Börner, Katy. (1998). CBR for design. In Bartsch-Spörl, B., Wess, S., Burkhard, H. D., and Lenz, M. (Eds.), *Case-Based Reasoning Technology: From Foundations to Application*, Springer Verlag, LNAI 1400 <http://link.springer.de/link/service/series/0558/tocs/t1400.htm>, chapter 8, pp. 201-234

8. CBR for Design

Katy Börner

8.1 Introduction

Design research has a number of goals, including a better understanding of design, the development of tools to aid human designers, and the potential automation of some design tasks. Computer-aided design is concerned with using computers to assist in the design process to produce better designs in shorter time. The reuse of well-tested and optimized designs is an important aspect for decreasing design times, increasing design quality, and improving the predictability of designs.

This chapter summarizes the main results of applying CBR to support human problem solving in design. Its special focus is on research carried out in Europe and in particular in Germany. We start by providing a general characterization of design tasks. An overview of applicable reasoning methods, as well as a survey of case-based systems for design assistance, will be provided in Section 8.3. Following this, Section 8.4 discusses characteristics of case-based design (CBD) using the domain of architectural design for illustration. Reoccurring problems in CBD are listed. Sections 8.5 and 8.6 describe a number of methods that have been developed to handle the complexity of design tasks. Section 8.5 introduces an algorithm for flexible case retrieval that allows for the combination of several similarity measures at query time dynamically. In Section 8.6 three approaches to structural similarity assessment and adaptation are presented and discussed. Section 8.7 introduces EADOCS, a multi-level and hybrid expert system for conceptual design tasks that achieves structural adaptation by case combination. To conclude, approaches for automating configuration tasks, as introduced in Chapter 6, are contrasted with approaches aimed at the assistant-like support of *design* tasks and directions for future research are pointed out.

Rather than preparing new descriptions of the Fish & Shrink algorithm (Section 8.5) and the EADOCS system (Section 8.7), collaborative participation was solicited from the original authors *Jörg W. Schaaf* and *Bart Netten*, respectively.

8.2 The Design Task

Design is concerned with the construction of an artifact from single parts that may be either known and given or newly created for this particular artifact. Constraints on the artifact may be rigidly or informally defined.

Routine, Innovative, and Creative Design. There is a general acceptance of the classification of design tasks into *routine*, *innovative*, and *creative* that has proved to be useful (Coyne et al. 1987; Gero 1990).

Routine design can be defined as design that proceeds within a well-defined state space of potential designs. That is, all the variables and their application ranges, as well as the knowledge to compute their values, are directly derivable from existing designs.

Innovative design can be defined as non-routine design that proceeds within a well-defined state space of potential designs. In contrast to routine design, the applicable ranges for values of variables may change. What results is a design with a familiar structure but novel appearance because the values of the defining variables (and their combinations) are unfamiliar.

Creative design can be defined as non-routine design that introduces new variables and, as a result, extends or moves the state space of potential designs.

The development of knowledge-based systems has concentrated on routine design tasks (also called configuration tasks, see Chapter 6), i.e., tasks that involve a well understood problem solving space where all decision points and outcomes are known *a priori*. Routine design problems are typically represented by a well defined set of configurable components, a set of constraints that the final design must satisfy, and operators that encode valid component configurations (cf. Section 6.4). There is a rich set of approaches that can be used to efficiently control the search for solutions. Configuration can be seen as a comparatively tractable design task that allows for a closed world assumption in knowledge-based systems development and can be completely automated.

Innovative and creative design tasks are usually described by incomplete knowledge about the number and type of components to arrange. Often, a complete set of constraints that the final design must satisfy is not available. The problem solution may correspond to a set of design solutions that can be ordered corresponding to preference criteria. Knowledge about the validity of solutions is not available in general. Often the complexity of this reasoning process requires an iterative process in which the level of abstraction is incrementally refined to establish the solution. That is, each solution reduces the design space and serves as a starting point for local search at a more concrete level.

8.3 Design Assistance

In recognition of the open ended nature of the input data and the lack of formal methods, approaches to innovative and creative design support have aimed at *assisting* the user rather than automating the architectural programming process. Usually, design systems work as interactive assistants employing user intervention whenever needed to generate or evaluate a proper solution.

In order to built assistance systems that are effective and can easily be integrated into the daily working environment, adequate user interfaces must be provided (Pearce et al. 1992). If drawings are the central medium for the communication (e.g., in architectural designs), then a design support system should offer a graphical, CAD-like user interface. Adequate user interfaces are an indispensable requirement to enable high interactivity. This is needed for two reasons. Firstly, the huge amount of knowledge necessary to support complex design tasks requires partially automatic knowledge acquisition; i.e., without bothering the user to answer thousands of questions. Systems that require an immense effort for knowledge elicitation will rarely if ever succeed. Filling in large forms to label and save each possibly useful experience is very difficult to integrate into the workflow of potential users. Secondly, the complexity of such design processes needs to be communicated in an appropriate, domain specific way. Highly user-interactive frameworks, which manage knowledge elicitation during the system usage, are a real challenge to enable efficient computer-aided support in design.

The more complex the real-world applications, the higher the need for having deeply integrated system architectures. Knowledge acquisition, learning, and problem solving are advantageously viewed as constructive and cooperative processes, in which the user is an integrated part (Aamodt 1991; Veloso 1994). Ideally, the system works as a *learning apprentice* (Mitchell et al. 1985) improving reasoning performance continually by requiring a user to accept, correct, or refuse solutions or to provide solutions for problems. The knowledge is integrated into the knowledge base, compiled, and exploited during continuous problem solving. By this close linkage of knowledge acquisition and problem solving, the system is able to provide incrementally increasing support that is adaptable to the user and the peculiarities of the chosen domain. However, the integration of problem solving and learning in a single system still demands progress in both fields. This is especially true for systems that aim at the support of design tasks.

8.3.1 Reasoning Methods

There are several approaches which have been proposed to handle design tasks. Among them are formulae (Coyne 1988), constraints (Thagard et al. 1990), rules and grammars (Fu 1974; Gonzalez and Thomason 1978), autonomous agents (Morgenstern 1993; Bhat 1995), case-based reasoning (Goel

1989; Domeshek and Kolodner 1992; Hinrichs 1992a), and prototype-based reasoning (Gero 1990).

Formulae, grammars, or complete and consistent sets of constraints can only be defined for *routine design* tasks. They are often constructed based on the a priori knowledge available to the designers and their experiences. In order to support *innovative design* tasks, classical AI problem solving methods are not applicable, in general. Here, the use of experience is of particular importance and case-based reasoning comes into play. In order to support *creative design* tasks, the application of analogical problem solving is advantageous. It allows for the transfer of knowledge across different domains with special emphasis put on structural dependencies. This chapter concentrates on the application of case-based reasoning to support *innovative* and *creative* design tasks.

8.3.2 Case-Based Design Systems

There are many case-based design systems described in the literature. They have been proposed for a variety of domains that range from well-structured domains, such as the design of mechanical devices, to largely informal domains, such as architectural design. For example, Cyclops (Navinchandra 1988) does landscape design. Archie, Archie-II (Domeshek and Kolodner 1992; Domeshek and Kolodner 1997), Seed (Flemming et al. 1997), Janus (Fisher et al. 1989), Cadre (Hua et al. 1993) and Fabel (Voß 1997) support architectural design. Kritik (Goel 1989) and Kritik-II (Bhatta 1995; Goel et al. 1997) combine case-based with model-based reasoning for the design of physical devices. Software interface design is supported by AskJef (Barber et al. 1992), Asp-II and Benton (Tsatsoulis and Alexander 1997). Cadsyn (Maher and Zhang 1991; Maher and Zhang 1993; Maher et al. 1996) solves structural design tasks. Cadet is a case-based design tool that aids the conceptual design of electro-mechanical devices (Sycara et al. 1992; Narashiman et al. 1997). Detailed descriptions of many of these systems can be found in (Maher et al. 1996; Maher and Pu 1997).

Every application domain requires special considerations about the knowledge representation used and the retrieval, solution adaptation, and solution verification applied. Many systems support architectural design tasks. Therefore, this domain is used in Section 8.4 to illustrate a number of reoccurring problems that have to be addressed by every implemented system that aims at the support of design tasks.

Focusing on research pursued in Europe, Sections 8.5 and 8.6 describe research results achieved in Fabel, a major research project in Germany (Voß 1997). The aim of Fabel was to investigate ways of supporting design tasks by case-based and model-based methods, thus bridging the gap between case-based systems (which so far did little more than presenting former cases to the user) and expert systems, which embody theories and heuristics. Fabel has been a research project with a strong application orientation. In the Fabel

prototype, diverse approaches and tools to case retrieval and case adaptation have been developed (see (Voß 1994) and (Börner 1995c) for surveys). While the tools are rather independent, they have comparable user interfaces and work together according to the paradigm of a virtual construction site (Hovestadt and Schmidt-Belz 1995). Section 8.7 introduces EADOCS, a system that applies *structural adaptation by case combination* for the expert assisted design of composite sandwich panels (Netten et al. 1995).

8.4 Characteristics of Case-Based Design

Maher and Gomez de Silva Garza (1997) list three reoccurring themes in the implementation of case-based design systems; the need to represent and to manage *complex design cases*, the need to augment cases with *generalized design knowledge*, and the need to *formalize a typically informal body of knowledge*:

Complex Cases. According to Gebhardt et al. (1997) *complex cases* can be characterized as case which:

- may have to be cut out of large data models (such as CAD plans representing entire buildings);
- may not be described sufficiently in terms of attributes but have to be represented structurally (e.g., by graphs);
- contains variables that do not statically describe a problem or a solution. Instead these variables dynamically take the role of, e.g., problem variables if they match the query;
- may be useful in multiple ways and allow for more than one interpretation;
- may have to be composed and adapted to solve a problem.

The need for complex case representations has several implications. Among them are:

- Often, a new problem has to be reinterpreted or reformulated to be comparable with past experiences.
- Without distinguished problem and solution parts, the overlapping parts of a problem and past case(s) have to be identified and case parts for transfer and combination must be chosen.
- Multiple case interpretations require a flexible combination of several similarity functions. Similarity assessment has to proceed over complex structures.
- Different aspects of a case (e.g., pragmatic features, the structure of cases) may have to be jointly considered for retrieval, match, and adaptation.

Taken together, complex case representations cause increased computational expense in the retrieval, matching and adaptation of cases. To guarantee answer times that are acceptable for real world applications, efficient memory

organizations directly tailored to the applied reasoning mechanisms are essential. Problems that relate to the amount and the structural complexity of the knowledge have to be addressed.

Generalized design knowledge. Design knowledge may include causal models, state interactions, heuristic models, heuristic rules, and geometric constraints. Often, this knowledge is not available for *innovative* and *creative* design tasks.

Lack of formal knowledge. In situations where only an informal body of knowledge is available, this may result in case representations that are suited to *support* human problem solving rather than automated reasoning or by focusing on tasks that can be formalized. However, this chapter concentrates on approaches that assist design tasks despite the informal character of the given knowledge.

8.4.1 Case-Based Architectural Design

Much work in CBD has been carried out in the domain of architectural design. Reasons for this may be the centrality of past experiences and the economical impact of design support. Both aspects are outlined in what follows.

In architectural design, prior experiences, typically represented by CAD-layouts, constitute the main source of knowledge. These layouts are used to inspire, guide, and communicate architectural work. By applying CBR, annotated layouts are employed directly in the case base. During problem solving, cases serve as a first guess to shortcut reasoning from first principles. Conversely, cases allow architects to refine, supplement, and qualify the rules of thumb, principles, and theories architects learned at school or university. The continuously increasing amount of electronically available data, the electronical connection of architectural bureaus, and the continuing standardization ease the access and reuse of this huge amount of design knowledge. The centrality of layouts directly suggests an application of case-based approaches to support design tasks.

Additionally, architectural design is one of the keystones to economic competitiveness. Each country spents about 30 percent of it gross national product (i.e., the annual total value of goods produced and services provided) for housing. In the modern competitive world, designers are under a constant pressure to turn out new and innovative products quickly. As a consequence, computational models for architectural design are important research topics and the development of computational models founded *Artificial Intelligence* paradigms has provided an impetus for much of current research in this direction (Coyne et al. 1988; Gero and Sudweeks 1994). For of all these reasons, the domain of architectural design was selected for illustration purposes here.

8.4.2 Knowledge Representation and Reasoning

As introduced in Section 1.3.8, four knowledge containers of a CBR system are distinguished; the *vocabulary used*, the *similarity measure*, the *case base*, and the *solution transformation*. We assume that the knowledge required to evaluate solutions is included in the container for *solution transformation*. While the content of the containers can be changed locally, the knowledge contained in these containers has to supplement each other. That is, the similarity measure has to be defined in such a way that it allows the comparison of queries and cases in their corresponding representations and to derive some degree of similarity from this comparison, for example, as a numerical value in the interval $[0, 1]$. The cases, the similarity measure, and the solution transformations will define the space of possible solutions.

Vocabulary. The selection of an appropriate vocabulary is to a great extend task and domain dependent. The vocabulary should be able to capture all the salient features of the design that are relevant to support problem solving in the selected domain.

Cases. Examining the way architects work reveals a wide variety of what should constitute a design case: the size varies from few design objects to a whole storey or even an entire building; the case layout could be a simple list of some properties or a complex structure involving many types of relations between components with composite attributes.

Among the major considerations in representing past design experience by cases are the *usage*, the *granularity*, the *level of abstraction,* and the *perspective* cases should have:

Usage: There are two ways cases can be used. On the one hand, cases may represent abnormal situations or exceptions while rules are applied to capture norms and regular situations. On the other hand, cases may successfully be used to capture regular or normal situations in a more natural way (especially if other knowledge is not available). Another decision refers to the use of positive *good* cases or the incorporation of negative cases helping to anticipate and thus avoid mistakes made in the past.

Granularity: In architecture, complete buildings have been taken as cases (Goel 1989; Domeshek and Kolodner 1992; Hinrichs 1992b). Other approaches promoted user-defined cases which are marked in an overall project in a creative way that is hardly definable and repeatable (Voß 1994). Aiming at a task-oriented user support, the grain size of cases matches the grain size of decisions the architect needs to make. A uniform, task-oriented methodology to derive cases out of CAD layouts representing entire projects was proposed by Janetzko et al. (1994).

Level of abstraction: Corresponding to their level of abstraction, Riesbeck and Schank (1989, p.12) distinguish *ossified cases*, *paradigmatic cases*, and *stories*. *Ossified cases* are like general rules of thumb and independent of the events they were originally derived from. They tend to be relevant

in only the areas for which they were originally intended and form the basis for making everyday decisions. *Paradigmatic cases* represent the one experience that was in any way relevant to what you are now experiencing and the task left is to find where the current situation differs in order to adapt the case to solve the new situation. They form the base of expertise that a person accumulates beyond the textbook rules that s/he has been taught. *Stories* relate by virtue of their complexity and myriad aspects to a large variety of possible circumstances. They can be indexed and, later, accessed in multiple ways. Creativity depends on the ability to analyze stories effectively and under various poin ts of view.

Perspective: Cases may be acquired according to a *state-oriented* perspective or a *solution-path* perspective. While state-oriented cases represent essentially the problem and its solution, solution-path cases refer to the process or operator that derives the solution from the problem description.

Similarity Measure and Case Retrieval. Basically, there are two different approaches to similarity assessment in CBR. The *computational approach* (Tversky 1977; Stanfill and Waltz 1986; Aha 1991), which is based on computing an explicit similarity function for all cases in the case base, and the *representational approach*, proposed by (Kolodner 1980; Kolodner 1984) using a structured memory of cases. Some techniques, such as *k*d-trees and CRNs (both introduced in Chapter 3), attempt to combine these two fundamental approaches.

Similarity approach. Case sets are stored in an unstructured way. Retrieval is performed on the basis of a similarity measure[1]. Most approaches to similarity assessment in CBR estimate the usefulness of cases, based on the presence or absence of certain features (cf. Domeshek and Kolodner 1992; Richter 1992a). The features are pre-classified as important with respect to retrieval. Similarity is assessed by a numeric computation and results in a single number which is intended to reflect all aspects of the similarity.

To reduce the complexity of structural comparisons, two-stage models have been proposed in the literature, e.g., the MAC/FAC model (Gentner and Forbus 1991). In the first stage, Many cases Are Called using a computationally cheap filter. Here, flat vector representations of a past case and a problem representing the predicates are matched to identify a number of possible candidate cases; information about the inter-relating structure between these predicates is not considered. In the second stage, Few of these possible candidates Are Chosen by carrying out a structure mapping between the problem and each candidate resulting in useful, structurally sound matches. In such a way, the complexity of phenomena in similarity-based access can be reduced.

[1] See also memory-based reasoning (Stanfill and Waltz 1986) or instance-based learning (Aha 1991; Aha et al. 1991)

Representational approach. For representational approaches, the case base is pre-structured. Retrieval proceeds by traversing the index structure, e.g., *memory organization packets* (Schank 1982; Kolodner 1984). Cases that are neighbors according to the index structure are assumed to be similar. Constraints on a problem serve as indices into the memory. Probing the memory, returns cases that provide a solution, some of the context as well as feedback for external evaluation. This information is used to determine how applicable the case is, how to adapt it, and how to avoid repeating previous failures.

If case-based reasoning is applied to support *innovative* and *creative* design tasks, then case retrieval requires:

Flexible case retrieval: The need to exploit different views on single cases requires a method for so-called flexible case retrieval. Given a large case base, a problem, and a number of aspects that are relevant for similarity assessment, a set of cases have to be retrieved which show similar aspects as in the actual problem. Be aware that the importance of certain aspects for similarity assessment is not known at memory organization time. That is, different similarity measures, each determining the similarity of a case and a query under a certain point of view have to be dynamically composed during retrieval. Approaches, such as kd-trees (Wess 1995) or Case Retrieval Nets (cf. Chapter 3), allow for dynamic weighting of certain features to be considered during similarity assessment but apply exactly one similarity measure.

Structural similarity assessment: In order to consider the structure of cases during retrieval, similarity assessment has to process structural case representations in which variables dynamically take the role of problem or solution variables.

Similarity assessment in terms of adaptability: Conventional CBR systems treat retrieval, matching (or justification), and adaptation separately and sequentially. Recent work, however, shows that the integration of these stages, especially retrieval in terms of adaptability, improves the suitability and performance of CBR significantly (Smyth and Keane 1993). A more general notion of case usability (instead of similarity) is required (Paulokat et al. 1992). Ideally, similarity should imply adaptability. That is, retrieved cases should be adaptable to correctly solve a current problem. Existing techniques for so called *adaptation-guided retrieval*, such as implemented in DÉJÀ VU(Smyth and Cunningham 1992; Smyth and Keane 1993) try to determine similarity without actually performing the computationally expensive adaptations.

Solution Transformation and Case Adaptation. In design, new situations rarely match old ones exactly. Even small differences between the actual problem and the most similar case may require significant adaptations. Adaptation plays a central role and comprises the selection of the parts of the past solution(s) that need(s) to be adapted, as well as the adaptation itself. Cunningham and Slattery (1993) distinguish three general kinds of adaptation:

(i) *Parametric adaptation* corresponds to the substitution, instantiation or adjustment of parameters.
(ii) *Structural adaptation*, as done by *transformational analogy* (Carbonell 1983b) or approaches that use grammars (Fu and Bhargava 1973), retrieves stored solutions and revises them by applying adaptation operators (or grammar rules) to solve a new problem.
(iii) *Generative adaptation*, such as *derivational analogy* (Carbonell 1986), reuses and adapts problem-solving episodes by replaying their derivation.

Given that architectural design is a weak theory domain, hardly any information about the relevance of features guiding the selection of similar (i.e., *adaptable*) cases is available. Often, the adaptation of prior layouts mainly corresponds to adding, eliminating, or substituting physical objects. Because of the variety and the possible high number of combinations of these modifications, adaptation knowledge is difficult to acquire by hand. Here, *structural adaptation* (as introduced in Sections 8.6.2 to 8.6.4) or *adaptation by case combination* (see Section 8.7) may provide a better approach.

8.5 Fish & Shrink Algorithm for Flexible Case Retrieval

This section proposes an algorithm for flexible case retrieval, named *Fish & Shrink*, that is able to search quickly through the case base, even if the aspects that define usefulness are spontaneously combined at query time.

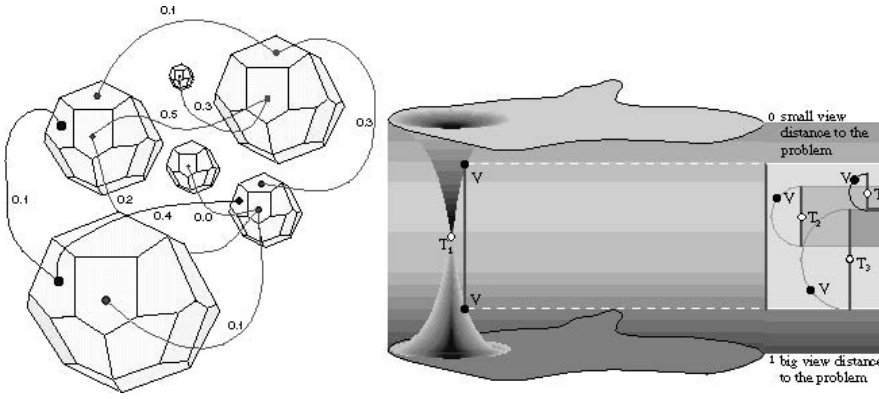
8.5.1 Required Functionality

Given rich and complex cases, as in the design domain, it is reasonable to use them in different contexts by regarding different aspects. For example, if an engineer and an interior decorator both try to solve a problem, they can use the same case base if each one gets result cases selected with focus on their own special needs. To exhaust the full potential of inherent solutions of cases, we suggest to represent cases with respect to as many different aspects as reasonable. Regarding different aspects leads to different representations of cases, each trying to catch a very specific view of them.

We do not have any formal definition of what an aspect is. Nevertheless the idea of regarding aspects leads to some formal constructs. We use the name of an aspect to identify a point of view and to index a pair of functions. Firstly, α_{name} denotes a *representation function*. As input, α_{name} takes the original representation of a case and as output it produces a representation in the aspect representation space Ω_{name} which emphasizes features important to the aspect idea and which is usually condensed. Secondly, the distance function δ_{name} takes two representations (from Ω_{name}) and calculates the distance of two cases with regard to this particular aspect. Fulfillment of the triangle inequality is recommended but not postulated. How the presented

approach handles violations of the triangle inequality is presented in Schaaf (1998).

As a result, a case base can be seen as a multiply linked set of polyhedrons (Figure 8.1(a)). Each polyhedron (representing a case) consists of several aspect representations linked to corresponding aspect representations of other cases by edges. These edges are labeled with the calculated distance between cases concerning one particular aspect.



(a) (b)

Fig. 8.1. Illustration of Fish & Shrink: (a) The case base as a network of cases. (b) The retrieval metaphor of sinking cases being dragged down by others

Up to this point, most standard approaches towards case base organization are very similar. Each network concerning one aspect can be seen as a separate case base with its own search structure. Problems arise in these standard approaches if *ad hoc* combinations (for example, a weighted combination) of several aspects should be allowed.

To describe a new approach, called the *Fish & Shrink* method, we first introduce the term *view distance SD* defined between two cases; the customized distance based on predefined aspect distances δ_i. As an example, the view distance may be the result of the function

$$SD(F_x, F_y, \mathbf{W}) = \sum_{i=1}^{n} w_i \delta_i(F_x, F_y)$$

where two cases F_x and F_y and a weight vector $\mathbf{W} = (w_1 ... w_n)$ serve as input. The weight vector holds an importance value for each aspect and can be manipulated in an *ad hoc* way; i.e., before each search request. Experts in case-based reasoning and domain experts are invited to invent more sophisticated view distance functions. They can be integrated online and tested.

Each view distance function ought to fulfill the triangle inequality. Violation can be detected while running the algorithm. Schaaf (1998) suggests a fault compensation based on empirical data.

8.5.2 Fish & Shrink

Given reasonable aspects, the corresponding representation functions, proper aspect distance functions, and a collection of view distance functions, we now define an anytime algorithm to search the case base.

The metaphor which led to the Fish & Shrink algorithm is sketched in Figure 8.1(b). Imagine cases being positioned in a continuous space. The closer a case (represented by a small circle or dot) is positioned to the top region of this space the smaller is its distance to the problem. While running, the Fish & Shrink algorithm successively picks up and directly tests cases from the *surface* which is represented as a plain. Each direct test has two effects: Firstly, the tested case sinks to a position according to its view distance to the problem. Secondly, cases in its neighborhood disappear from the surface and from the bottom. This effect is represented by two craters with the tested case at their joint peak. The craters indicate that cases in the nearby neighborhood of sunk test cases are dragged down deeper than cases afar. Closeness to the surface recommends cases to further examination.

Figure 8.1(b) shows what happens while the series of cases $(T_1, ..., T_4)$ is being directly compared to the problem. Firstly, T_1 sinks according to its view distance to the problem. A case labeled V in the neighborhood of T_1 becomes part of the two craters. Using the triangle inequality, minimum and maximum distance between V and the problem are computed. Both distances are represented by the position of V within the craters of test case T_1. The remaining interval of possible distances between V and the problem (represented by a line) is called the *uncertainty interval* of V. Further direct tests of cases from the surface can only shrink this interval. This is what happens to the uncertainty interval of V while testing cases $(T_2, ..., T_4)$. To simplify illustration of these tests, only the uncertainty intervals and case positions are pictured. If no case is left upon the surface, the surface level starts to sink and cases which have been sunk earlier reappear.

The main idea represented in Figure 8.1(b) is that it should be more efficient to avoid searching in the nearby neighborhood of cases which have already been found to be inappropriate. Inappropriateness is seen and treated as a gradual statement and depends on how close the neighborhood was.

Thus, the fundamental concept of the formal Fish & Shrink algorithm is the *uncertainty interval* which is applied to each case. Initially (when a new retrieval task starts), it is set to $[0, 1]$. The initial value represents the fact that we do not know anything about the distance between the case and the user's problem. The Fish & Shrink algorithm tries to position the uncertainty interval and minimize its length. Both increase knowledge about the

corresponding case. The target of Fish & Shrink is to maximize the number of shrunk uncertainty intervals and the amount of shrinking in each step without losing reliability. Between two shrinking loops, the algorithm tries to interpret the position and length of uncertainty intervals to find whether user demands have already been fulfilled or not. The algorithm conceptually performs the steps depicted in Figure 8.2.

Input: The case base CB, the problem A, a vector of aspect weights **W** and the actual view distance function SD.
Output: A sequence of cases with increasing distance to the problem.

1. Represent problem A in all aspects with corresp. field in $\mathbf{W} \neq 0$.
2. Set uncertainty interval for each case to $[0, 1]$.
3. Set precision line to 0.
4. While user demands not fulfilled and not interrupted:
   a) Move precision line according to prescription and present all cases that have been passed.
   b) Select (fish) a test case T depending on the position of the precision line.
   c) Calculate actual $SD(A, T, \mathbf{W})$ (direct test).
   d) $\forall$ view neighbors V of T do (indirect test):
      i. Minimum distance between V and A := $Max\Big(|SD(A,T) - SD(T,V)|,\ minimum\ distance\Big)$;
      ii. Maximum distance between V and A := $Min\Big(SD(A,T) + SD(T,V),\ maximum\ distance\Big)$;
   e) Recalculation of view neighbors finished.

Fig. 8.2. A sketch of the Fish & Shrink algorithm

To follow the steps of the algorithm, some more terms need to be explained. Firstly, in line 3, the *precision line* is set to zero. The precision line is a concept corresponding to the idea of having a surface level. Its task is to mark the smallest minimum distance of all uncertainty intervals (with length > 0) during retrieval. If an uncertainty interval (even with length > 0) is passed by the line, the corresponding case belongs to the output stream. A predefined movement prescription of the precision line determines the behavior of the algorithm. By using special prescriptions, answers to the following questions are possible:

- Which cases are better than a threshold?
- Which are the k best cases without ranking?
- Which are the k best cases with ranking?
- Which are the k best cases with ranking and their exact distances to a given problem?

The biggest challenge in developing precision line prescriptions is to avoid over-answering a user demand. For example, if a user wants to know the k

best cases, there is possibly no interest in getting a ranking of them. Leaving unasked questions unanswered can help to save time.

Each case V with $SD(T,V) \geq 0$ is called a *view neighbor* of T. The loop starting in line `4d` examines the view neighbors of a test case T. The uncertainty interval of each view neighbor of T is recalculated in that loop. Recalculation of the uncertainty intervals is done by using three interpretations of the triangle inequality.

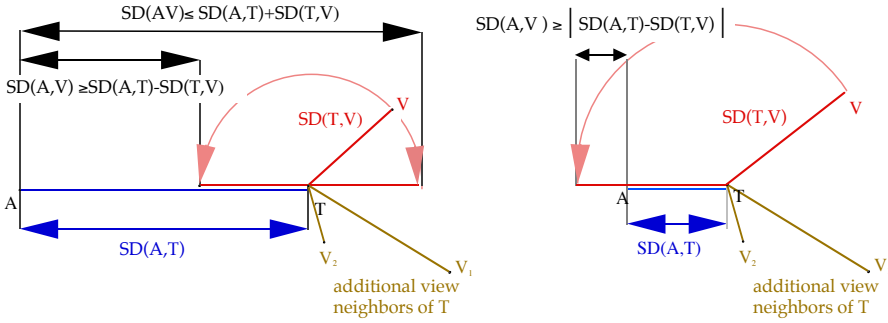


Fig. 8.3. Three interpretations of the triangle inequality to shrink uncertainty intervals of view neighbors of directly tested cases

Two interpretations, shown in Figure 8.3, maximize the lower limit (minimum distance). One interpretation minimizes the upper limit (maximum distance) - compare lines `4di` and `4dii` of Figure 8.2. As mentioned, recalculation can only shrink intervals. Because every direct test (line `4c`) shrinks at least one interval to zero, Fish & Shrink definitely terminates. It can stop before having tested all cases if an interrupt condition becomes true. The interrupt condition depends on the user demand and is usually described within the precision line prescription.

The Fish & Shrink algorithm delivers the correct and complete set of result cases if the triangle inequality holds for distance functions. In contrast to most competitors in the field of retrieval algorithms for case based reasoning, Fish & Shrink does not make the *closed world assumption* that undocumented similarity automatically means implicit dissimilarity.

Fish & Shrink normally comes up with the requested results without searching the whole case base. Up to now, we have not been able to give exact information about the theoretical complexity. Some empirical tests have shown a positive behavior (Schaaf 1998). Systematic tests to confirm the evident performance of Fish & Shrink and the applicability of the presented approach have still to be performed.

To summarize, the Fish & Shrink algorithm enables the efficient exploitation of the different views on single cases. While in most CBR approaches to retrieval exactly one (sometimes dynamically weighted) similarity measure is

used, Fish & Shrink combines different similarity measures dynamically at retrieval time time, thus, achieving flexible case retrieval.

8.6 Approaches to Structural Similarity Assessment and Adaptation

As stressed in Section 8.4, case retrieval for *innovative* and *creative* design tasks requires structural similarity assessment in terms of adaptability, as well as structural adaptation. This section introduces approaches that define the *structural similarity* (cf. Section 1.3.3) between structured case representations, i.e. graphs, via their *maximal common subgraph* (*mcs*) (Börner 1993; Jantke 1994). Given a new problem, the structurally most similar case(s) are retrieved. One out of several *mcs* is transferred and the remaining case parts are transferred as needed. Additionally, the *mcs* may be used to represent and access classes of structurally similar cases in an efficient manner. The remaining case parts can be seen as proper instantiations of *mcs*, i.e., as a special kind of adaptation knowledge that allows the adaptation of past cases to solve new problems. In such a way, structural case representations and limited domain knowledge is explored to support design tasks. The approaches have been exemplarily instantiated in three modules of the design assistant system FABEL-Idea that generates adapted design solutions on the basis of prior CAD layouts. For more details see Börner et al. (1996).

8.6.1 Required Functionality

The application domain used for motivation, illustration, and evaluation is architectural design. In particular we are concerned with supporting the *spatial layout*[2] of rooms and pipe systems in rectangular buildings.

Past experiences, stored as cases, correspond to arrangements of physical objects represented by CAD layouts and refer to parts in real buildings. Each object is represented by a set of attributes describing its geometry (i.e., position and extent in three dimensions) and its type (e.g., fresh air connection pipe). Concentrating on the design of complex installation infrastructures for industrial buildings, cases correspond to pipe systems that connect a given set of outlets to the main access. Pipe systems for fresh and return air, electrical circuits, computer networks, phone cables, etc., are numerous and show varied topological structures. For the retrieval, transfer, and adaptation of past cases to new problems not the geometry and type of single objects but their topological relations are important.

Because of the above, objects and their (topological) relations need to be represented and considered during reasoning. Therefore, a compile function

[2] *Spatial layout* can be seen as a representative of design tasks in general (Coyne 1988).

is used to translate attribute value representations of objects and their relations into graphs. In general, objects are represented by vertices and relations between objects are represented by edges. Reasoning, i.e., structural retrieval and adaptation, proceeds via graph-based representations. A recompile function translates the graph-based solution into its attribute-based representation that may be depicted graphically to the user. Concentrating on different aspects of structural similarity assessment and adaptation, different compile functions are appropriate, resulting in different graph representations of cases with their corresponding expressive power and reasoning complexity. They are explained in detail in Sections 8.6.2 to 8.6.4.

Figure 8.4 shows a design problem and its solution. Spatial relations (touches, overlaps, is_close_to) that can be used to represent cases structurally are visualized by arrows in the figures.

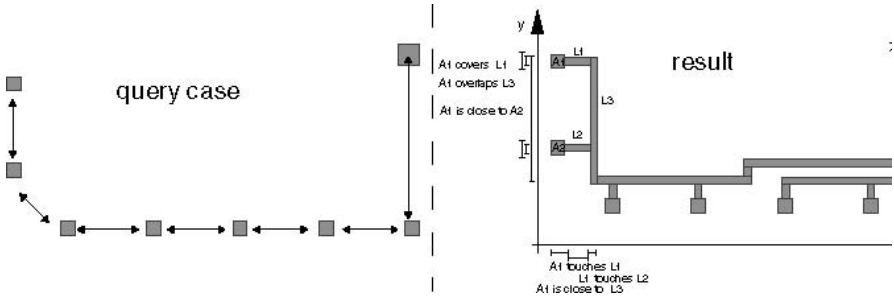


Fig. 8.4. Domain example: A problem and its solution

Given a base of cases represented by graphs and a structurally represented problem, reasoning proceeds as follows: Firstly, one or a set of cases that show a high *structural similarity* to the problem is/are retrieved. Here, structural similarity is defined via the maximal common subgraph (*mcs*) of a case and a problem. Assuming that cases and problem share more in common than their maximal common subgraph, *structural adaptation* proceeds by transferring and combining case parts that connect unconnected problem objects to the *mcs*.

In the following, we define the functionality of *structural similarity assessment and adaptation* by the mappings required to transfer a set of cases and a problem into exactly one solution or a set of problem solutions.

We give some basic notations first. A graph $g = (V^g, E^g)$ is an ordered pair of vertices V^g and edges E^g with $E^g \subseteq V^g \times V^g$. Let $mcs(G)$ denote the set of all maximal common subgraphs of a set of graphs G, with respect to some criteria. If there is no danger of misunderstanding, the argument of *mcs* will be omitted. Let Γ be the set of all graphs, and O be a finite set of objects represented by attribute values for geometry and type. $\mathcal{P}(\Gamma)$ denotes the power set of Γ; that is, the set of all subsets of Γ.

The mappings needed to accomplish the required functionality are depicted in Figure 8.5. Knowledge is denoted by circles and boxes denote functions. The indices $\_a$ and $\_g$ refer to attribute-value and graph representations, respectively. Arrows denote the sequence of mappings. The double arrow refers to the interaction between the compile and recompile function applied.

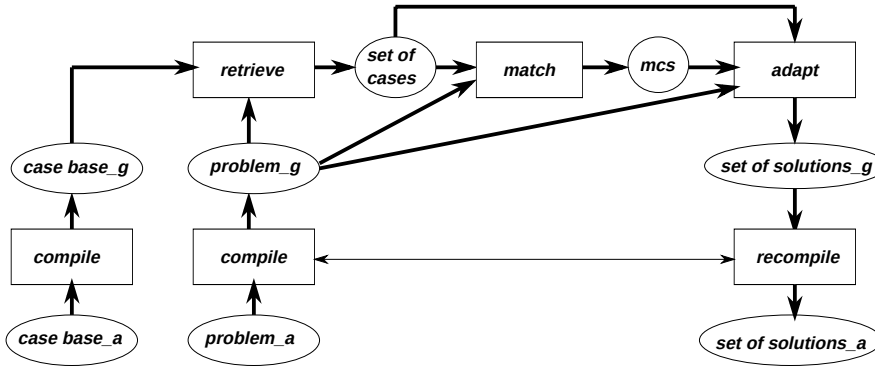


Fig. 8.5. Mappings required to accomplish the desired functionality

In order to access and interact via CAD layouts (that are represented by attribute values of their constituting objects) but to reason via their topological structure, there must be a way of translating attribute value representations of cases into graph representations and vice versa. Therefore, a compile function has to be defined that maps the attribute value representation of a set of objects representing a case or a problem into its structural representation:

$$\mathsf{compile} : \mathcal{P}(O) \rightarrow \Gamma$$

Inversely, the function recompile maps the graph representation of a set of objects denoting a solution into their attribute value representation:

$$\mathsf{recompile} : \Gamma \rightarrow \mathcal{P}(O)$$

The concept of structural similarity allows the selection of one or more cases, that are suitable for solving a problem. It is used by the function retrieve, that maps a set of cases (i.e. the case base) and a problem into a set of candidate cases that are applicable to solve the problem:

$$\mathsf{retrieve} : \mathcal{P}(\Gamma) \times \Gamma \rightarrow \mathcal{P}(\Gamma)$$

The retrieve function uses (sometimes repeatedly) a function named match that maps two graphs into their maximal common subgraph(s) *mcs*:

$$\mathsf{match} : \Gamma \times \Gamma \rightarrow \mathcal{P}(\Gamma)$$

As for adaptation, a mcs is selected and transferred to the problem. If needed, vertices and edges of the selected set of candidate cases are combined to complement the problem resulting in a set of solutions:

$$\mathsf{adapt} : \mathcal{P}(\Gamma) \times \Gamma \times \Gamma \rightarrow \mathcal{P}(\Gamma)$$

Given a finite object set and a compile function, we can restrict the last four mappings to finite domains, i.e., Γ may be replaced by $\mathsf{compile}(\mathcal{P}(O))$.

In the following, three approaches are presented that provide the defined functionality. The approaches differ in the compile and recompile functions applied, the graph representations (trees or arbitrary graphs) used, the memory organization applied and the retrieval and adaptation strategies proposed.

8.6.2 Topo

Topo (Coulon 1995) is a module that considers geometric neighborhoods as well as structural similarity to support the case-based extension and correction of three-dimensional rectangular layouts.

Compile and Recompile. The compile function used by Topo detects binary topological relations of various types. The type of a relation is determined by the application dependent attributes of involved objects and their 3-dimensional spatial relations. Topo's compile function projects each layout to the three geometric dimensions. For each projection, 8 different directed relations (similar to the temporal relations of Allen 1984) and several classes of disjoint intervals can be detected. Thus, a given object may be in one of 16 relationships for each dimension leading to $16^3 = 4096$ different 3-dimensional relationships (Coulon 1995).

Retrieval. Topo uses the retrieval algorithm Fish & Shrink (see Section 8.5) and an attribute-value based similarity function to retrieve a case that shows a high similarity to the problem at hand. Subsequently, a match function is applied to determine the maximal common subgraph of the graph-based represented case and the problem. Finding matching subgraphs is known to be a NP-complete problem (Babel and Tinhofer 1990). Instead of searching for a common subgraph of two graphs, Topo searches for a maximal clique in one graph representing all possible matches between the two graphs, called their *combination graph*.

Building the Combination Graph. Using the transformation given in Barrow and Burstall (1976), the vertices in the combination graph represent all matchings of compatible vertices in the graphs. Figure 8.6 shows an example. The graphs f and g contain objects of type a and b connected by directed relations. The type of a relation is defined by the types of its objects. Two vertices are connected in the combination graph if and only if the matchings represented by the vertices do not contradict one another. The matchings

$(R_2(a,b) \Leftrightarrow R_8(a,b))$ and $(R_5(b,b) \Leftrightarrow R_{10}(b,b))$ are connected because both relations occur in both graphs in the same context. Both are connected by a shared object of type b. $(R_2(a,b) \Leftrightarrow R_8(a,b))$ and $(R_1(b,a) \Leftrightarrow R_6(b,a))$ are not connected because the matched relations share an object of type a in graph f but do not share any object in graph g.

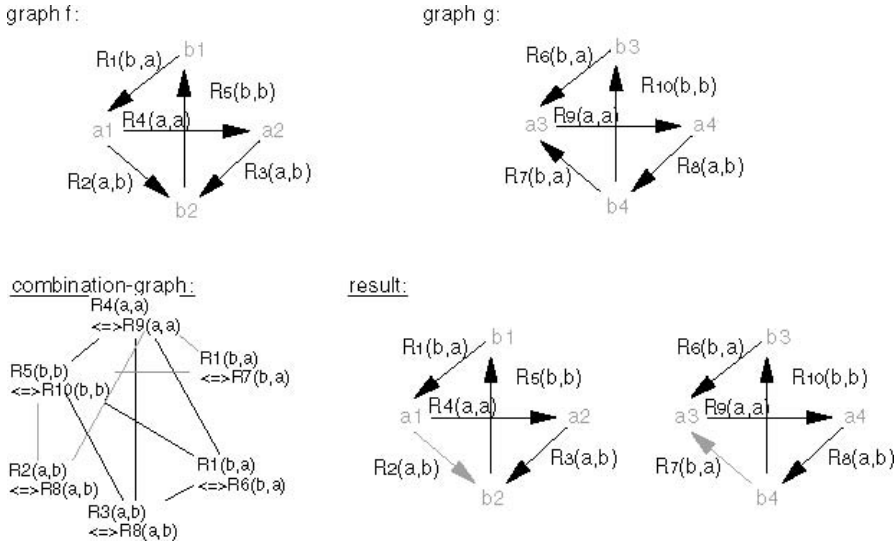


Fig. 8.6. Transformation of the problem of finding the maximal common subgraph to the problem of finding a maximal clique in a graph. The maximal clique and the corresponding matching are marked in black

A General Maximal Clique Algorithm. The algorithm of Bron and Kerbosch (1973), called *max-clique*$_{BK}$, finds all cliques in a graph by enumerating and extending all complete subgraphs. It extends complete subgraphs of size k to complete subgraphs of size $k+1$ by adding (iteratively) vertices which are connected to all vertices of the complete subgraph.

Given two graphs, the first similarity function returns the size of the maximal common subgraph relative to the minimum size of both graphs. Because a maximal common subgraph cannot be larger than the smaller of both graphs, the result is always a rational number between 0 and 1. In order to avoid the NP-complete search for the maximum common subgraphs, a second similarity function is defined. It compares the sets of relations occurring in both graphs. The result is the number of compatible relations relative to the minimum number of detected relations of both graphs, leading to a result which is also between 0 and 1. The second similarity function obviously returns an upper bound of the first similarity function, but has no NP-complexity.

Adaptation. TOPO extends, refines, and corrects layouts by case adaptation. Given that case parts can take the role of a problem or a solution,

Topo uses the heuristics that every object of the case which is not found in the problem might be part of the solution. After determining the common subgraph of a case and a problem, Topo transfers those case objects to the problem that are connected to the common subgraph by a path of topological relations. The user may influence this process by selecting types of objects to transfer.

During transfer, Topo may change the size of transferred objects to preserve topological relations. For example, a window is resized in order to touch both sides of a room. To avoid geometries which are impossible for an object, Topo limits the resizing to geometries which occurred in the case base. Additionally, statistics about the frequency of topological relations occurring in the case base for each type of objects supports the evaluation of solutions.

8.6.3 Macs

Macs extends the approach of structural similarity and adaptation to arbitrary graphs. Clumping techniques are applied in order to reduce the number of NP-complete matches during retrieval.

Compile and Recompile. The function compile guarantees the transformation of an attribute value represented case or problem into its graph representation. The graph representation should reflect the structure of the domain objects. In the selected domain, this can be achieved by representing all objects of a case as vertices and representing the topological relations between neighboring vertices by edges. Vertices may be labeled by the type of the object or by additional qualitative and quantitative attributes. Edges can be labeled with a type name of the topological relationship together with additional attributes. If the layout contains a certain number of spanning objects, such as pipes or beams, then it is more suitable to represent the spanning objects as edges and the remaining objects as vertices and, if necessary, to label them with their type name and necessary additional attributes.

The function recompile transforms vertices and edges of a graph-based solution into objects and relations of a concrete layout. In case of labeled graphs, this mapping is unique. Otherwise, the problem of non-unique mappings has to be solved. See Bartsch-Spörl and Tammer (1994) for examples.

Organization of the Case Base. In this approach, we consider arbitrary graphs which may be either directed or undirected and labeled or unlabeled. Computing structural similarity equates to the computation of the maximal common isomorphic subgraph(s). Macs applies a backtracking algorithm that realizes the function match to compute maximal common subgraphs of two arbitrary graphs (Tammer et al. 1995). In general, the maximal common subgraph of a set of graphs is not unique. For any collection of graphs $C \in \mathcal{P}(\Gamma)$, we use $mcs(C)$ to denote the set of maximal common subgraphs of all graphs in C with respect to counting vertices and edges. In order to

determine a unique representative $\Theta \in mcs(C)$ for each class C, we introduce some *selection operator* $E : \mathcal{P}(\Gamma) \to \Gamma$.

Usually, the peculiarities of the application domain lead to a number of preferable selection operators:

- *other similarity concepts* based on labels of vertices or edges of graphs within $mcs(C)$ may be applied or
- *graph–theoretic properties* which characterize structures preferred in the domain can be exploited.

Given $mcs(C)$ and E, the *structural similarity* of any class C of graphs is denoted by $\sigma(C) = E(mcs(C))$ in the sense of Börner et al. (1993).

In order to reduce the number of NP-complete mappings during retrieval, cases are organized and indexed corresponding to their structural similarity. Applying clumping techniques, a case base CB is partitioned into a finite number of case classes. Each class consists of a set of graph-represented cases and is indexed by $E(mcs(C))$, called its *representative*.

Retrieval from a Structured Case Base. Assuming that a case base CB with $N = |CB|$ is partitioned into $k = \lfloor\sqrt{N}\rfloor$ case classes $CC_i, i = 1, \ldots, k$ each represented by its unique representative $E(mcs(C_i))$, retrieval proceeds in two steps:

1. Determine the maximal similar representative to the given problem g as well as the corresponding case class.
2. Determine the most similar case in the selected case class.

The retrieval result is a set of cases denoted by $S_{i^*} \subseteq CB_{i^*}$ were, for all cases $g^j_{i^*} \in S_{i^*}$, $mcs(\{g^j_{i^*}, g\}) = \mathsf{match}(g^j_{i^*}, g)$ holds. That is, the number of vertices and edges shared by a graph in S_{i^*} and the problem g are identical.

In such a way, the number N of comparisons can be reduced to $2\sqrt{N}$ matches in the best case. Note that the maximal similar representative/case determined in the first and the second retrieval step may not be unique. An appropriate *selection operator* has to be applied or user interaction is required to come up with a candidate case that can be used for adaptation.

Adaptation. The selected cases are proposed one after the other to a selection algorithm, or to the user, who selects the most suitable one.

In general, the number of vertices and edges constituting a problem is smaller than those representing a solution. Hence, a selected case may correspond to the problem solution itself. However, in building design, solutions are hardly ever identical and the selected case has to be adapted to solve the problem. Due to the lack of appropriate adaptation transformations, MACS realizes adaptation by transferring case structure resulting in a supplemented problem; i.e. a solution. The function adapt for a problem g and a selected case, $g'_{i^*} \in S_{i^*}$, uses the result of $h = \mathsf{match}(g, g'_{i^*})$; i.e., the list of matching vertices of g and g'_{i^*}. The *adapted graph* (solution) is generated from graph g by adding all walks in g'_{i^*}, which do not have an isomorphic mapping in g

but which begin and end with vertices of h. These walks may be sequences of edges which are incident with vertices not corresponding to a vertex of g excluding the begin and end vertices.

8.6.4 CA/Syn

Conceptual analogy (CA) is a general approach that relies on conceptual clustering to facilitate the efficient use of past cases in analogous situations (Börner 1995a). CA divides the overall design task into *memory organization* and *analogical reasoning*, both processing structural case representations. In order to ground both processes on attribute value representations of cases, a compile and a recompile function need to be defined.

Compile and Recompile. The function compile guarantees the uniform transformation of an attribute value represented case into its structural normal form; i.e. a tree. Especially suited for the design of pipelines, compile maps outlets into vertices and pipes into edges. Inversely, recompile maps vertices and edges into outlets and pipes. Geometrical transformations, such as translation and rotation, are considered. Representing the main access by a square, outlets by circles, interconnecting points by circles of smaller size, and pipes by line segments, Figure 8.7 (left bottom) illustrates six cases representing pipe systems. Each of them shows a tree like structure. The main access corresponds to the root R, outlets correspond to leaves L. Cross-points of pipes or connections of pipe segments are represented by internal vertices I. Pipes correspond to edges. Each object is placed on the intersecting points of a fixed grid (not shown here) and can be uniquely identified by its x and y coordinates and its $type \in \{R, I, L\}$. Pipes connect objects horizontally or vertically. Thus, a case can be represented by a set of vertices and a set of edges representing connected_to relations among these objects.

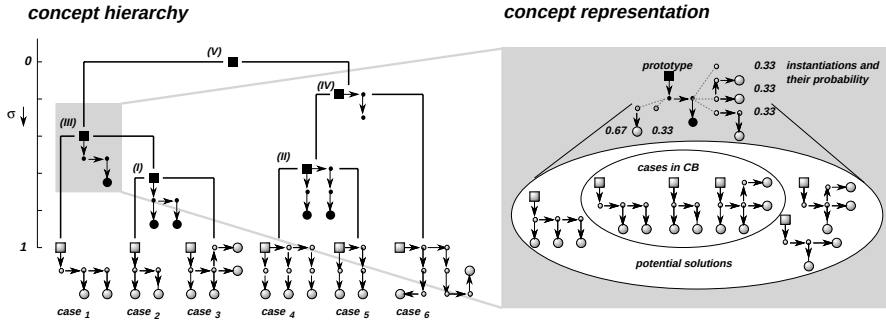


Fig. 8.7. Concept hierarchy and concept representation

Formally, a structurally represented *case* $c = (V^c, E^c)$ is a tree. A *case base* CB is a finite set of cases. A typical design *problem* provides the main

access, the outlets, and perhaps some pipes; i.e., it is a forest. A *solution* of a problem contains the problem objects and relations and eventually adds intermediate vertices from past cases and provides the relations that connect all outlets to the main access.

Memory Organization. Memory Organization starts with a case base CB providing a significant number of cases as well as a structural similarity function σ.

To explain memory organization, some basic definitions will be given first. A *case class* CC is a non-empty subset of CB. The (unique) maximum common, connected subgraph of the cases in CC containing the root vertice is denoted by $mcs(CC) = (V^{mcs}, E^{mcs})$. The structural similarity[3] is defined as $|E^{mcs}|$ divided by the total number of edges in the cases of CC:

$$E\sigma(CC) = \frac{|E^{mcs}|}{|\cup_{c \in CC} E^c|} \in [0,1]$$

Given a case base and a similarity function σ, a *case class partition* CCP is a set of mutually disjoint, exhaustive case classes CC:

$$\begin{aligned} CCP \quad = \{ \quad & CC_i \mid \bigcup_i CC_i = CB \ \wedge \\ & \forall i \neq j (CC_i \cap CC_j = \emptyset) \ \wedge \\ & \forall i \neq j ((c_1, c_2 \in CC_i \wedge c_3 \in CC_j) \\ & \rightarrow \sigma(c_1, c_2) \geq \sigma(c_1, c_3) \wedge \sigma(c_1, c_2) \geq \sigma(c_2, c_3)) \\ \} & \end{aligned}$$

A *case class hierarchy* CCH is the set of all partitions CCP of $CB = \{c_1, .., c_N\}$:

$$CCH = (CCP^0, CCP^1, ..., CCP^{n-1})$$

Memory organization starts with a set of cases $CB = \{c_1, .., c_N\}$ represented by trees. Nearest-neighbor-based, agglomerative, unsupervised conceptual clustering is applied to create a hierarchy of case classes grouping cases of similar structure. Clustering starts with a set of singleton vertices representing case classes, each containing a single case. The two most similar case classes CC_1 and CC_2 over the entire set are merged to form a new case class $CC = CC_1 \cup CC_2$ that covers both. This process is repeated for each of the remaining $N - 1$ case classes, where $N = |CB|$. Merging of case classes continues until a single, all-inclusive cluster remains. At termination, a uniform, binary hierarchy of case classes is left.

Subsequently, a concept description $K(CC)$ is assigned to each case class CC. The concept represents the $mcs(CC)$ (named prototype) of the cases in CC and a set of instantiations thereof, along with the probability of these

[3] Note that the structural similarity function is commutative and associative. Thus it may be applied to a pair of cases as well as to a set of cases.

instantiations. The probability of an instantiation corresponds to the number of its occurrences in the cases of CC divided by the total number of cases in CC. The *mcs* denotes the structure relevant for similarity comparisons and serves as an index in the case base. The instantiations (subtrees) denote possibilities for adaptation. The probabilities will direct the search through the space of alternative instantiations.

In such a way, large numbers of cases with many details can be reduced to a number of hierarchically organized concepts. The concrete cases, however, are stored to enable the dynamic reorganization and update of concepts.

Analogical Reasoning. Analogical Reasoning is based on concepts exclusively. Given a new problem, it is classified in the most applicable concept, i.e., the concept that shares as many relations as possible (high structural similarity) and provides instantiations that contain the problem objects that are not covered by the prototype (adaptability)[4]. Thus instead of retrieving one or a set of cases, the function classify maps a concept hierarchy $K(CCH)$ and a problem p into the most *applicable* concept $K(CC)$.

Next, the *mcs* of the most applicable concept is transferred and instantiated. Instantiations of high probability are preferred. Each solution connects all problem objects by using those objects and edges that show the highest probability in the concept applied. Instead of *adapting* one or more cases to solve the problem, the function instantiate maps the concept representation $K(CC)$ of a case class CC and the problem p into a set of adapted solutions $S_{CC,p}$. In general, there exist more than one applicable concept. The set of all solutions $S_{CB,p}$ of a CB for a problem p equals the union of solution sets $S_{CC,p}$.

Finally, the set of solutions may be ordered corresponding to a set of preference criteria: (1) maximal structural similarity of the solution and the concept applied, (2) maximal probability of edges transferred, and (3) minimal solution size.

If the solution was accepted by the user, its incorporation into an existing concept changes at least the probabilities of the instantiations. Given that the problem already contained relations, it might add new instantiations or even change the prototype itself. If the solution was not accepted, the case memory needs to be reorganized to incorporate the solution provided by the user.

Figure 8.7 (left) depicts the organization of cases into a concept hierarchy. N cases are represented by $2N-1$ case classes respective concepts $K(CC)$. Leaf vertices correspond to concrete cases and are represented by the cases themselves. Generalized concepts in the concept hierarchy are labeled (I) to (V) and are characterized by their *mcs* (prototype) denoted by black circles and line segments. The representation of concept no. (III) representing $case_1$ to $case_3$ is depicted on the right hand side of Figure 8.7. The instantiation of

[4] Note that the most similar concept may be too concrete to allow the generation of a solution.

its prototype results in $case_1$ to $case_3$ as well as combinations thereof. Given a new problem, the most applicable (i.e., most similar) concept containing all problem objects is determined in a top-down fashion. The set of problems that may be solved by concept no. (III) corresponds to the set of all subtrees of either concrete or combined cases, containing the root vertice.

The general approach of *Conceptual Analogy* has been fully implemented in SYN, a module of a highly interactive, adaptive design assistant system (Börner 1995b). While its compile and recompile functions are especially suited to support the geometrical layout of pipe systems the general approach to structural similarity assessment and adaptation is domain independent. See Börner and Faßauer (1995) as well as Börner (1997) for a more detailed description of the implementation.

8.6.5 Comparison

All three approaches, TOPO, MACS, and CA/SYN, apply structural similarity assessment and adaptation to retrieve and transfer case parts to solve new problems. However, the approaches differ in their focus on different parts of the CBR-scenario. In the following, the strengths and limitations of each approach and their domain specific and domain independent parts are discussed.

First of all, it must be recognized that the definitions of the compile and recompile functions strongly depend on the domain and task to support. The higher the required expressibility of structural case representations the more complex is the graph matching, i.e., the less efficient are retrieval and adaptation. The application of labeled graphs (TOPO) or trees (SYN) allows the inversion of the function compile to recompile. This can not be guaranteed for arbitrary graphs (MACS). Whereas the representation of cases by trees (SYN) guarantees unique *mcs*, this does not hold for graph representations, as in TOPO or MACS. Domain specific selection rules need to be defined or extensive user interaction is necessary to select the most suitable *mcs*. This may be advantageous during retrieval allowing the selection of different points of view on two graphs (the *mcs* and a problem) but may not be acceptable for efficient memory organization over huge case bases.

While TOPO performs no structural retrieval at all, MACS and SYN retrieve a set of cases from a dynamically organized case base. MACS uses a two-level case organization. The lower level contains the concrete cases grouped into classes of similar cases. The upper level contains graphs describing the *mcs* of classes of similar cases. It does a two stage retrieval selecting the case class of the most similar *representative* first and searching in its cases for the most similar concrete case(s). SYN uses a hierarchical memory organization, i.e., a case class hierarchy. Each case class is represented intentionally by a concept representing the unique *mcs*, instantiations thereof, as well as the *probabilities* of these instantiations. Given a new problem, the most applicable concept of the concept hierarchy is searched for.

In order to compare graph representations, Macs and Topo need to apply graph matching algorithms (clique search and backtracking) that are known to be NP-complete. For this reason, Topo uses the Fish & Shrink retrieval algorithm, reducing retrieval to the computationally expensive match between a selected case and a problem. Macs memory organization allows the reduction of the number of matches required to search through N cases to $2\sqrt{N}$ in the best case. Syn's restriction to represent cases by trees reduces expressibility but offers the advantage to match efficiently.

For adaptation, Topo investigates the compatibility of object types and relation types of layouts. The frequency of relations in past layouts is exploited to come up with preferable positions for solution objects. Macs realizes a simple variant of case adaptation by adding all walks of the case which do not have an isomorphic mapping in the problem but begin and end with vertices of their *mcs*. CA/Syn concentrates on efficient structural case combination. Therefore, the approach integrates the formation of hierarchically organized concepts (i.e., concept hierarchies) and the application of these concepts during analogical reasoning to solve new problems. It is unique in its representation of concepts by the *mcs* and its *instantiations* plus *probabilities*. Its definition of *applicability* allows the efficient selection of the most similar concept that is neither too general not too concrete and guarantees the generation of a problem solution. The instantiation of its *mcs* is guided by the probabilities of these instantiations, resulting in a set of solutions that may be ordered corresponding to a set of preference criteria.

8.7 Structural Adaptation by Case Combination

Eadocs[5] is an interactive, multi-level, and hybrid expert system for aircraft panel structures (Netten et al. 1995; Netten 1997). The primary design objective is to find a feasible and optimal concept for a panel design.

The structure of a design defines the set of components, their configuration and parameter values. The configuration defines the set of design variables that can be retrieved and adapted. Parameter values should be assigned within the context of a configuration. Retrieval and adaptation of design structures should, therefore, be distinguished for configurations, components and parameters.

In many applications, specialist operators are only available for parameter adaptation. Other approaches are necessary to adapt configurations. The Eadocs system adapts the configuration, components, and parameters by combining cases.

[5] Expert Assisted Design of Composite Sandwich Panels

8.7.1 Required Functionality

EADOCS' task is to support the conceptual design phase. In this phase, a designer specifies the initial requirements, objectives, and preferences. The system suggests, evaluates and modifies alternative solutions for the configuration and discrete design parameters. Initial specifications are elaborated from EADOCS' conclusions or a designer's new input. The optimum panel concept is the starting point for more detailed (numerical) analysis and optimization.

Conceptual design is regarded as an innovative reasoning process for configuration and parametric design. Plans for designing components are not available. Only partial models for evaluating behavior are available.

EADOCS divides the iterative design process into four subsequent phases. Each phase solves a specific and more fine-grained iterative step that results in a set of sub-solutions. A sub-solution is an alternative starting point for the next phase. The level of detail, type of design decisions, and type of reasoning changes with each phase (Netten et al. 1995).

Modifying the R^4 CBR-cycle introduced by (Aamodt and Plaza 1994), Figure 8.8 depicts the four phases in bold. The first phase selects a set of prototype solutions and configures them qualitatively. In the second phase, case-based reasoning is applied to solve two tasks: Firstly, complete case solutions are retrieved to generate quantitative conceptual solutions. The second task is to adapt the retrieved concepts by integrating other case components. The third phase revises the parameters heuristically, while the fourth phase optimizes the concepts numerically.

The case base covers only a small part of the problem and design spaces, which is also biased by previous design intentions. It is unlikely that a case is completely similar to a new problem, and their behavior is not representative for the feasibility and optimality upon reuse of their solution (Netten and Vingerhoeds 1997). Additional information is required to guide retrieval.

EADOCS provides this information in the first phase in which the best solutions are selected and configured into a set of prototypes. In the second phase, it is assumed that the prototype defines the design space of feasible and optimal configurations from which conceptual solutions can be retrieved. If no case can be retrieved for the best prototype, the solution is relaxed to the next best prototype.

8.7.2 Sandwich Panel Structures

Thin-walled skin panels are applied as components of semi-monocoque aircraft structures. A skin alone cannot provide sufficient stiffness. Several types of panels can be configured to provide additional stiffness to the skin, such as a sandwich panel. A sandwich panel has a core of honeycomb or foam material, sandwiched between the skin and an additional inner skin. Figure 8.9 gives an example of a symmetrical sandwich panel with skin laminates of

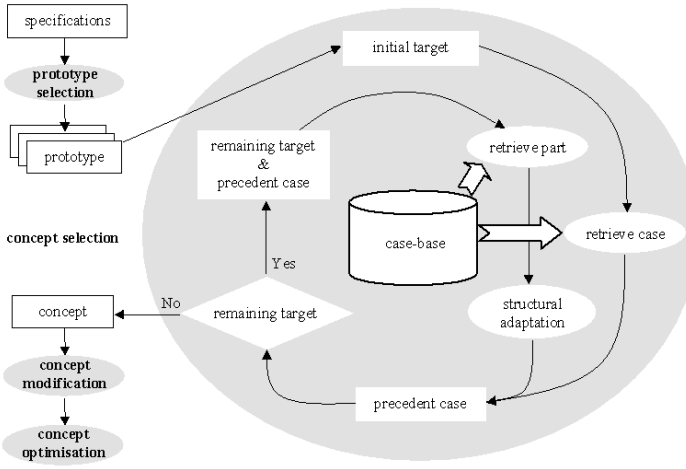


Fig. 8.8. EADOCS design process

three layers and a honeycomb core. The skins can be made out of light alloy sheet material or as composite laminates. A laminate is made out of several layers and a layer is made out of thin plies of fiber reinforced material.

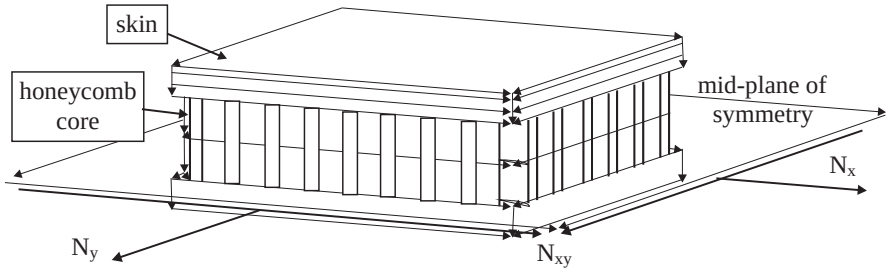


Fig. 8.9. Example of a sandwich panel

The configuration defines the type of panel and the lay-up of each laminate. A prototype is an abstract representation of a panel, and only defines the types of panel, material, and laminates a concept may be selected from. The concept defines all laminates in detail. The dimensions of the panel, core or stiffeners are continuous parameters. For each layer, discrete design pa-

rameters are defined for the type of fiber reinforced plastic (FRP) material, the number of plies, and the orientation of the plies.

A sandwich panel has one of the simplest panel configurations. Other types of panels can be configured, for example, with longitudinal stiffeners joined to the skin. Each stiffener is composed of one or more laminates.

The most important function of a panel is to carry aerodynamic and structural loads. A loading condition is defined as a potentially critical combination of normal and shear (N_{xy}) loads. A normal load is a combination of tension or compression loads in the longitudinal (N_x) and transverse (N_y) panel directions. Typically, several loading conditions are specified for the most severe operational conditions.

Behavior of panels is modeled in terms of stress, strain, strength, stiffness, cost, and weight. The strength is primarily a function of the strength, stress, and strain of individual plies. The stiffness is a function of the panel components and their dimensions. Behavior is analyzed with existing numerical techniques, which require a completely defined concept as input. Prototypical behavior is represented qualitatively as categories of behavior (Netten and Vingerhoeds 1997).

A designer specifies an initial problem by a set of loading conditions and panel dimensions. Objectives are defined for optimality and feasibility, while preferences may be defined for the prototype solutions. For feasibility, panel behavior should withstand each of the loading conditions. For optimality, the cost and/or weight of the panel should be minimized.

8.7.3 Cases and Case Retrieval

EADOCS has an object oriented data structure. Classes are defined at several levels of abstraction for function, behavior, and structure. A design case is an instance of a design problem with objects, or parts, for these classes. Figure 8.10 gives an example of a sandwich panel with a symmetric laminate of 4 skin layers.

All objects are stored in the case with their relations. The buckling behavior relates to the panel, while stress and strain behavior relate a loading condition to a specific layer. The ordering of layer objects defines the lay-up of a laminate and is defined by a layer number. Generic relations about configuration, such as the relation between a skin laminate and the core, are not defined in cases.

Cases are indexed on three levels. A case part is indexed by its features as an object of its class, while the class is indexed by its abstract classes. A case part is also indexed by its relation to other objects as a part of the case. The complex indexing hierarchy enables the retrieval of features, parts, and complete cases. The current implementation of EADOCS indexes features onto discrete values, which is sufficiently accurate when only a few cases can be retrieved.

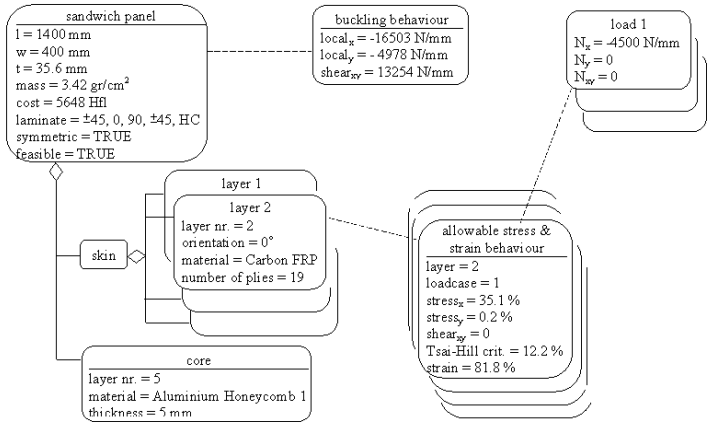


Fig. 8.10. Example of a sandwich panel case

The case base is consulted after selection of a prototype from phase 1 (see Figure 8.8). The initial target restricts retrieval to cases with the configuration of the selected prototype. The initial target for retrieval is defined by objects, or sub-targets, for the specifications and the selected prototype. Each sub-target is indexed in a similar way as its counterparts in the case base. The similarity is measured locally between case parts and sub-targets, and globally as the aggregate similarity of a case for the target.

8.7.4 Case Combination

One case is retrieved that best matches the initial target. Local similarities identify which initial sub-targets are satisfied by this precedent case. Repair of functional or behavioral requirements usually requires the adaptation of several components. Unfortunately, sub-targets for functional and behavioral requirements cannot be causally related to components. Specialist operations are available for the local adaptation of some layer parameters. These adaptations are only useful after adaptation of the laminate lay-ups. Additional information for configurational adaptations may be retrieved from the case base. Searching the case base again for the remaining sub-targets only will not reveal the necessary adaptations.

Predicting and adapting behavior for the remaining sub-targets is essentially different from retrieving a solution to the initial target. Here, it can be assumed that designs with a similar structure will behave similarly as well. Predictions and adaptations can be retrieved from cases with a structure similar to the precedent case. Combination of new components requires four steps:

1. A new target for retrieval is defined by the structure of the precedent case and the remaining sub-targets.
2. Cases are retrieved for the new target. For these cases:
   a) Behavior of the precedent case is predicted from cases with a similar structure.
   b) Adaptations can be retrieved from the differences in functionality, behavior, and structure of cases from 2a.

The assembly of components is defined in the structural model. This model is also applied for indexing and can now be searched for other components. The similarity of a new case to the new target identifies whether features or components should be substituted or inserted. Not all differences in step 2b should be retrieved for adaptation. Some of these differences are non-conservative adaptations; i.e., adaptations that violate the feasibility of previously satisfied requirements. Usually, a small set of allowable or conservative adaptations can be defined in conjunction with the modifications of phases 2a and 2b. In EADOCS, the number of plies in a layer is never reduced, and layer orientations and materials are never changed but inserted in a new layer.

The major advantage of this adaptation approach is that new structural solutions can be generated automatically. The operations for steps 1 and 2 are relatively simple, require little search and little domain knowledge. Their success depends primarily on the coverage of the case base; i.e., the existence of cases with similar structures that have been designed for different purposes. Experiments with EADOCS have shown that, even for a small case base, design improvements can be quite significant. A large number of possible loading conditions can be covered by only a few cases, and combination of these cases strongly increases coverage (Netten 1997). The retrieved structural adaptations are rudimentary but can be revised. Many such discrete adaptations, however, could not have been suggested from heuristics or numerical optimization routines (see also Bladel 1995).

8.8 Discussion and Summary

This chapter started with a general characterization of design tasks. Concentrating on *innovative* and *creative* design tasks, we gave an overview of applicable reasoning methods and case-based design systems. The domain of

architectural design was used to illustrate the general problems that have to be solved in order to provide efficient design support. Finally, we presented approaches to complex case retrieval, to structural similarity assessment, and to structural adaptation.

To conclude, we contrast approaches for *compositional case adaptation* used for automating *routine design* or *configuration* tasks, as introduced in Chapter 6, with approaches for *structural adaptation* aiming at the interactive, assistant-like support of *innovative and creative design* tasks. Finally, the importance of adequate user interfaces and its influence on future research in CBD is discussed.

Compositional and structural case adaptation. Adaptation is central to CBR systems for configuration (i.e., routine design) as well as for innovative and creative design. It can work by assimilating parts of different cases to meet a new specification.

In configuration, a new specification is given by a well defined set of configurable components, a set of constraints that the final configuration solution must satisfy, and configuration operators that encode valid component configurations (cf. Section 6.4). The term *compositional adaptation* refers to the retrieval, adaptation, and subsequent composition of multiple cases (Redmond 1990; Sycara and Navinchandra 1991). Additional knowledge is employed to repair edges of solution chunks such that they work together as a whole.

Innovative and creative design tasks differ in that constraints or repair knowledge is not available in general. *Structural adaptation*, as introduced in Section 8.6, refers to the transfer and completion of the most specific common structure shared by a class of structurally similar cases. Exclusively, cases of high structural similarity are combined.

However, in both approaches adaptation is compositional rather than rule or operator based. Both approaches share the risk that locally optimal case components from different cases and, thus, different contexts will not produce globally optimal solutions when combined.

Adequate human-computer interfaces and future research in CBD. As discussed in Section 8.3, adequate user interfaces are an indispensable requirement to built effective assistance systems. Therefore, one major direction of research is appropriately designed human-computer interfaces that are easy to integrate into the workflow of designers.

In architectural design, CAD tools are very powerful in managing exact numbers and measurements. They provide less support during the abstract or *sketchy* phase at the beginning of a design process in which conceptual decisions are made and major constraints are established. Aiming at an efficient support of the early stages of design development, *Virtual Reality* techniques and fast computer graphics can offer new ways of human-computer interaction as well as efficient possibilities to support, e.g., architectural design. Multiple building partners at different locations can virtually enter the design and experience the space that gets created. Multimodal interaction directly

in three-dimensional space, using two hands and audio, enables the user to formulate design ideas in a much more intuitive way. Using AI techniques, so called interface agents can be provided that are trained by each user to adapt to their individual preferences. They may support navigation and manipulation in 3D as well as retrieval, adaptation, and evaluation of design solutions.

A promising research result in this direction is the spatial modeling tool SCULPTOR (Kurmann 1995; Kurmann 1997). The tool and the agents (a *Navigator*, a *Sound Agent* and a *Cost Agent*, are implemented so far) that are connected with it allow direct, intuitive, and immersive access to three dimensional design models. Through interactive modeling in a virtual space, an easy way of generating and manipulating architectural models is enabled. SCULPTOR's connection to IDIOM (Smith et al. 1995; Smith et al. 1996), a tool that supports spatial configuration using previous designs in domains such as architectural design, circuit layout or urban planning, results in a design system that provides extensive design support via a highly interactive human-computer interface.

The design of intuitive and efficient human-computer interfaces will have strong implications on the knowledge representations used (they have to support graphical interaction as well as structural retrieval and adaptation) and the reasoning mechanisms applied to support design tasks.

Acknowledgments

The author thanks Jörg W. Schaaf for providing Section 8.5 and Bart Netten for composing Section 8.7. Carl-Helmut Coulon and Elisabeth-Ch. Tammer deserve thanks for commenting on Sections 8.6.2 and 8.6.3, respectively. Hans-Dieter Burkhard, Mario Lenz, Michael M. Richter, and Brigitte Bartsch-Spörl gave many critical comments and improvements to the chapter. Nonetheless, the paper reflects the authors personal view on case-based design research.

The research within the joint project FABEL was supported by the German Ministry for Research and Technology (BMBF) under contract no. 413-4001-01IW104. Project partners in FABEL have been German National Research Center of Computer Science (GMD), Sankt Augustin, BSR Consulting GmbH, München, Technical University of Dresden, HTWK Leipzig, University of Freiburg, and University of Karlsruhe.