "Ontology-Driven Document Enrichment: Principles, Tools and Applications," *Int'l J. Human-Computer Studies,* to appear.
[64] M.A. Musen, "Modern Architectures for Intelligent Systems: Reusable Ontologies and Problem-Solving Methods," *AMIA Ann. Symp.,* C.G. Chute, ed., 1998.
[65] A. Newell and H.A. Simon, "Computer Science as Empirical Enquiry: Symbols and Search," *Comm. ACM,* vol. 19, no. 3, pp. 113–126, 1976.
[66] N.J. Nilsson, *Principles of Artificial Intelligence.* Los Altos, Calif.: Morgan Kaufmann, 1980.
[67] K. O'Hara, "The GDM Grammar v.4.6. VITAL Project Report NOTT/T252.3.3.," available from the author at AI Group, Dept. of Psychology, Univ. of Nottingham, UK, 1995.
[68] K. Orsvärn, "Principles for Libraries of Task Decomposition Methods—Conclusions from a Case-Study," *Advances in Knowledge Acquisition,* N. Shadbolt et al., eds., pp. 48–65, 1996.
[69] K. Poeck and F. Puppe, "COKE: Efficient Solving of Complex Assignment Problems with the Propose- and-Exchange Method," *Proc. Fifth Int'l Conf. Tools with Artificial Intelligence,* 1992.
[70] A.R. Puerta, J.W. Egar, S.W. Tu, and M.A. Musen, "A Multiple-Method Knowledge Acquisition Shell for the Automatic Generation of Knowledge Acquisition Tools," *Knowledge Acquisition,* vol. 4, pp. 171–196, 1992.
[71] F. Puppe, *Systematic Introduction to Expert Systems: Knowledge Representation and Problem-Solving Methods.* Berlin: Springer-Verlag, 1993.
[72] J.T. Runkel, W.B. Birmingham, and A. Balkany, "Solving VT by Reuse," *Int'l J. Human-Computer Studies (IJHCS),* pp. 403–433, 1996.
[73] *Special issue on Sisyphus, Int'l J. Human-Computer Studies (IJHCS),* A.Th. Schreiber and B. Birmingham, eds., vol. 44, nos. 3-4, 1996.
[74] A.Th. Schreiber, B. Wielinga, J.M. Akkermans, W. Van De Velde, and R. de Hoog, "CommonKADS: A Comprehensive Methodology for KBS Development," *IEEE Expert,* vol. 9, no. 6, pp. 28–37, 1994.
[75] N. Shadbolt, E. Motta, and A. Rouge, "Constructing Knowledge Based Systems," *IEEE Software,* vol. 10, no. 6, pp. 34-38, 1993.
[76] W. Swartout, Y. Gil, and A. Valente, "Representing Capabilities of Problem-Solving Methods," *Proc. 1999 16th Int'l Joint Conf. Artificial Intelligence (IJCAI-99),* 1999.
[77] M. Shaw and D. Garlan, *Software Architectures: Perspectives on an Emerging Discipline.* Prentice-Hall, 1996.
[78] D.R. Smith and M.R. Lowry, "Algorithm Theories and Design Tactics," *Science of Computer Programming,* vol. 14, pp. 305–321, 1990.
[79] "Components of Expertise," *AI Magazine,* vol. 11, no. 2, 1990.
[80] M. Stefik, *Introduction to Knowledge Systems,* San Francisco: Morgan Kaufman, 1995.
[81] J. Stout, G. Caplain, S. Marcus, and J. McDermott, "Toward Automating Recognition of Differing Problem-Solving Demands," *Int'l J. Man-Machine Studies (IJMCS),* vol. 29, no. 5, pp. 599–611, 1998.
[82] A. Stutt and E. Motta, "Recording the Design Decisions of a Knowledge Engineering Community to Facilitate Re-Use of Design Models," *Proc. Ninth Banff Knowledge Acquisition for Knowledge-Based System Workshop (KAW '95),* 1995.
[83] A. ten Teije, "Automated Configuration of Problem Solving Methods in Diagnosis," PhD thesis, Univ. of Amsterdam, The Netherlands, 1997.
[84] P. Terpstra, G. van Heijst, B. Wielinga, and N. Shadbolt, "Knowledge Acquisition Support Through Generalized Directive Models," *Second Generation Expert Systems,* M. David et al., eds., 1993.
[85] M. Valasek and Z. Zdrahal, "Experiments with Applying Knowledge Based Techniques to Parametric Design," *Proc. Int'l Conf. Eng. Design—ICED '97,* A. Riitahuhta, ed., vol. 1, pp. 277-280, 1997.
[86] W. Van de Velde, "A Constructivistic View on Knowledge Engineering," *Proc. 11th European Conf. Artificial Intelligence (ECAI-94),* 1994.
[87] G. van Heijst, P. Terpstra, B.J. Wielinga, and N. Shadbolt, "Using Generalized Directive Models in Knowledge Acquisition," *Current Developments in Knowledge Acquisition,* T. Wetter et al., eds., 1992.
[88] G. Wiederhold, "Mediators in the Architecture of Future Information Systems," *Computer,* vol. 25, no. 3, p. 3849, 1992.
[89] G. Wiederhold and M. Genesereth, "The Conceptual Basis for Mediation Services," *IEEE Expert,* p. 3847, Sept./Oct. 1997.
[90] B. Wielinga, J.M. Akkermans, and A.Th. Schreiber, "A Formal Analysis of Parametric Design Problem Solving," *Proc. Ninth Banff Knowledge Acquisition for Knowledge-Based System Workshop (KAW '95),* 1995.
[91] D.M. Yellin and R.E. Strom, "Protocol Specifications and Component Adapters," *ACM Trans. Programming Languages and Systems,* vol. 19, no. 2, pp. 292–333, 1997.
[92] Z. Zdrahal and E. Motta, "An In-Depth Analysis of Propose & Revise Problem Solving Methods," *Proc. Ninth Banff Knowledge Acquisition for Knowledge-Based System Workshop (KAW '95),* 1995.
[93] Z. Zdrahal and E. Motta, "Improving Competence by Integrating Case-Based Reasoning and Heuristic Search," *Proc. 10th Banff Knowledge Acquisition for Knowledge-Based System Workshop (KAW '96),* 1996.

**Dieter Fensel** studied mathematics, sociology, and computer science in Berlin. In 1989, he joined the Institute AIFB at the University of Karlsruhe. His major subject was knowledge engineering and his PhD thesis in 1993 was about a formal specification language for knowledge-based systems. From 1994 until 1996, he was a visiting scientist at the group of Bob Wielinga at the SWI Department in Amsterdam. During this time, his main interest were problem-solving methods of knowledge-based systems. Then, he was back as a senior researcher at the Institute AIFB and he focused on the use of Ontologies to mediate access to heterogeneous knowledge sources and to apply them in knowledge management and electronic commerce. Currently, he is an associate professor at Vrije Universiteit Amsterdam working on intelligent information integration (i.e., semantic web) and electronic commerce.

**Enrico Motta** received a degree in computer science from the University of Pisa in Italy and the PhD degree in artificial intelligence from the Open University, England. He is a senior research fellow and assistant director at the Knowledge Media Institute of the Open University in England. He has been conducting research in knowledge systems for over a decade and has published more than 60 papers with topics such as knowledge acquisition, knowledge sharing and reuse, knowledge management, formal languages, and multiagent systems. His current research focuses on reusable components for knowledge modeling—ontologies and problem solving methods—and the application of knowledge modeling techniques to knowledge management. Application domains include engineering design, electronic news publishing, medical decision making, and scholarly discourse. Dr Motta is on the editorial board of the *International Journal of Human-Computer Studies* and a member of several international program committees. He has recently authored a book on knowledge sharing and reuse, entitled *Reusable Components for Knowledge Modelling*, which is published by IOS Press.