The Knowledge Engineering Review, Vol. 12:1, 1997, 59–89

Case reusing systems—survey, framework and guidelines

ANGI VOSS

GMD—German National Research Center for Information Technology, FIT, D-53754 Sankt Augustin, Germany

Abstract

The discipline of case-based reasoning develops techniques to retrieve and reuse old solutions for new problems. To reuse solutions, several case adaptation systems have been built and many are under development. They deal with different tasks in different domains, but a methodology is still lacking. From an analysis of up-to-date case adaptation systems, this article moves towards a methodology by providing a common framework and some guidelines. As key issues case adaptation techniques, global strategies, and the tailoring of cases are discussed.

1 Survey

Without adaptation, case-based reasoning (CBR) systems are restricted both in scope and application. To reuse cases effectively in new situations they must be adapted to account for differences between the retrieved case and some new problem. Research in the field has produced a diversity of adaptation techniques to cope with a wide range of tasks, domains and knowledge sources. Scientific consolidation and commercial success of CBR will crucially depend upon finding guidelines that help system analysts and system engineers to ask the right questions about the domain and the task in order to design the right kind of CBR system.

This article analyses several systems that reuse cases in a sophisticated way. All systems are operational, though most are research prototypes, and all are objects of active research[1]. Thus the reader will get an up-to-date view of developments in the field of case reusing systems. The article does not stop with a mere survey. An innovative framework is developed that addresses major issues in the design of a case reusing system, and all systems will be presented according to this framework. From the design decisions encountered in the systems, first guidelines are extracted. They address questions like: Why are cases useful? Why should they be reused? What are the alternatives to case-based reasoning? Why is CBR better? The answers may help a reader who is confronted with a new application to decide whether cases could profitably be reused in that application and how to arrive at the central design decisions for building a CBR system.

The article approaches its subject matter iteratively. As an introduction to the advantages of CBR and the problems encountered, some key issues from the systems to be presented are extracted. Grounds for the framework are prepared by briefly consulting three survey articles from the literature. The next section is devoted to the reuse of a single case, and the following one to the reuse

[1]The choice of systems is historic. First contacts to system designers were established via questionnaires that were distributed to the participants of the first international workshop on case-based reasoning, ICCBR'95 (Veloso and Aamodt, 1995). Intensive discussions and interviews followed at the conference and at a special session on case adaptation (Voss et al., 1996). First conclusions were published in Voss (1996a,b). Further information was obtained in preparation of a workshop on case adaptation at the European AI conference, ECAI'96.

of more than one case. Both are subdivided into framework, systems, and guidelines. The last section considers aspects that are common to both, especially the choice of cases.

2 Preview of systems: motives for and problems with CBR

Table 1 indicates the scope of the CBR systems selected for closer analysis. Like knowledge-based systems, they can be characterized by the task, or kind of problems they solve, and by the domain, which provides the knowledge and cases. Tasks are usually divided into analysis tasks, like classification, diagnosis or decision support, and synthesis tasks, like planning, configuration or design. Typically, the search space for synthesis tasks is larger, because solutions have to be constructed from multiple components, while for analysis, solutions can often be selected from a predefined set. The CBR systems selected cover a broad range of domains and all kinds of tasks. There is a focus on synthesis where reuse can be expected to be more sophisticated because of the large search space.

The following preview of the systems exhibits reasons why cases are reused, what the advantages are with respect to other approaches, what kinds of problems are encountered, and how they can be tackled. The systems will be considered in more detail in sections 4 and 5. For a thorough understanding, the reader should consult the literature.

2.1 Steps in case-based reasoning

Why retrieve cases? To take an example, classification is the kind of task most commercial case-based reasoning systems support. As in the INRECA system (Auriol et al., 1995), cases are represented as lists of attributes and values, and there is one distinguished attribute for the class of the case. Given an unclassified case, how can the class be identified? Typically, one would apply some clustering technique to the set of cases to obtain a discrimination tree. Each node in the tree discriminates between the values of one attribute until, at the leaves, the class is determined. Problems with this machine learning approach arise when the attribute values cannot be obtained in the right order or when some of them are entirely unknown. As an alternative, one could compare the incomplete query case directly with the other cases and fetch the most similar one (e.g., the nearest neighbour). This is case-based classification[2].

Table 1 Types of tasks and domains covered by the selected adaptation systems

Systems	Analysis			Synthesis			Domain
	Classification	Diagnosis	Decision support	Planning	Configuration	Design	
INRECA	X	X	X				e.g. travels
MoCAS		X					machines
CHESS			X				chess
Déjà Vu				X	X	X	software
EADOCS						X	aircraft panels
AAAO, ToPo						X	buildings
IDIOM					X	X	buildings
Composer				X	X		assembly
RESYN/CBR				X			molecules
PARIS				X			manufacture
PRODIGY/ANALOGY				X			transport, routes
CAPlan/CBC				X			manufacture

[2]Nonetheless, case-based classification systems may use discrimination trees to speed up search time—in cases where the discriminating attributes are known.

Why reuse cases? For classification, case retrieval is sufficient; there is no need to adapt the case for further reuse. A slightly more difficult task is decision support, where on the basis of existing cases an offer shall be made. For instance, the customer supplies some requirements, and the system shall offer some travelling arrangements. Using heuristic knowledge one would probably first apply abstraction rules to determine a candidate class of travels, and then refinement rules to adjust it to particular requirements. The problems with such rule-based approaches are known: how to justify the rules, how to acquire and how to maintain them? Using cases from previous travelling offers, one would not store a case for each potential combination of requirements. Instead, one stores representative cases and adds rules for adjusting them to the particular requirements at hand. This case-based approach is often more acceptable than pure rules, since proposing a similar case and applying a few repairs seems to be self-explanatory and more intuitive to many humans. Most commercial case-based reasoning systems offer this kind of case reuse by simple attribute adaptation rules. Again, INRECA is a good representative.

Why retain cases? Another kind of decision support is offered by the CHESS system (Kerner, 1995). It analyses and evaluates a CHESS board position. For that purpose, it has a library of fragments of game positions together with explanations, which was acquired from experts. The system compares the given board position with the fragments and adapts their explanations. As a drawback, the system does not improve with use. Therefore the results of the analysis can be retained as new cases, and be retrieved and reused together with the manually acquired fragments.

Why revise cases? The design of composite materials, like fibre-reinforced composite panels, starts with requirements on desired behaviours and preferred components. There is a small set of possible components, but they have a large set of parameters influencing the behaviour. Numerical algorithms exist, but they are expensive and can only be applied to simulate and correct sufficiently instantiated combinations of components. As there are no heuristics available for producing such combinations, one currently has to follow an expensive generate-and-test approach: the human designer, from his experience, generates an instance and applies the expensive algorithms, iterating until a satisfactory solution is found. In the system EADOCS (Netten et al., 1995), cases are used to store the experience. A most similar case is retrieved to obtain a combination with restricted parameters, and qualitative simulation rules are used to adapt it. The result is still tentative and has to be checked and possibly corrected by the numeric algorithms. Thus, an externally revised and approved version is obtained that can be retained as a new case.

2.2 How to adapt cases?

Adaptation is the major step in case reuse. It can be approached in different ways.

With poor knowledge: heuristic substitution and structure transfer The systems mentioned so far, INRECA, CHESS, EADOCS, all perform adaptation by heuristically substituting some values in the case retrieved. This approach to adaptation is often insufficient if the cases are complex structures, like a layout of a large building. To reuse it, or parts of it, in another context, cannot be done by heuristically adjusting a few values. Instead, for instance in the system TOPO (Coulon, 1995), substructures are extracted, copied and fit to the context of a new building.

With deeper knowledge: tools and problem solvers The systems mentioned so far performed heuristic adaptation. There is no guarantee that the result will be correct. They do so because they operate in domains where only heuristics are available. But case-based reasoning is also applied in domains where tools or problem solvers exist. Their search space is so huge and sometimes NP-complete that starting from a good guess is necessary. The good guess is the case retrieved, and for adaptation the tool or problem solver is applied. An example is the system AAAO (Adami, 1995). Given a layout of spaces, it places the columns according to statical and aesthetical constraints. Columns are implemented as intelligent agents that negotiate and move around until all are satisfied

with their positions. This process can be accelerated enormously by starting with a good initial distribution of the columns as obtained from a case with a similar spatial layout.

2.3 What if a single concrete case is insufficient?

For some time, case-based reasoning focused on retrieving and reusing a single case. If more cases are retrieved, the first case is tried for reuse, and the others are kept for backtracking when adaptation of the first case fails. Later on case-based reasoning was applied to tasks where reusing a single case would not work. This led to new strategies and to special kinds of source cases.

Abstract cases In model-based diagnosis, a machine is modelled by its components, and their input–output behaviours are modelled by rules. A diagnosis is given together with a causal explanation, which is a trace of rule applications relating the observed symptoms to the diagnosed faults. A forward chaining rule interpreter could perform an exhaustive search to diagnose a faulty machine, but the search space is prohibitively large. To shortcut search, cases could be used as follows: a case with the same symptoms and components is replayed by applying the rules from the case to the new problem. But a large number of cases would be needed. Therefore in the system MoCAS (Bergmann et al., 1994), machines are described at different levels of abstraction, and cases, too, describe faulty machines at different levels of abstraction. By starting with the most abstract cases, the search space becomes tractable. Usually, the most concrete case that can be replayed is still an abstract case. But it will limit the search space sufficiently so that forward chaining can be used to adapt the case.

Partial cases For assembly sequence planning, a set of components is given that can be assembled in various ways to obtain different products. The problem is that the parts must be assembled in particular order. When there are many parts and many ways to combine them, one would need too many cases to cover all possibilities, even if the cases can be adapted. Therefore, in the system COMPOSER (Purvis and Pu, 1995) cases contain only assembly plans for very few components. To develop an assembly plan for a larger product, multiple, overlapping cases are retrieved and simultaneously adapted. Assembly plans are represented as constraint networks with a consistent solution and adaptation is performed by a constraint repair algorithm.

Subcases To design an apartment, it would be inefficient to adapt the complete plan of another apartment. Human designers select and reuse pieces from different apartments, but they like to have the context provided by an apartment. In the system IDIOM (Smith et al., 1995), each apartment constitutes a case, which is divided into subcases that can be retrieved and reused individually.

Composite cases A route through a large city could be planned from scratch using a map of the city. But the search space is tremendous, and one should better reuse cases with routes planned previously. The problem is that only parts of these cases will be useful in planning a new route. Storing a route in fragments is no good solution, because practically any fragment may be useful. Statically splitting a case into disjoint subcases is inadequate, because subcases may overlap. Instead, in the system PRODIGY/ANALOGY (Haigh and Veloso, 1995), the route is stored as a whole but indexed by all its intermediate goals (or cross-sections). The indexing scheme superimposes a flexible, dynamically interpreted subcase structure upon a case.

Hierarchical cases Cases contain solutions. If the task is classification, the cases specify classes; for design, they contain designs, for planning they contain plans. Therefore, if a complex problem must be decomposed, cases with decompositions can be used. In the system DéJà Vu (Smyth and Keane, 1995), two kinds of cases are used, cases that decompose a problem into subproblems, or subproblems into even finer subproblems, and cases that actually solve a small subproblem.

Strategies for case-based reasoning? The steps to be performed for case-based reasoning, retrieve, reuse, revise, and retain, are straightforward if only a single case is reused. But if multiple cases are reused, there are more possibilities. Cases can be retrieved individually or together, and they can be

adapted individually or together. Before retrieval, the problem can be decomposed, or decomposition cases can be used, and after adaptation, solutions may have to be combined. These options lead to different strategies for case-based reasoning.

3 Flashbacks

Three brief flashbacks at the literature shall prepare the grounds for a framework for case reuse.

3.1 Steps in case-based reasoning

Case-based reasoning is a complex process that consists of several steps. However, a standard decomposition and vocabulary has not yet evolved (c.f. Kolodner, 1993; Riesbeck and Schank, 1989; Slade, 1991; Watson and Marir, 1994; Aamodt and Plaza, 1994, to cite a few milestone articles). The view assumed here is close to Aamodt and Plaza (1994). A few deviations reflect terminology as established in the FABEL project. FABEL was a large German project on the use of CBR to support the complex task of building design. As shown in Figure 1, a *problem* gives rise to a *query case* for which a *source case* is *retrieved* from the *case base*. It can be *reused* to solve the query case. External validation can be applied to *revise* the case, and then it can be *retained* and added to the case base. All steps can be decomposed further. This part focusses on reuse. It may require to *match* source and query cases in order to find out their similar components. Then deviating or missing parts can be *extracted* from the source case and be *adapted* to fit the query case. Among these steps, adaptation is the most critical one. Therefore, it is often used to mean the whole process of case reuse.

In Aamodt and Plaza (1994) all steps are refined in detail. Compared to Figure 1 there are a few renamings like new and old case instead of query and source case, solved and tested/repaired case instead of adapted and confirmed case. More importantly, they do not subsume matching under reuse but under retrieval. In fact, matching is sometimes performed already during retrieval and then referred to again during reuse, but often retrieval is based on a faster, more superficial comparison than the deep match required for adaptation. So the borderline between retrieval and reuse is not so strict.

3.2 Characteristics of single-case reuse

The reuse of cases has been a challenging task for quite a while. Often the use of the retrieved cases has been left to the user. Sometimes there was not enough knowledge for automatically adapting a source case, or it was too expensive to acquire, or automatic adaptation was too inefficient or too

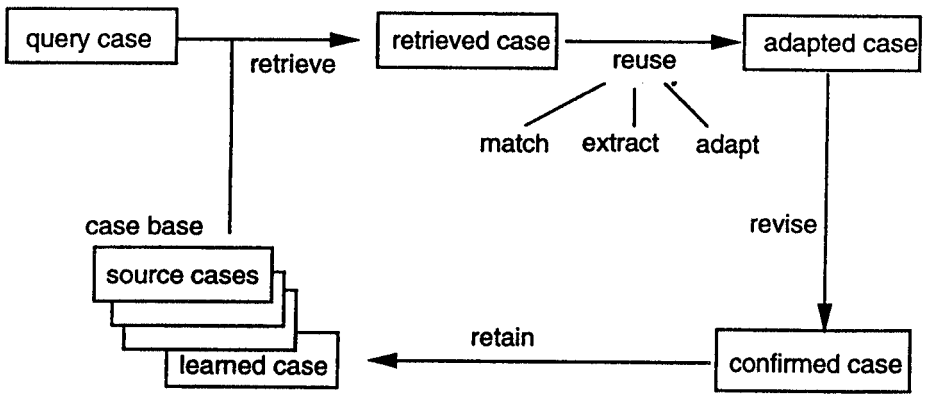


Figure 1 Steps in case-based reasoning

ineffective. Nonetheless, there have been attempts at building case-adapting systems, and there have been first attempts to survey and classify them.

Kolodner (1993), in her book, devotes two chapters to "Adaptation methods and strategies" and "Controling adaptation". In the first, she distinguishes:

- The data affected by adaptation: a value or a structure.
- The kind of effect: a substitution or a transformation.
- The method used. Ten alternatives are listed: the substitution methods reinstantiation, parameter adjustment, local search, query memory, specialized search, case-based substitution; the transformation methods commonsense transformation, model-guided repair; other methods: special-purpose adaptation and repair, derivational replay.
- The guidance (or rather knowledge) needed by the method. Six alternatives are given: structural, causal, cases, specialized, ad hoc commonsense.

The first one of these four criteria essentially allows to distinguish simple *value-affecting adaptations* from more sophisticated *structure-affecting* ones. The second criterion is related: typically, values are *substituted* and structures are *transformed*, but some systems do both, affecting values and structures. In practice, such a categorical distinction may be difficult.

One can also distinguish *local* adaptations that are confined to a small part of the involved case, perhaps only one attribute, from *global* ones affecting the whole problem, in particular due to constraints connecting many parts. Local and global adaptations are the endpoints of a continuum.

Also, one should make clear for the first or second criterion whether it is applied to the source case or to the query case.

The criteria on methods and knowledge are related, too, because each method has its special knowledge needs. The alternatives proposed are rather fine-grained.

Kolodner's chapter on "Controling adaptation" discusses how to find what needs to be adapted and how to select a suitable adaptation method. Due to the systems available at the time of writing, only systems adapting the problem by means of a single source case are considered.

3.3 Characteristics of multi-case reuse

A later comparison (Hanney et al., 1995) deals with systems that reuse multiple cases. It applies the following criteria to case adaptation systems:

- The type of task; it can be identification (meaning analysis), prediction and design (meaning synthesis).
- Is adaptation performed at all?
- The number of source cases for solving a problem, which can be one or more.
- Complexity of case solutions, which can be atomic-valued, compound but not amenable to adaptation or compound with parts that can be modified by adaptation processes.
- In compound adaptable systems, partial solutions can be independent or interacting.
- Within a system, four main types of adaptation knowledge are distinguished on the basis of the role they play in the adaptation process: target elaboration, role substitution, subgoaling and goal interaction.

Clearly, this more recent article addresses a new aspect which concerns the integration of case-based reasoning in a larger problem solving context. The analysis of adaptation knowledge reflects the experience that case-based reasoning can be applied to data at different stages of problem solving. It essentially asks for the role an item to be adapted plays in problem solving: is it a problem, a task, a plan, a strategy or a solution? The criteria concerning number of cases, complexity and interactions all indicate that adaptation need not be restricted to a single case and a single problem.

These questions are relevant for the systems considered in this study. But for didactic reasons, we start with single-case adaptation. One could also say that the next section, on single-case adaptation,

provides a micro-view, the following section, on multi-case adaptation, provides a macro-view, and the final section combines both to give guidelines on building case adaptation systems.

In contrast to Kolodner (1993) and Hanney et al. (1995), a few representative, more recent systems are compared that reflect current research. They exist as prototypes and are subject to active research.

4 Reuse of a single case: adaptation

This section proposes a framework for describing systems that reuse a single case. It applies it to the systems selected and extracts a first set of guidelines. The major issue in the design of such single-case systems is the approach to adaptation, because it influences many other decisions. Adaptation remains an issue with case-based reasoning systems that reuse multiple cases. But there, strategic questions of how to handle more than one case move into foreground. Therefore, this section focuses on case adaptation, and the next section on strategies.

4.1 Framework: three types of domain

There are many ways to characterize case reusing systems, even systems for single cases. Other than Kolodner (1993) or Hanney et al. (1995), whose classifications were characterized in section 3.2, the focus here is geared towards knowledge engineering. One of the driving forces behind CBR was the hope that the bottleneck of knowledge modelling and acquisition could be relaxed by the use of cases. This section will show that this hope has to be reduced substantially when cases have to be adapted.

Adapting a source case to a query case is indeed a kind of problem solving. The query case constitutes the problem, the adapted case contains the solution, and the solution shall be obtained with the help of the source case. Assuming this perspective, it is worthwhile to look for existing from-scratch problem solvers or for tools that cover at least part of the problem solving process. If they can be reused for adaptation as they are, they prescribe the case representation and the kind of information that can be reused from the source case for the query case. Otherwise, specially tailored adaptation techniques must be developed. They, too, constrain the case representation: what can be put explicitly into the case representation need not be implicitly inferred by the techniques.

As problem solvers and tools are based on theories about the domain, case adaptation systems should be analysed depending on the theoretical understanding of the domain: is there a complete theory, a partial theory, or none? Related are the (additional) knowledge engineering effort for adaptation, the potential effect to be achieved by adaptation, and the size and required quality of sources cases that can undergo adaptation.

4.1.1 Problem solvers for adaptation

In well-understood domains, there is a *complete theory of problem solving* and there exist from-scratch problem solvers—though they may be prohibitively slow. With them comes a well-defined search space. The source case should help to shortcut exploration of this space and cut down search time. This kind of adaptation shall be called *generative* because a solution can be generated from any starting point:

- There are various ways how information from a source case can influence a problem solver. First, query and source case can be turned into an intermediate point in the search space that acts as a better starting point for the problem solver. This kind of generative adaptation shall be called *solution modification*.
- Alternatively, the problem solver starts from scratch, but information from the source case influences its choices. This variant shall be called *derivation modification* because the path to, or derivation of the solution is reused.
- In its extreme form, called *derivational replay*, the problem solver exactly reproduces the problem

solving steps from the source case[3]. The border between derivational replay and other derivation modifications is hard to draw, and will not be applied here.

In principle, there are two different search spaces, one for solutions and another one for derivations. In practice, the distinction may be blurred. Some systems combine search in both spaces, and especially for planning systems, both views can be applied: on the one hand, a plan is a derivation (of a solution), on the other hand, it is a solution (for a derivation). Thus, derivation modificiation can be viewed as solution modification: It modifies a solution that contains a derivation.

For solution modification, the source case must contain a solution that can be (partially) re-used for the query case. For derivational replay, the source case must contain information about the problem solving trace that can be replayed. For derivation modification, both kinds of information can be used. Furthermore, care must be taken that information which is extracted from the source case and combined with the query case conforms to the problem solver's input conditions so that it can process the case. If the problem solver cannot deal with inconsistencies, it may be possible to remove them before adaptation yielding larger holes for completion; similarly, holes in the intermediate solution may have to be filled tentatively.

If existing problem solvers are applied for adaptation, there is hardly any additional knowledge acquisition effort, and case-based reasoning may achieve sometimes enormous gains in efficiency. Also, requirements on the quality of the case retrieved are not so strong: the lower the quality, the longer adaptation will take, but it will not become infeasible. Last but not least, from-scratch problem solvers can achieve powerful transformational effects on both source and query case.

4.1.2 Tools for adaptation

In less-understood domains, there will be no complete theories as a basis for problem solvers but *partial theories* as a basis for *tools* that can support steps in problem solving. Like problem solvers, tools have special input conditions:

- *Corrective tools*, like constraint satisfaction algorithms, need a complete network and then compute solutions. Constraint relaxation algorithms even need the network together with tentative values for the variables and then apply corrective substitutions; but they might be able to process inconsistent solutions.
- More rarely, there may be *generative tools* that can complete a partial, but consistent solution.
- More often, there may be *assessment tools* for checking a solution. They can be used for adaptation in order to check a tentatively amended query case for correctness. Iterative adaptation may finally lead to a correct solution. This generate-and-test approach is not very efficient.

Tools differ in the kind of effect they can achieve on the source or query case: it can be more substitutional or more transformational in nature. If a tool does not cover the entire adaptation step it must be supplemented with some heuristic methods or with manual correction. Some knowledge may be needed to prepare the source case for the tool or to postprocess its result. The quality of the source case must be better than for adaptation based on a complete theory, because a tool cannot fall back to from-scratch problem solving.

4.1.3 Heuristics for adaptation

In domains without even partial theories neither problem solvers nor tools can be exploited, but case-based reasoning is particularly attractive because it may help to make formerly unsolvable

[3]The distinction between solution modification versus derivation modification is old but has been given different names: Carbonell (1986) speaks of transformational versus derivational analogy, while Cunningham et al. (1994) call it substitutional or transformational adaptation *versus* generative adaptation.

problems solvable at all. Here, heuristics must be used. They may be elicited from experts, or they may automatically be learned from source cases:

- *Heuristic adaptation* has long been restricted to rather local effects on the source case, typically value substitution. More effectful transformational heuristics have been domain-specific.
- Only in the last few years, inspired from the field of analogical reasoning (Carbonell, 1986), more general *structural adaptation* methods have been developed which can achieve more powerful transformations. They require cases with a rich structure, as can be found in many synthesis applications. Structural adaptation assumes that the structure of a case is meaningful. Though constraints and requirements may be unknown, the source cases comply to them and thus contain implicit, compiled knowledge. Preliminary to structural adaptation, correspondences between query case and source case must be established to find out which parts or properties of the cases are similar and which are different, and what information to extract. This information can be modified with heuristics while it is transplanted to the query case.

The outcome of heuristic adaptation strongly depends upon the quality of the source case to be adapted, because there are no compensatory inference capabilities.

4.1.4 Criteria for classifying adaptation

The approach to adaptation is a major decision. It constrains the particular adaptation technique, which in turn constrains the case representation, how the parts to be adapted are determined and what can be achieved by adaptation. These criteria will be applied to the selected CBR systems in the next section.

- *What approach and technique to adaptation is chosen?*
  In her book, Kolodner lists 10 techniques, or methods, for adaptation. For the present purposes, they are too low level for discrimination. Therefore, adaptation techniques are subsumed under a few approaches: *generative adaptation* (solution modification or derivation modification), *tool-based adaptation* (generative, corrective or for assessment) and *heuristic adaptation* (substitutional heuristics or structural adaptation). The adaptation technique constrains or determines the following four criteria.
- *Case representation language*
  In principle, every knowledge representation language can also be used for cases. Cases may contain flat attribute-value lists, they may contain graphs, especially plans and constraint nets, they may contain sets of objects, etc. At a more abstract level, the difference between a description of a *single object* (subsuming feature lists), a *set of objects* (as in a layout) and a *graph* is significant. The case representation language must be processible by the adaptation method. For instance, one cannot apply graph matching procedures to attribute-value lists.
- *Establishing correspondences*
  Correspondences between source and query cases must be established in order to detect differences that can be extracted and reused. The parts to be compared can be determined *statically*, independent of query and source case, or *dynamically* by performing some kind of match or unification. Static distinctions typically split a source case into a problem part and into a solution part, so that problem parts are matched and solution parts are extracted. But in principle, every source case can be reinterpreted dynamically, e.g., by providing another index. So this distinction is rather soft. Of course, source cases can be used more flexibly when they are matched dynamically. Establishing correspondences can already be done during retrieval and is a necesssary prerequisite for adaptation.
- *Data extracted from the source case*
  Essentially, one can copy the *complete* source case or extract *parts* of it. The data extracted contain a *solution* or a *derivation* of the solution. Typically, when parts are extracted, the query case will be adapted, and when the complete source case is copied, this copy will be adapted. If parts are extracted, one can ask further with Kolodner (1993) whether it is a value or a (more

complex) structure. Here 'value' or 'structure' refers to the case representation language mentioned to in the second criterion.

- *Effect of adaptation*
  This criterion essentially depends upon the previous one. One can apply Kolodner's distinction between *substitutional and transformational* effects, and refine the latter to additive, subtractive and restructuring transformations. Since adaptation involves two cases, it must be made clear whether the effect on the source case or on the query case is meant.

4.2 Selected systems

In the following, the selected CBR systems are classified by the approach to adaptation and are studied with respect to the other criteria suggested. Some of the systems actually adapt multiple cases, but if one ignores the different provenience of the parts to be adapted, one can describe the systems as if they were adapting a single case.

4.2.1 Generative adaptation by derivation modification

The systems below operate in domains with a complete theory. They perform generative adaptation by using a from-scratch problem solver and apply it to information about the derivation of the solution. All systems below use a causal model of the domain, either a description of operators with their preconditions and effects, or a description of causal relations between symptoms and diagnoses.

Deriving an abstract diagnosis: MoCAS This system diagnoses faults in technical systems (Bergmann et al., 1994). It was applied to diagnose CNC-machines represented by up to 110 components and 356 attributes. MoCAS does not only identify a faulty component, it also explains it by causal links from the symptoms to the fault. The core idea behind MoCAS is to base retrieval and reuse not only on the solution, nor on its entire derivation, but on an explanation of its correctness. Explanations are generated by a from-scratch problem solver.

The domain model consists of a hierarchy of machine components with part-of relations. A component has input and output ports and its behaviour is modelled by a set of rules relating the states of the ports. Since in this domain there is a high variation of problems, the search space is huge. To constrain the search, components are modelled at different layers of abstraction which are linked by abstraction rules.

A solution is a completely interpreted system with instances of rules relating symptomatic states to a diagnosis. A diagnosis is a component identified as malfunctioning by a derivation of rules. Technically, a solution is represented as a so-called explanation graph. The graph has two kinds of nodes, fact nodes for states of ports or diagnoses and rule nodes for instances of rules. A problem is a partially uninterpreted system, where some components have ports with unknown state. Technically, it is represented as a set of fact nodes.

The problem solver is essentially a forward-chaining rule interpreter that expands the fact nodes of the problem to a complete explanation graph with a diagnosis. It can operate at any layer of abstraction. In order to refine an abstract diagnosis to a concrete one, the rule interpreter is reapplied at the lower layers, but now the search space is considerably reduced by the abstract solution.

A query case is a problem. A source case contains solutions of a problem at some level of abstraction. In a subsequent retainment phase, a concrete solution is automatically transformed to the higher layers of abstraction using the abstraction rules.

Given a query case for retrieval, its facts are abstracted through all layers. To compare the query with the source cases, the problem solver is applied to replay their explanation graphs, starting at the most abstract cases. If the explanation graph can be replayed identically, replay is repeated at the next lower layer. A most similar source case is one with an identical explanation (graph) at the

lowest possible layer of abstraction. It is used to constrain the diagnostic reasoner in producing an explanation graph at the concrete layer. The result is the adapted source case.

Deriving abstract plans: PARIS This system is based on a domain-independent planner (Bergmann et al., 1994). It was applied to plan the manufacturing of rotary symmetric workpieces on a lathe. MoCAS and PARIS must be considered as an attempt to apply the same principle to two different domains, diagnosis and planning. While MoCAS constructs an explanation linking certain symptoms to a diagnosis, PARIS constructs a plan to reach certain goal states from certain initial states.

The domain model of PARIS consists of partial state descriptions and of operators to transform them. Operators exist at different layers of abstraction. The problem solver is a hierarchical planner. It constructs a graph of operators that transform the initial state to the goal state.

The operators correspond to the rules in MoCAS, and the planner to the rule interpreter. The rest is analogous: The source cases in PARIS are plans described at different layers of abstraction. Each layer contains a graph representing a plan. For retrieval, a problem with its initial and final state descriptions is abstracted and iteratively compared with the source cases. This is done by replaying the plans, starting at the most abstract layer. The most concrete plan that can be replayed identically is expanded to a concrete plan. By using source cases at different layers of abstraction, PARIS cuts down the search space.

Deriving concrete plans: PRODIGY/ANALOGY This system is a generic case-based planner that was applied to a logistic transportation domain (Veloso, 1994) and to planning routes in Pittsburgh (Haigh and Veloso, 1995). The system is based on PRODIGY, a generic from-scratch planner. The domain theory is about operators and other static knowledge, such as the map of Pittsburgh.

Source cases are derivational traces of both successful and failed decisions in past planning episodes together with their justifications. Source cases can be indexed multiply so that individual parts of them can be extracted and reused. A query case contains an initial state description in terms of some predicates and the goals to be achieved.

Retrieval returns a part from a source case that covers part of the query. The planner re-interprets the justifications from the source case, reusing successful decisions when the justifications hold true and avoiding them when they failed. Replanning with the domain theory is necessary to bridge gaps between the case part and the query or when the domain knowledge invalidates part of the retrieved solution. For instance, in route planning, a source case may contain segments that cannot be used anymore because the map has changed.

CAPlan/CbC This system is based on CAPlan, a partial-order non-linear planner, and was applied to plan the manufacturing of rotary symmetric parts (Muñoz-Avila and Huellen, 1995). The system was inspired by PRODIGY/ANALOGY, and both systems are quite comparable. In contrast to the logistic transportation domain of PRODIGY/ANALOGY, the domain of CAPLAN/CBC is more structured. There is static domain knowledge about dependencies between goals that is used to accelerate retrieval. As a consequence, the parts of the source cases that are matched are statically determined: initial states and final goals, rather than intermediate goals as in PRODIGY/ANALOGY.

4.2.2 Generative adaptation by solution modification

The systems below apply from-scratch problem solvers not to replay steps to a solution, but to modify or extend a solution copied from a source case.

Plan transformation: RESYN/CBR This system adapts plans to synthesize organic molecules (Napoli and Lieber, 1994). A molecule is represented as an undirected graph of atoms and bonds. There is a class hierarchy for atoms and bonds. Two (invertible) relations exist between molecule graphs. First, molecule *m1* cosubsumes a molecule *m2* if there is a type-respecting subgraph

isomorphism from *m1* to *m2*. Second, a molecule can be reduced to another one. A synthesis plan for a molecule *m* is a chain of reductions leading from *m* to simpler molecules. The cosubsumption and reduction relations are operationalized by graph rewriting rules. Additionally, for each relation (or rule) there is a transformation function. Given a molecule *m* and a rule *r* that could be applied to *m1*, then the function associated with *r* constructs a molecule *m2* such that the relation *r* holds between *m1* and *m2*. The transformation functions are used for adaptation and the rules for matching. Each rule is associated with an estimated cost.

A source case contains a plan to synthesize some molecule. The molecule constitutes the problem part and the plan the solution part of the case. A query case contains a molecule. To compare a source case with a plan for molecule *ms* to a query case with a molecule *mq*, a so-called similarity path is constructed between the molecules *ms* and *mq*. Such a path describes a matching between the two molecules. It consists of a sequence of rules for reverse cosubsumption starting from *ms* and leading, say, to *ms'* and a sequence of rules for the reduction relation starting from *mq* and leading to, say, *mq'*. These paths are constructed iteratively by A* search. As a last step, a cosubsumption between the two open ends *ms'* and *mq'* is established. The costs of the path amount to the sum of the costs of individual rule applications. The source case with the cheapest similarity path is chosen for reuse.

The similarity path guides reuse. Starting from the plan in the source case, for each rule in the similarity path the corresponding transformation function is applied to all nodes in the plan. The last transformation yields a plan for the query molecule *mq*. This is the solution. Compared to the original plan, only substitutional effects took place.

Its task being a kind of planning, one would expect RESYN/CBR to replay derivations. Instead, it treats the plan from the source case not as a plan but as a graph, and transforms it into a solution for the query case. It could do this by blind forward search, but this would be an extremely stupid generative problem solver and its search space is tremendous. Therefore during retrieval, the system constructs a derivation from the problem part of the source case to the query case. The derivation describes a sequence of transformations. By applying them simultaneously to all nodes in the plan, search can be avoided entirely during adaptation.

Agent-based transformation: AAAO AAAO (Adami, 1995) adds construction elements to a spatial layout. So far, AAAO is specialized to MIDI constructions[4]. Given the desired spatial layout of a floor, AAAO performs the first and main step which is to fix the positions of the columns on the floor. AAAO has a from-scratch problem-solving method based on intelligent agents. Design objects (like columns) are realized as active autonomous objects. They watch their environment, evaluate their own position according to a set of constraints, and have a set of reactions to improve their environment or their own position, e.g., moving to a better position, or creating new objects, or even destroying themselves in case of serious conflicts. They work in parallel, there is no overall strategy to create the solution. The constraints concern static requirements of MIDI as well as aspects of aesthetics or usage of the rooms. The process is supervised by a manager, which simulates parallelism and guarantees termination. Otherwise, the manager has no problem solving competences. The solution evolves from the distributed competences of the active autonomous objects which in parallel assess their environment and react according to a rather simple pattern. Figure 2 gives some snapshots of AAAO at work.

With its from-scratch method, AAAO can start with any initial distribution of columns on a floor plan. It can generate the initial column positions randomly or use a set of simple rules. But if the initial column positions are taken from a source case with a similar spatial layout, the process can be accelerated considerably. Another advantage in starting with a good case may be that hidden

[4]MIDI is a special steel frame construction kit, developed by Fritz Haller.

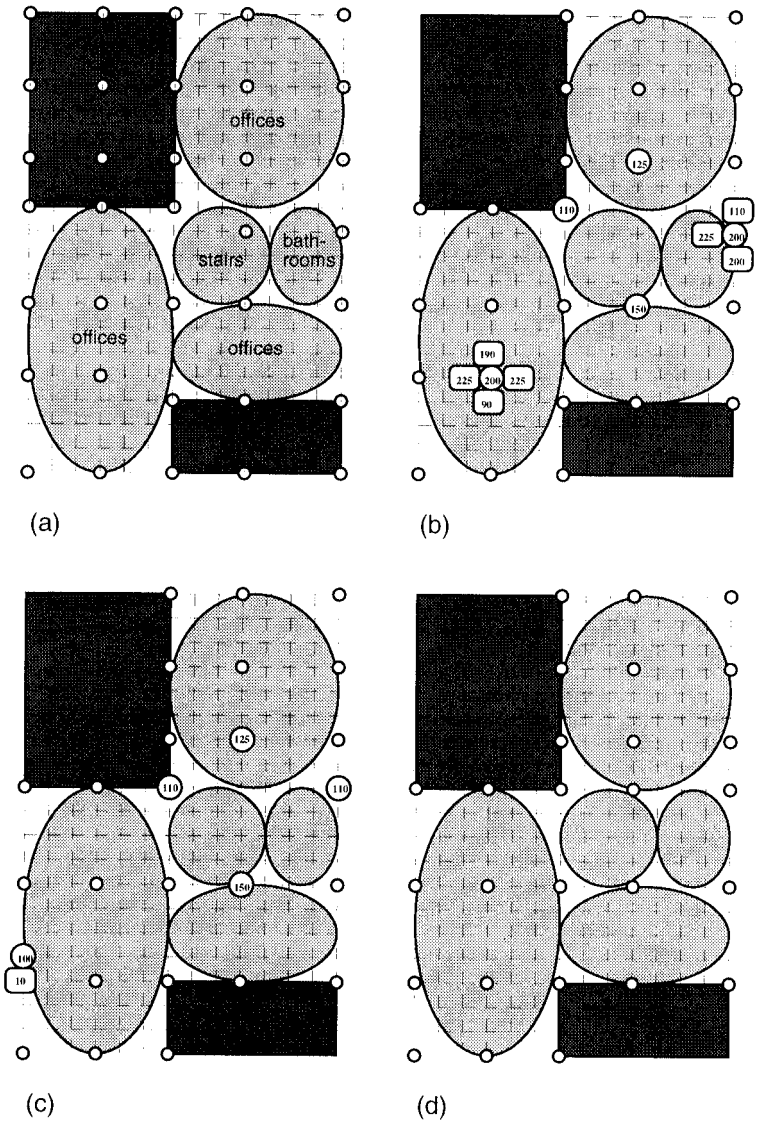


Figure 2 An example to illustrate the AAAO method. (a) Query case with an outline of the building and zones of intended use (indicated as grey ellipses; the dark shades denote areas outside the building). In addition, there is an initial distribution of columns (small white circles) which does not yet satisfy statical and architectural needs; (b) the situation after all columns outside the building have been removed and the others have evaluated their position. For five columns a score of over 100 indicates the violation of hard constraints. Those five columns have then evaluated their neighbouring positions. They will move to the neighbouring position with the least conflict, i.e., the lowest score; in (c) this was done and the columns have once again evaluated their position. There are still serious conflicts for four columns and the process continues analogously to (c), but this time one more column is created; (d) final result, where at least all hard constraints are satisfied

qualities in the source case are possibly preserved during adaptation, thus gaining better solutions without the need to model every aspect of quality.

4.2.3 Adaptation by tools

The systems below use a tool rather than a generative problem solver for adaptation. Both use constraint satisfaction and relaxation algorithms. The algorithms must be supplied with the variables to be constrained, with the constraints to be satisfied, and with tentative values for the variables. This information is extracted from the source case. Actually, the COMPOSER system

operates in a theoretically well-understood domain, but it does not use a generative problem solver, because it can outperform it by repairing a good source case. COMPOSER shows that good source cases in combination with a good tool may be an alternative to generative adaptation.

Dynamic constraint satisfaction: COMPOSER This system was applied to motor assembly and to car configuration problems (Purvis and Pu, 1995). Though these problems can in principle be solved from-scratch, COMPOSER only relies on a tool: a dynamic constraint-repair algorithm.

Source cases contain primitive constraint satisfaction problems and their solutions. They are represented as attribute-value pairs. Attributes describe the constraints, variables, their values, and some additional information useful for comparison. A query case is a partial source case which does not contain the constraints and the values.

During retrieval, a structural match between the variables is established with the help of additional attributes. In the assembly task a mating relation, which indicates the parts to be adjoined, is used for matching. Then a nearest neighbour similarity is computed based on all common attributes. For adaptation, the constraints and their values are extracted from the retrieved case and replaced by matching variables. The result is input into the constraint satisfaction algorithm. It can replace values to repair inconsistencies, and it can introduce new variables or delete old ones. This may achieve limited transformational effects. By supplying a good source case and using an efficient repair algorithm, the approach proved to be better than, or at least as good as, from-scratch problem solving, where first the constraints must be generated and then values had to be assigned to the variables in a consistent way.

Constraint satisfaction with continuous variables: IDIOM This system supports the layout of apartments. The background knowledge allows to enrich a partial layout by constraints (Smith et al., 1995). There is no complete problem solver in this domain. The tool used allows to solve and relax constraints, and especially constraints with continuous variables, as they arise in geometric design. With the method only rectangular spaces and objects can be processed, and to increase runtime efficiency, only linear or simple non-linear constraints are allowed.

Since the architects for whom IDIOM has been built only want to reuse their own designs, which they are sufficiently familiar with, retrieval is not supported and must be performed manually. Therefore, given a partial layout as a query case, the user will add some objects, spaces, constraints, and values from some source case containing a complete apartment layout. He may generate further constraints spontaneously or by applying domain knowledge. Additional constraints are automatically inserted to maintain the topology during the subsequent constraint satisfaction process. The result is a globally consistent layout.

4.2.4 Heuristic adaptation by substitution

The next systems reuse cases without even the help of a tool. They have to rely on heuristics. The first group only changes values in the source case. Actually, in the domain of CHESS there is a knowledge-based problem solver, but it is heuristic in nature. Therefore, one might classify CHESS as a generative adaptation system. The system in the second group, TOPO, transfers structures and achieves more powerful adaptation effects.

Substitution at different levels: DéJà Vu This system configures plant control software (Smyth and Keane, 1995; Smyth and Cunningham, 1992). There is no background knowledge beyond adaptation heuristics, so-called adaptation specialists. They consist of two parts. A precondition checks whether the rule is applicable to a source case, which typically involves some matching of variables. An action part performs the adaptation. Its effect is rather substitutional, though limited transformations may occur. The precondition parts of the adaptation rules can be used to check whether and how easily a source case can be adapted.

DéJà Vu introduced two novelties: an adaptation-guided retrieval method and an integration of case-based reasoning with problem decomposition methods. The developers of the system argue

that not always the most similar source case is the easiest to adapt, especially, when similarity is based on superficial features. As a remedy, retrieval must search for adaptable source cases. In DéJà Vu the knowledge about adaptability is stored in the precondition of adaptation rules. Consequently, retrieval in DéJà Vu not only returns a most adaptable source case, but also the adaptation rules to be applied, and for each a matching of the variables that have to be adapted.

The second novelty results in the introduction of two kinds of source cases. Abstract cases decompose a given problem (or even a subproblem) into subproblems, and concrete cases provide a piece of software solving a subproblem. Source cases consist of a feature set describing the goals to be achieved and a solution. The relation that links an abstract case to other cases that satisfy one of its goals imposes a hierarchical structure on the case base. The abstract source cases in DéJà Vu provide a means to store decomposition knowledge explicitly. This is useful in domains where there is no general, static decomposition knowledge available. Also, retrieval becomes very efficient because traversing the hierarchy is fast.

During configuration, the problem is decomposed into subproblems, and, for atomic subproblems, solutions are inserted. The current state is stored on a blackboard. Given a partially completed plan on the blackboard, a decomposition specialist chooses a next subproblem. A source case is retrieved and adapted, and an integration specialist updates the blackboard.

DéJà Vu is the first system applying substitutional heuristics for a synthesis task. To do that, it had to introduce two tricks or novelties, as they say. Substitution has limited effects and must be applied to small source cases. Solution in synthesis tasks tends to be complex. Therefore, the solutions had to be split into small cases and abstract cases had to be added to control the reuse process: small substitutions in abstract cases effect larger transformations in the composite solution. Also, substitution needs cases of good quality, and this means easy to adapt cases. Therefore adaptation-guided retrieval is a must.

Approximative substitutions: EADOCS This system refines prototypical designs of fibre-reinforced composite panels into initial conceptual designs (Netten et al., 1995). Case-based reasoning is preceded by a selection of the prototype, and it is succeeded by numerical optimization.

The domain knowledge consists of an object hierarchy, of adaptation rules and of approximative behaviour prediction rules. The object hierarchy consists of classes for panels, their components, the design objects as part of the components. Components have behaviours and attributes to characterize the behaviours. Additional attributes characterize the solution. Adaptation rules allow to adjust behaviour attributes, and behaviour prediction rules allow to further constrain the attributes of behaviours in the context of the partially specified components.

Initially, a problem is specified by functionalities to be achieved, preferences and optimiality criteria. From this a prototype is derived. It specifies behaviours to achieve the functionalities and intervals restricting the attributes of the behaviours. This is input to case-based reasoning, which must configure a complete object by selecting a panel type, components, and design elements with concrete values for all discrete attributes. Subsequent numerical optimization concerns the continuous attributes.

The number of different designs in terms of the object structure is relatively small, but the combination of attribute values is huge. Numerical optimization methods are expensive, so that the space of possible solutions cannot be exhausted even approximately. Case-based reasoning shall help to short-cut the search and to reduce expensive numerical optimization by constraining the attributes.

A source case contains a composite object with concrete or constrained attribute values. It is indexed by the behaviours and behaviour attributes of its components, by its solution attributes, and by its components. An associative memory is used that links indexes to components and components to their source cases.

A query consists of the initial problem specification and the selected prototype. Retrieval returns the source case with the most similar components. They are substituted in the prototype and their behaviour attributes are modified by the adaptation rules.

Adapting position explanations: CHESS This system analyses and explains CHESS board positions for educational purposes (Kerner, 1995). Input is a CHESS board position and output is a comprehensive positional analysis. Case-based reasoning is integrated with a knowledge-based heuristic approach.

The domain knowledge consists of three parts. First there is a discrimination tree for so-called basic game patterns. They define small configurations involving a small number of pieces and fields. Next, there is a database of general, so-called basic explanation patterns, with at least one for each basic game pattern. Third, there is a set of so-called learning strategies for specialization, generalization and replacement. They can be applied to a basic game pattern and one or more basic explanation patterns and produce a new basic explanation pattern.

Using this knowledge, a board position can be analysed without cases as follows: With the discrimination tree, the board position is analysed into a set of basic patterns. They constitute subproblems which are solved independently. For each basic pattern, general basic explanations are fetched from the database and returned. Learning strategies are applied. All resulting basic explanation patterns are offered to the user to be included in the solution. One could call this a generative problem solver, but the assessment of game patterns is rather heuristic.

This approach was extended to cases to acquire new basic explanation patterns. The key idea is to retain new basic explanation patterns and to retrieve and reuse them together with the basic patterns from the database. A basic explanation pattern is relevant for a basic game pattern if it explains a similar basic game pattern. Similarity is defined as being close together in the discrimination tree.

Heuristic attribute adjustment: INRECA

This system can be applied to all analytic tasks (Auriol et al., 1995). It operates on source cases containing objects with attributes. A query case contains partially instantiated objects. Their attribute values are matched against those of the source cases. From the most similar source case the missing attributes are copied to the query case. Thereby they can be adjusted by applying heuristic adaptation rules. Optionally, for classification and diagnosis tasks, attributes can be distinguished as solution features. Then only these attribute values are copied and adapted.

As background knowledge, INRECA has an object hierarchy, and completion and adaptation rules are attached to the object classes. Completion rules derive missing attribute values for a source case, adaptation rules produce missing attribute values for a query case given a source case.

In contrast to the other systems presented, INRECA is not sophisticated. It has been included because it represents case adaptation as performed in commercial systems—if adaptation is performed at all.

4.2.5 Heuristic adaptation by structure transfer

Heuristic structure transfer: TOPO This system refines and extends layouts by case adaptation. The layouts consist of geometrical objects of various types, for example of a spatial arrangement of rooms, pipes or furniture. TOPO represents a layout by a graph, consisting of objects as nodes and spatial relations as edges (Figure 3).

There exists no theory dividing a layout statically into problem and solution parts, therefore TOPO determines dynamically the parts to be transferred from a source case to a query case. It uses the heuristic, that every object of the source case, which is not found in the query case, might be part of the solution. TOPO finds the common part of a query case and one source case by determining the common subgraph of their two graphs. It offers the opportunity to transfer all objects from the source to the query case which are connected to the common part by a path of spatial relations. The user might specify the part to be transferred by selecting types of desired objects.

During transfer TOPO might change the size of transferred objects, if it is nescessary to preserve a spatial relation. For example a window is resized to touch both sides of a room. To avoid impossible sizes TOPO creates a statistic for each type of object about its geometry occurring in the case base and changes the size of an object to usual geometries only.

The adaptation described so far is additive only and this is the main aim of TOPO. A second way of

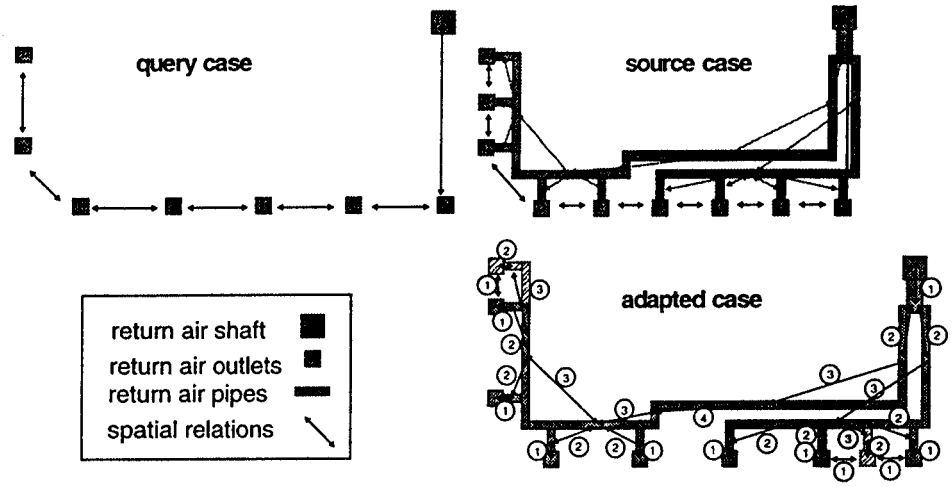


Figure 3 Structure transfer by TOPO. Both, query and source cases contain a shaft for return air and an arrangement of return air outlets. In the source case, they are connected by pipes. There is no unique match, and one is chosen by chance (dark arrows in search and source case). The paths are transferred incrementally as indicated by the numbers in the adapted case. It is not always desirable to transfer all paths, as the ones leading to outlets A and I. Therefore, the designer is asked before transfer, or he can repair the layout, or additional domain-specific heuristics must be added

using TOPO is transformational. TOPO searches in the query case for objects being in a seldom spatial relationship. For this reason TOPO creates a second statistic about the frequency of spatial relations occurring in the case base for each type of objects. For example, 100% of the outlets of the case base touch pipes and 2% of the chairs touch a shelf. In case of detecting seldom relationships of a query object, TOPO asks the user whether the position of the query object shoud be changed or not. If the position is confirmed to be changed TOPO searches for an object of the source case of same type as the query object which is related to the same type of objects, but by more frequent spatial relations.

4.3 Guidelines

Table 2 characterizes the surveyed systems using the criteria suggested in section 4.1—the systems are sorted by their task as in Table 1. Systems that restrict correspondences dynamically perform a match. It may be a graph matching as in TOPO, a structure match as in COMPOSER, a subcase match as in the route-planning application of PRODIGY/ANALOGY, or it may be guided by a discrimination tree as in CHESS. RESYN/CBR performs a match not to identify common parts, but to determine similarity. These matches are expensive and are computed only once. Most of the systems start with a fast preselection, compute the match, and then extract and adapt the cases. Thus, there is a smooth transition from retrieval to reuse.

Most criteria in Table 2 are strongly influenced by the adaptation technique. It directly determines the case representation and the effect of adaptation. It also determines how correspondences are established and which data are to be extracted. The adaptation technique is almost entirely dependent on the domain, on the methods and knowledge available. Therefore not much advice can be given for choosing a particular technique. The reader is referred to Kolodner (1993) for a thorough discussion of these issues.

But some advice can be given for the approach to adaptation, i.e., for the class of techniques that could be used, and the advantages and disadvantages that will ensue. As Figure 4 shows and is explained below, it inversely depends on the theoretical understanding of the domain. It influences the additional knowledge engineering effort, the effect of adaptation, and the size and quality of the source cases.

Table 2 Characteristics of selected adaptation methods

System	Approach to adaptation	Case representation	Correspondences	Data extracted	Effect on source case
MoCAS	derivation modification	graph	static	complete	transformation
PARIS	derivation modification	graph	static	complete	transformation
PRODIGY/ANALOGY	derivation modification	graph	dynamic	partial	transformation
CAPlan/CbC	derivation modification	graph	static	complete	transformation
AAAO	solution modification	set of objects	static	partial	transformation
RESYN/CBR	solution modification	graph	static	complete	substitution of structures
COMPOSER	corrective tools	graph	dynamic	partial	mostly substitution
IDIOM	corrective tools	set of objects graph	—	partial	substitution
DéJà Vu	heuristic substitution	object	static	partial	mostly substitution
EADOCS	heuristic substitution	object	static	partial	substitution
CHESS	heuristic substitution	set of objects	dynamic	partial	substitution
INRECA	heuristic substitution	object	static	partial	substitution
ToPo	structural transfer	graph set of objects	dynamic	partial	transformation

A problem solver for adaptation If available, most systems use a generative problem solver. Good reasons are given in Figure 4: there is less knowledge engineering overhead and results are good even if the quality of the cases retrieved is low. But as COMPOSER illustrates, a good tool applied to good source cases may outperform a problem solver.

Using a problem solver, the systems favour derivation modification. Modifying derivations requires less source cases, and it better cuts down the search. Planners and model-based reasoners are good candidates for derivation modification. However, RESYN/CBR demonstrates that a plan may also be transformed, rather than being replayed. But then the derivation knowledge should be obtained from other sources, like retrieval.

A tool for adaptation Usually, the second best choice is to use tools for adaptation. Since there are not so many generative tools and since assessment tools with their generate-and-test approach are not very efficient, tools are mostly used to repair solutions, i.e., to achieve substitutional adaptations. Prominent candidates are constraint satisfaction or relaxation algorithms. They can repair inconsistent and incomplete solutions. But the constraint network must be extracted from the cases retrieved.

Heuristic adaptation Heuristics must be used if there are neither tools nor problem solvers. Most commercial case-based reasoning systems allow to extract values from attributes in a source case and adjust them with some heuristic rules, as in INRECA. More powerful are structure transferring approaches, but they require a structured case representation, like a graph in TOPO. The result of heuristic adaptation strongly depends on the quality of the cases retrieved and on the assumption that similar cases have similar solutions.

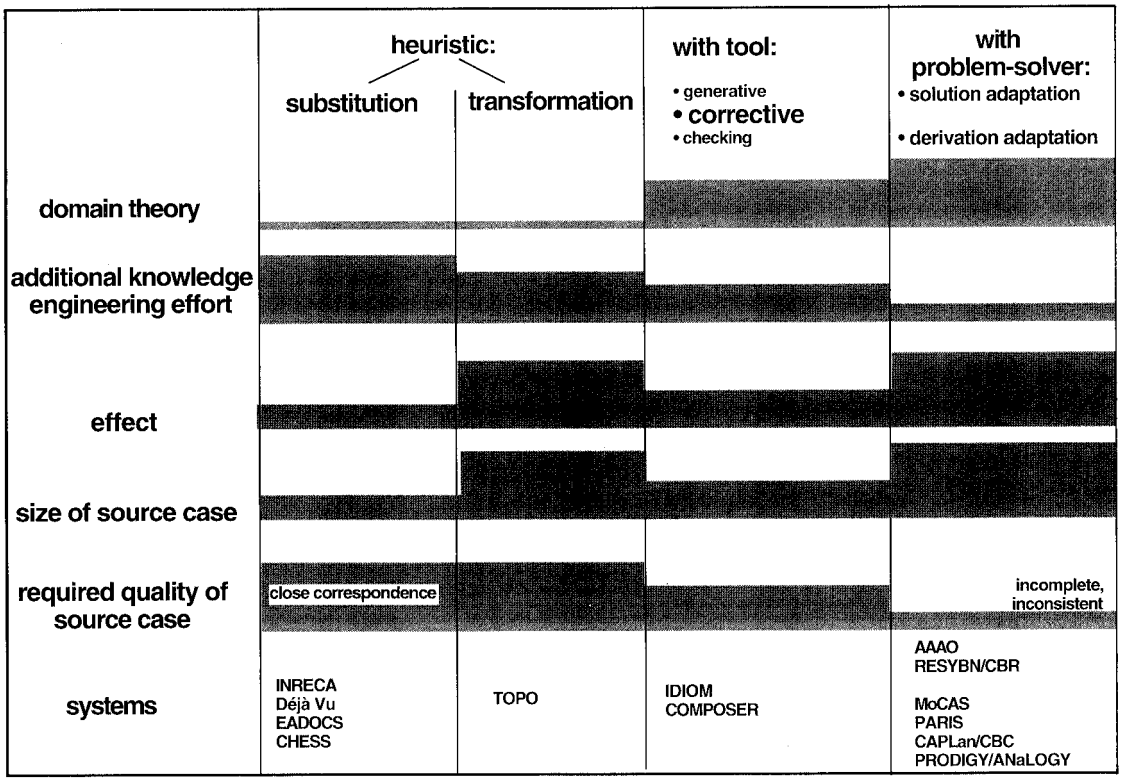


Figure 4 Approaches to adaptation are compromises between different trade-offs. The size of the bars per row indicates for each approach to adaptation (column) the relative amounts of domain theory, knowledge acquisition efforts, size of source cases, effects of adaptation and quality of source cases

5 Reuse of multiple cases: strategies

In the previous section, it was assumed that the given problem can be interpreted as a query case and that a single source case is sufficient to solve the problem. This assumption will now be relaxed. Besides adaptation, as described in section 4, a new issue is brought into focus: the way how multiple cases are retrieved and reused. Again, a framework will be proposed, it will be applied to the systems, and some guidelines will be extracted.

5.1 Framework

So far it has been assumed that the given problem directly constitutes a query case and that a single source case is sufficient to solve the problem. This assumption will now be relaxed.

Solving a problem by a single case is only possible if each source case contains a sufficiently complete solution. There may be several reasons why this condition might be violated: It may be difficult to acquire such source cases, they may not be intuitive to the user, they may not be adaptable because adaptation methods are not powerful enough, or problems and solutions are so complex and unique that too many source cases would be needed in the case base. When more than one source case is needed, how does the system know which cases to retrieve? Is there knowledge about problem decompositions and are there source cases for specific subproblems? Or do the source cases specify problem decompositions? Or are suitable combinations of cases dynamically considered during retrieval? Or is the adaptation technique so powerful that it can fit together badly fitting cases? Depending on the answers to these questions different strategies can be implemented. Figure 5 shows the options.

First, a set of cases must be retrieved for the problem. This can be achieved by decomposing the problem into subproblems entirely before case-based reasoning, for instance by heuristics or according to a theory. Then each of the subproblems is interpreted as a query case, and for each

one a source case can be retrieved. Or the problem is directly interpreted as a query case and a set of source cases is retrieved, either iteratively or in a single step.

Second, the retrieved cases have to be adapted. This can be done case by case or simultaneously.

Third, if there are several adapted cases left, they have to be composed to a single, complete solution. These considerations give rise to a modification of Figure 1 into Figure 5.

In a concrete system, several of these options may be combined. For instance, the problem may be decomposed and for each subproblem multiple cases may be retrieved. The retrieved cases can be concrete cases which contain a partial solution for the query case, or they can be abstract cases which suggest a further decomposition. Some cases may be adapted simultaneously and some others may be adapted individually. Then the outcome has to be composed to a final solution.

To characterize the strategy of a particular system, a good notation is needed. It should distinguish expansive steps like multi-case retrieval and problem decomposition, contractive ones like multi-case adaptation and composition, and steps that do not essentially affect branching. Also, the notation should allow to distinguish case-based processes from others.

For this purpose, an abstract kind of data flow diagram is used. It has three types of nodes: problem nodes (P) indicating a problem or subproblem to be solved, case nodes (C) indicating a source case that has been retrieved for a problem, and solution nodes (S). Nodes can be linked by 1:1, n:1, 1:n, and n:m edges indicating data flow between the nodes. Edges can be classified by the types of nodes they link. Edges leading to case nodes (→ C) represent retrieval, edges leaving case nodes (C →) represent adaptation. Adaptation takes at least two arguments, a query and a source case; for sake of simplicity, the adaptation edge is not connected to the query case, which can always uniquely be determined from the context. The remaining edges are independent of case-based reasoning. Figure 6 lists major primitives. They suffice to describe the systems selected at a level of abstraction that exhibits both similarities and differences.

Interesting is the adaptation edge from a case node to multiple problem nodes (column 1 row 2): the case must be an abstract case whose adaptation yields a set of subproblems. Two edges stem from iterative strategies that take into account intermediate solutions: an edge from a problem node and from a solution node to a case node indicates iterative retrieval (column 4 row 1); and an edge from a case node and from a solution node to a solution node indicates iterative adaptation and

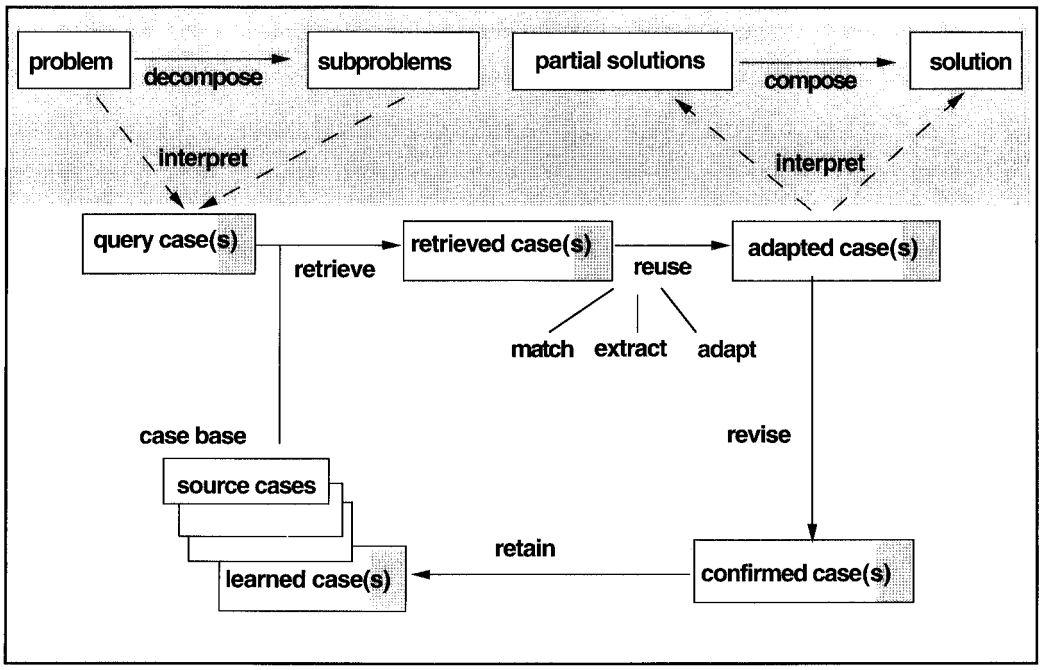


Figure 5 Case-based reasoning with a single case (white background) and with multiple cases (shaded background). The dashed lines indicate options

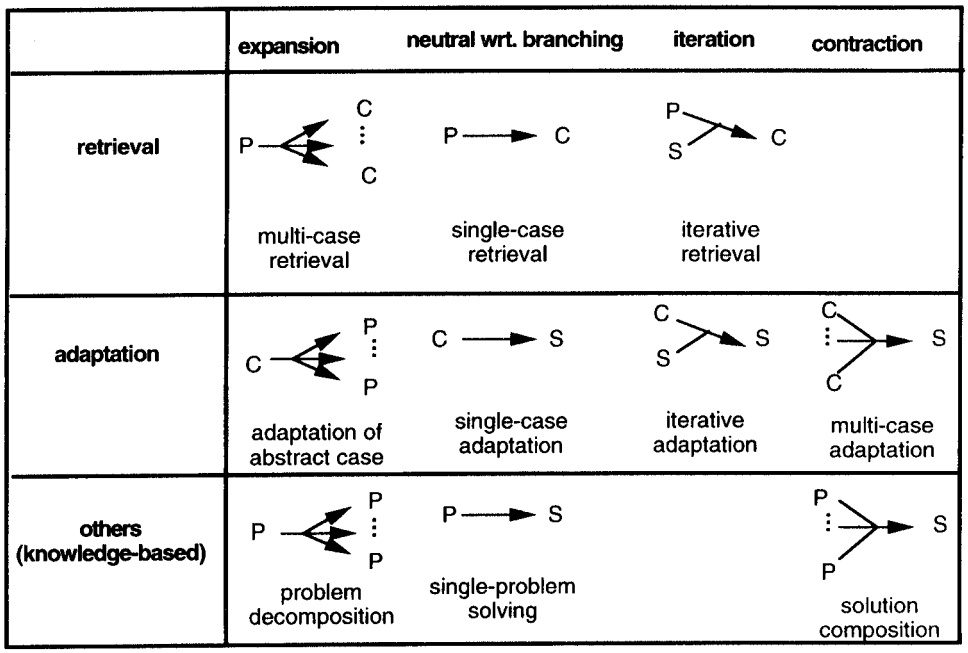


Figure 6 Potential constituents of strategies for case-based reasoning

integration (column 4 row 2). A bit exotic is a retrieval edge from multiple problems to multiple cases (column 3 row 1). It occurs in only one of the systems, namely where subproblems are obtained by external decomposition, a single source case with similar subproblems is retrieved, and from this source case further subcases for the subproblems are extracted. This example illustrates how difficult things may become in practice. To describe other case adaptation systems or to change the level of detail, this set of primitives may have to be extended.

There are many ways to combine the primitives to form strategies for case-based reasoning. Figure 7 shows the ones realized in the systems studied:

- In the *single concrete case strategy*, a single source case is retrieved and then reused. All systems in section 4.2 were described as if realizing this strategy.
- In the *single abstract case strategy*, a single, abstract case is retrieved and reused. But if the abstract case contains a problem decomposition, reusing this abstract case involves a simultaneous solution of its subproblems. In MoCAS and PARIS, the abstract case specifies subproblems that are solved by the generative problem solver.
- The *iterated abstract case strategy* starts by iteratively retrieving abstract cases, introducing more and finer subproblems. Atomic subproblems are iteratively adapted.
- The *iterated concrete case strategy* retrieves and reuses a single concrete case. But the result may be insufficient so that the procedure must be iterated.
- The *simultaneous strategy* provides a set of source cases in a single retrieval step, and then adapts and integrates them in a single reuse step.
- *Embedded strategies* start with a problem decomposition and end with a solution composition, both of which are independent of case-based reasoning. In between, case-based reasoning is applied to all subproblems. The figure shows the strategy of the CHESS system. It embeds an iterated simultaneous strategy with a sophisticated retrieval. Details follow later.

All strategies, apart from the iterative one, can be split into a half for retrieval (left hand side) and another half for reuse. These halves can be recombined. For instance, simultaneously retrieved source cases could be adapted iteratively, and in the iterated abstract case strategy, all concrete source cases could be adapted simultaneously. Indeed, the iterated abstract case strategy is an amalgamation of the single abstract case strategy and the iterated concrete case strategy.

Apart from the strategy, semantical criteria for decomposition and composition must be established. The next section will show that these strongly depend on the task.

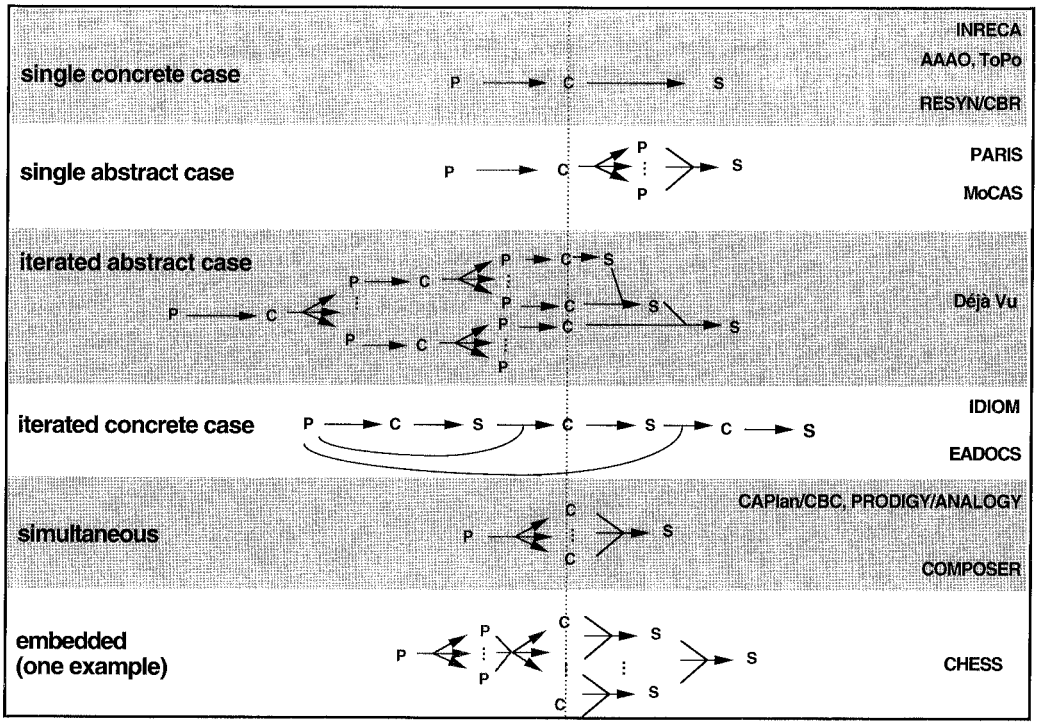


Figure 7 Some strategies for case-based reasoning realized by the systems studied in this section. The dotted line separates retrieval parts from reuse parts

5.2 Selected approaches

Many systems described in section 4.2 are multi-case reusing systems, although they were then treated as if adapting a single source case. Now the systems are revisited to analyse the strategy they employ.

5.2.1 Reusing a single abstract case

MoCAS and PARIS These systems reuse a single source case, which is usually an abstract case (c.f. section 4.2.1). Therefore, both systems realize a single abstract case strategy. Each abstract case contains a derivation, or explanation, that defines a problem decomposition. Each abstract operator or abstract explanation must be refined and the results must be integrated. This is done using the from-scratch problem solver. To find the best adaptable source case, retrieval starts with the most abstract source cases and proceeds to more concrete cases.

5.2.2 Reusing abstract cases iteratively

DéJà Vu The source cases in DéJà Vu (Smyth and Keane, 1995) are tailored so that they are easy to adapt heuristically (c.f. section 4.2.4). As a consequence, they are too small to solve a problem in a single step. Therefore, the single-case retrieval described in section 4.2 is iterated. Since the partial solutions may interact, there are two kinds of adaptation heuristics. Local ones adjust a concrete source case only with respect to the query case, and global heuristics repair conflicts between the new solution and the rest of the configuration. Their effect is usually transformational, but still limited.

5.2.3 Reusing concrete cases iteratively

The following systems retrieve and adapt case by case. The resulting intermediate solutions are taken into account by the next retrieval step. Care must be taken that the iterative process converges.

EADOCS EADOCS retrieves and reuses a single source case to instantiate a prototype design with objects and to constrain or determine their attributes (c.f. section 4.2.4). The source case retrieved is the one with the most similar components, i.e., components that achieve the behaviours specified in the prototype. However, both, the possible combinations of behaviours and of behaviour attribute values are huge so that the retrieved case will be insuffcient in general. After adapation it still will not satisfy all specified behaviours. These insuffiencies are interpreted as new query cases. For the worst insufficiency, a component is retrieved that fits the context of the current design and that produces the desired behaviour. It is superimposed on the current design, and the process is iterated to detect and handle the next insufficiency. In this approach, components are treated as subcases. They influence the retrieval of their embedding source case and they can be retrieved independently.

The designers of EADOCS consider it a serious problem that the design might not converge. Therefore, for repair only components are used that fit the current design so that they can be superimposed. The order in which repairs are performed may affect convergence.

IDIOM In IDIOM (Smith et al., 1995) source cases are layouts of apartments (c.f. section 4.2.3). Since one seldom designs a complete apartment from a single source case, source cases are statically split into small subcases, so-called design objects. By iteratively adding a subcase to an evolving layout, one can more easily construct more complex solutions than by reusing a single source case. The user retrieves the subcase and extracts some parts, which are then adapted as described in section 4.2. For this approach to be successful, interactions between the constraints must be loose enough so that the solution can converge.

5.2.4 Reusing cases simultaneously

The systems below adapt a set of retrieved source cases simultaneously because there are strong interactions between them. Also all systems retrieve the source cases simultaneously. It seems that simultaneous retrieval can better guarantee to find complementary or compatible (pieces of) source cases.

PRODIGY/ANALOGY The description of PRODIGY/ANALOGY in section 4.2.1 was simplified. The system retrieves and reuses parts from multiple source cases. During retrieval, a set of source cases is searched that together cover the goals of the query case. To keep the set small, source cases that cover larger parts are preferred. Retrieval may be expensive because the initial situation must be taken into account to guarantee that the goals can be reached from this situation. To outperform a generative planner in general, retrieval must be extremely fast. An indexing scheme suitable for one domain may be inappropriate for another. For route planning, indexing uses domain-specific knowledge about routes: routes are split into subroutes corresponding to subcases. Retrieval does more than return a set of source cases. It indicates which parts of the cases are relevant and how they contribute to the overall solution. In the route planning task, the case parts are returned in the order in which they shall be combined.

For reuse, the planner can merge the retrieved case parts in different modes: sequentially, as in route planning, interleaved, guided by the user, or, at each decision point, reusing a randomly selected case among the candidates retrieved. Anyway, adaptation then proceeds as described before. The planner replays the retrieved case parts in one of the modes. Replanning with the domain theory is necessary to bridge gaps between case parts or to bridge invalid pieces in a retrieved solution.

CAPlan/CbC This system uses multiple source cases similarly to PRODIGY/ANALOGY, but it does not extract parts from the cases (c.f. section 4.2.1).

COMPOSER In this system, source cases contain only primitive constraint satisfaction problems and their solution (c.f. section 4.2.3). To solve a query case, several source cases are needed. During retrieval, a set of source cases is determined that cover the whole query case. Using only small source cases and combining them for each query case, the case base can be kept relatively small. Source

cases may overlap because the constraint repair algorithm can create and delete constraint variables dynamically. For adaptation, the constraint networks of all source cases are united and the constraint repair algorithm is applied. When combining multiple source cases, the adaptation or integration process may diverge. The authors claim to have mastered this problem by using the constraint repair algorithm for simultaneous adaptation. The critical parameters are overlapping variables, i.e., variables which occur in more than one source case. If these variables are not too constrained, case-based reasoning will most probably converge and be better than reasoning from scratch.

5.2.5 Embedding case reuse

The next system exemplifies an embedded strategy: the case-based reasoning part is preceded by external problem decomposition and succeeded by external solution composition. Both, problems and source cases are decomposed in the same way, using the same discrimination tree. Thus, it is easy to retrieve subcases suitable for a particular subproblem (a basic game position).

CHESS This is a multi-case reuse system and the description in section 4.2.4 was simplified. For each chess board position, CHESS returns a set of basic explanation patterns. But it composes them into a single solution, called a multiple explanation pattern. Such patterns can be stored as source cases. As subcases, each source case contains its basic explanation patterns. The problem, a board position, is first analysed into its basic game patterns, the subproblems. For each subproblem, all subcases for similar basic game positions are retrieved. Then the source case containing the largest number of similar subcases is selected. Retrieval is sophisticated, because the subcases are retrieved from a single source case which is retrieved first of all taking into account all the subproblems. This is expressed by the edge from multiple problems to multiple cases in Figure 7.

After retrieval, the subproblems are solved iteratively. For each subproblem, basic patterns are retrieved both from the database of general patterns and from the subcases of the retrieved case. Learning strategies are applied to various pairs of basic explanation patterns. The resulting basic patterns are turned into a multiple explanation pattern which can be retained as a new source case if the user agrees.

5.3 Guidelines

The systems analysed decompose their problems in a task-specific way: planners decompose by subgoals, systems for technical diagnosis, configurators and design systems by components and subconstellations. In choosing their strategy the systems analysed exhibit a tendency to simplicity—CHESS is the only system realizing an embedded strategy. What are the reasons behind this?

Single or multiple case reuse—or problem reduction? If solutions are sufficiently simple and adaptation techniques are sufficiently powerful, one should try to reuse a single case. As a trick, it might be possible to formulate and adapt a skeleton case at some higher level of abstraction, as is done in PARIS and MoCAS. Alternatively, one may retrieve a single source case that solves only part of the problem; this is essentially what happens in TOPO. If a single case is insufficient, one has to reuse multiple cases.

Multi-case strategies: retrieval without decomposition? Available static problem decomposition knowledge should be exploited for an embedded strategy. It reduces retrieval time. CHESS is the only system starting with a static problem decomposition. Reasons might be that external decomposition knowledge often is not available, that it is not so suited, or that the integration of case-based reasoning with other problem solving techniques is yet in its infancy—then CHESS would be a pioneering system. Last but not least, multi-case retrieval and reuse is just flexible enough.

Indeed, by retrieving multiple source cases for a single problem, a problem decomposition is achieved a posteriori: every source case defines a subproblem in the query case. And with the

addition of new source cases new problem decompositions may be found. So multi-case retrieval is more flexible than a priori decomposition. To cite the author of EADOCS: a static decomposition declares how to reformulate a problem, a set of source cases declares what problems can be solved.

A compromise between static decomposition and decomposition by concrete cases is the use of abstract cases as in DéJà Vu. They allow to retain previous decompositions explicitly.

Multi-case strategies: reuse without composition? The strategy for case reuse depends on the adaptation technique and on the interaction between the cases retrieved. Simultaneous adaptation, it seems, can well cope with interactions, while parallel single-case adaptations should only be applied to independent source cases, as in CHESS. So, if available, simultaneous adaptation techniques should be preferred. Otherwise, one should try to extend a single-case adaptation technique into an iterative one. But care must be taken that the iteration will converge. For that purpose the order in which source cases are adapted may be crucial, a problem recognized in EADOCS.

In general, the solution need not be composed in the same way as the problem has been decomposed, as DéJà Vu demonstrates. But typically, simultaneous retrieval prepares well the ground for simultaneous reuse, because it can select compatible source cases; and incremental retrieval is suited for incremental reuse, because it is more up-to-date. That means the preceding iteration will have changed the query case and iterative retrieval can take this into account. External solution composition is only required if source cases are adapted independently, as in CHESS.

6 Source cases—the third factor

The conclusions from the micro level of single-case adaptation in section 4.3 and from the macro level of multi-case reuse in section 5.3 indicate that both aspects are related to a third factor in the design of a case reuse system: the size and content of the source cases and the relations betweeen them. This section is devoted to this neglected yet central affair. The systems need not be revisited, because source cases were discussed together with the strategies in section 5.2. A last group of guidelines will be given to relate the choice of source cases to the approach to adaptation and to the strategy.

6.1 Framework

Both adaptation technique and overall strategy have impacts on the size and content of the source cases and the relations betweeen them. The single and abstract case strategies need large source cases, the other strategies need smaller ones. The adaptation technique prescribes its input: the case representation and the quality of the cases retrieved. In general, heuristics require simpler cases, problem solvers can deal with complex ones, tools are in between.

Besides, *case elicitation* and *cognitive adequacy* are two additional major factors influencing the design of source cases: cognitive adequacy means that the users should recognize source cases as meaningful units. Case elicitation refers to the process of extracting source cases from primary data sources. It may involve data decomposition or composition to tailor the source cases to the potential query cases.

Figure 8 shows alternative kinds of source cases as encountered in the systems studied. They can be compared with respect to—at least—three features. A source case may contain a *complete* or a *partial* solution. It may be *concrete* or *abstract*, where a concrete source case contains particles that can occur in the final solution, while an abstract source case contains derivational information like a solution path, justifications, or subproblems. A source case can be isolated or related to others, e.g., to subcases or to more abstract cases raising subproblems to be solved. In the systems studied, not all combinations of features occur, and five classes of source cases can be distinguished:

- *Complete source cases* which additionally are concrete and isolated.
- *Partial source cases* which additionally are concrete and isolated.

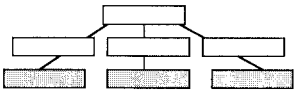
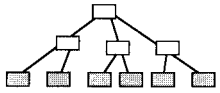
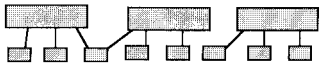
types of cases	contents		relations	visualisation
partial	partial	concrete	—	
complete	complete	concrete	—	
layered	complete	concrete & abstract	abstraction	
hierarchical	partial	concrete & abstract	partial refinement	
composite	complete & incomplete	concrete	part of	

Figure 8 Alternatives for source cases

- *Composite source cases*, which are concrete, complete, and related to concrete but partial *subcases*.
- *Hierachical (sets of) source cases*, which are partial and form a hierarchy with concrete cases at the bottom and abstract cases above.
- *Layered (sets of) source cases*, which are complete and form a hierarchy with concrete cases at the bottom and abstract cases above.

6.2 Guidelines

The different kinds of source cases represent compromises between several trade-offs, as shown in Figure 9 and explained below.

Complete source cases They preserve more context, e.g., the whole building as a context for all its rooms. Users tend to prefer them because of the context. But complete cases may be hard to reuse, especially complex ones as occur in synthesis tasks. Only parts of them may be relevant for the problem at hand. Therefore, parts of a complete case may have to be extracted dynamically, as in TOPO, and possibly, parts from several complete cases are needed.

Partial source cases They can be combined flexibly. But one needs more of them and retrieval becomes more difficult because complementary source cases must be found. As COMPOSER demonstrates, partial cases may overlap and one must make sure that the adaptation method can cope with this overlapping.

Composite source cases They are a good compromise between complete and partial cases. Composite cases preserve the context for their subcases. A composite source case may contain a unique decomposition into subcases, as in IDIOM or CHESS, or multiple ones defined by multiple indexes, as in EADOCS or PRODIGY/ANALOGY. Subases may be shared between composite cases, as in EADOCS, or exclusively be part of a single one, as in CHESS. Often both, composite cases and subcases, can be accessed directly and indirectly, so that retrieval can be flexible, proceeding from the composite case to its subcases (EADOCS), vice versa from subcases to the composite case (CHESS), or accessing only the subcases (IDIOM, CHESS, PRODIGY/ANALOGY, EADOCS) or only the composites.

Layered source cases Complete or composite source cases are adequate when solutions have strongly interacting parts that should not be broken into independent partial cases. The same is true of layered source cases. As in PARIS and MoCAS, a complete case is represented at different levels of abstraction. At each layer, all relevant interactions are reflected and a complete solution is

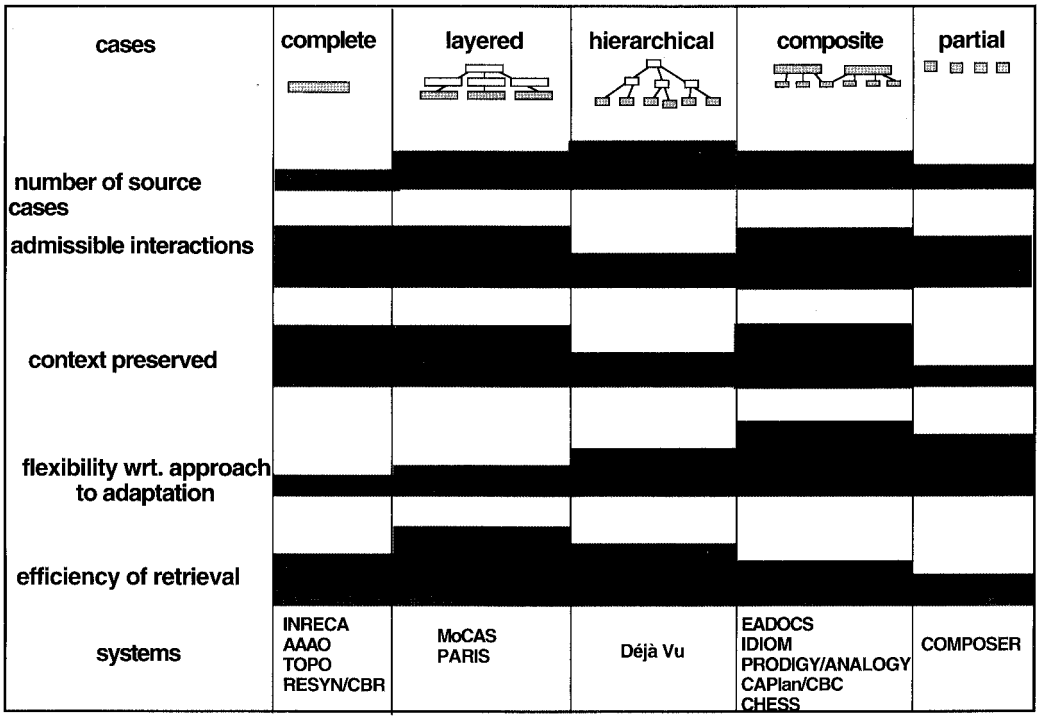


Figure 9 The kind of source cases is a compromise between different trade-offs (rows). Depending on the kind of source cases (column), more or less source cases may be needed with more or less potential interactions between them. The source cases may preserve more or less context, allow more or less different approaches to adaptation, and require more or less efficient retrieval. The relative degrees are expressed by the size of the bars

presented. But abstract cases are more general and can be reused to produce very different solutions. Thus, the number of source cases can be reduced.

Hierachical source cases—They combine abstract and partial cases. The context of the concrete cases is less preserved than in composite cases, but better than with isolated partial cases. That means interactions within a complete solution should correspondingly be limited in scope. Hierarchical cases are a good means to aquire new decomposition knowledge by explicitly storing it as abstract cases. Retrieval of hierarchical cases is efficient. One has to descend the hierarchy stepwise.

7 Major design decisions

7.1 How to design a case reusing system

Assume a new application is to be built, and the question is whether case reuse would be feasible or even promising. With the application, the task and the domain are predetermined. So one could set out to identify knowledge and methods that are available or easy to obtain. Next, the approach to adaptation could be selected. It constrains the kind of source cases, which in turn is closely related to the strategy. Guidelines for adaptation and strategies were developed in the preceding two sections, guidelines for source cases in the previous section. But some more global recommendations can be extracted from the systems studied. The major decisions in the design of these systems are summarized in Table 3 and related to the task. This figure can be interpreted as follows (cf. Figure 10).

A single complete source case To keep things simple, one should try to reuse a single, complete and concrete source case. If the problems are sufficiently simple one should try heuristic adaptation. If

Table 3 The major design decisions in the systems studied. Cases are the primary sorting criterion, strategies the second

System	Task	Approach to adaptation	Strategy	Source cases
INRECA	analysis	heuristic substitution		
AAAO	design	solution modification	single case	complete
ToPo	design	structure transfer		
RESYN/CBR	planning	solution modification		
MoCAS	diagnosis			
		derivational	abstract case	layered
PARIS	planning			
Déjà Vu	synthesis	heuristic substitution	iterated abstract case	hierarchical
COMPOSER	planning configuration	tool-based	simultaneous	partial
CHESS	decision support	heuristic substitution	embedded	composite
EADOCS	design	heuristic substitution	iterated concrete case	
IDIOM	design configuration	tool-based		composite
PRODIGY/ANALOGY	planning	derivational	simultaneous	composite
CAPlan/CbC				

they contain complex structures, one can try heuristic structure transfer. Otherwise one should try a from-scratch problem solver. Substitutional heuristics are a good candidate for analysis tasks, heuristic structure transfer and solution modification suit design tasks. If a problem cannot be solved by a single case, one may solve it partially, as in TOPO.

A single layered source case As another trick to avoid multiple source cases, it might be possible to formulate and adapt a skeleton case at some higher level of abstraction, as is done in PARIS and MoCAS. Tailored to such layered source cases are the single abstract case strategy together with a

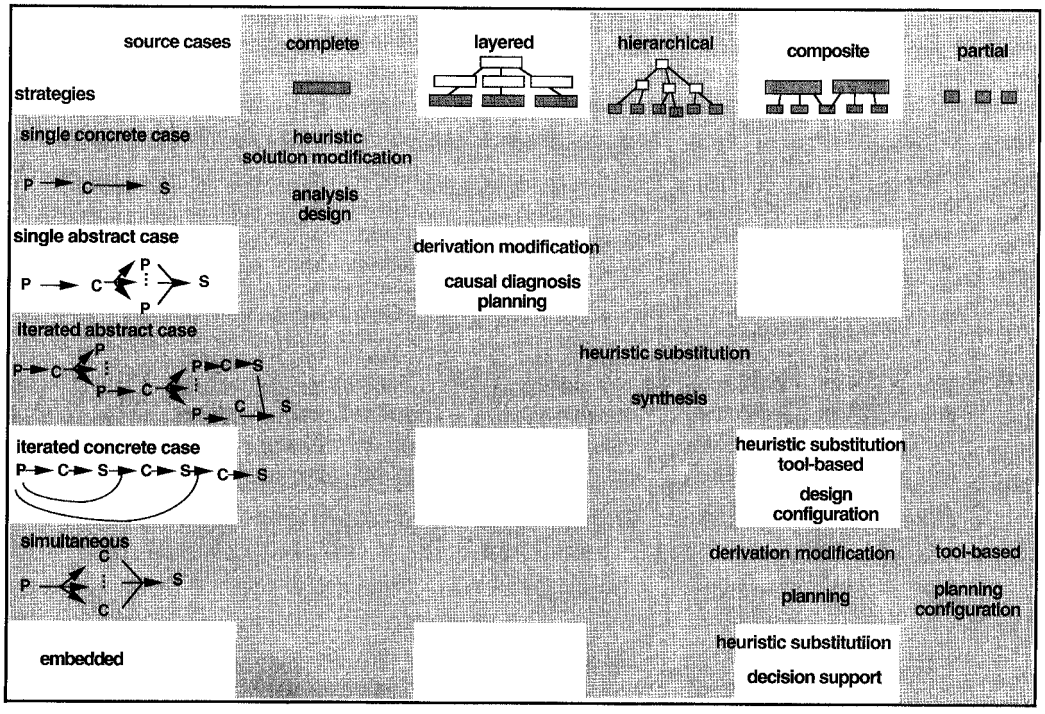


Figure 10 Dependencies between strategy, cases, approach to adaptation, and task

problem solver for adaptation by derivation modification. This is a good choice in hierarchical planning and model-based diagnosis.

Embedded case-based reasoning If a single case is insufficient, one has to reuse multiple source cases. Available static knowledge for problem decomposition knowledge and solution composition should be exploited for an embedded strategy. Depending on the interactions between the subproblems, special adaptation techniques have to be employed. Parallel single-case adaptations, as in CHESS, should only be applied to independent source cases.

Multiple cases and heuristics A compromise between external problem decomposition and a posteriori decomposition by concrete cases is the use of hierarchical cases as in DéJà Vu. They retain previous decompositions explicitly. Tailored to this kind of cases is the iterated abstract case strategy. It allows to tackle synthetic tasks by a simple adaptation technique like heuristic substitution. This approach must be chosen when the problem is too complex for a single case and only heuristics are available for adaptation.

Multiple cases and tools or problem solvers In general, existing problem solvers or tools should be exploited, even if a single source case is not sufficient. Often, the tools or problem solvers can process input that has been assembled from multiple partial or composite source cases, or the input can be preprocessed correspondingly. Composite cases are more versatile than partial ones, because typically both, composite cases and subcases, can be accessed directly and indirectly. In the systems studied, partial source cases are only used simultaneously. Composite source cases are also used for the iterated concrete case strategy, probably because they present the cases in small bits while preserving their context.

Simultaneous adaptation, it seems, can best cope with interactions between partial solutions. Otherwise, one should try to extend a single-case adaptation technique to an iterative one. For that purpose the order in which source cases are adapted may be crucial for the process to converge, a problem recognized in EADOCS. Among the systems studied, the iterated strategy is applied in synthetic tasks for adaptation by heuristic substitution and tools. The simultaneous strategy is used for tool-based adaptation in planning and configuration.

Source case elicitation The problem of eliciting suitably tailored source cases is closely related to the strategy and the adaptation technique. Source cases should be cut so that they fit the query cases and are easy to adapt. Thus, for gaining new source cases, a complex solution should be decomposed using the same heuristics, knowledge or technique that is used to decompose a problem or to retrieve multiple cases. For instance, CHESS decomposes both problems and source cases according to its discrimination tree.

7.2 When to reuse cases?

In the past different reasons have been put forward for reusing cases rather than pure knowledge. The domain may not be understood well enough, an existing problem solver may be too slow, people may accept cases better than rules, knowledge acquisition and maintenance may be reduced. This study has shown that more sophisticated applications, in particular for synthesis tasks, will require the adaptation of existing cases. But adaptation is like problem solving: it requires knowledge and knowledge acquisition, it may be complex and NP-complete. To the extent that we understand (formalize, measure) problem solving, we will understand this branch of adaptation. This is the bad news. The good news is that existing problem solvers can be and have been accelerated by the reuse of cases, and that less understood domains could be tackled with cases combined with heuristic approaches to adaptation and human interaction.

As experience with case-based prototypes progresses, researchers should turn to techniques to predict the costs and risks of reusing cases, both to choose between knowledge-based or case-based reasoning, or to build hybrid systems.

Acknowledgement

I would like to thank Frierdich Gebhardt, Wolfgang Gräther, Barbara Schmidt-Belz, Carl-Helmut Coulon, and Hans Voss for their comments and suggestions.

References

Aamodt, A and Plaza, E, 1994. "Case-based reasoning: foundational issues, methodological variations, and system approaches" *AI Communications* **7** (1) 39–59.

Adami, P, 1995. "Adaptation by active autonomous objects (AAAO)" In: K. Börner (ed.), *Modules for Design Support* 46–50, GMD, Sankt Augustin.

Auriol, E, Manago, M, Althoff, K-D, Wess, S and Dittrich, S, 1995. "Integrating induction in case-based reasoning: methodological approach and first evaluations" In: J-P Haton, M Keane and M Manago (eds.), *Advances in Case-based Reasoning: Second European Workshop, EWCBR-94* 18–32, Springer-Verlag.

Bergmann, R, Pews, G and Wilke, W, 1994. "Explanation based similarity: a unifying approach for integrating domain knowledge into case-based reasoning for diagnosis and planning tasks" In: S Wess, K-D Althoff and MM Richter (eds.), *Topics in Case-based Reasoning: First European Workshop, EWCBR-93, selected papers: Lecture Notes in Artificial Intelligence 837* 182–196, Springer-Verlag.

Carbonell, JG, 1986. "Derivational analogy. A theory of reconstructive problem solving and expertise acquisition" In: RS Michalski, JG Carbonnell and TM Mitchell (eds.), *Machine Learning: An Artificial Intelligence Approach, Vol* 2 371–392, Morgan-Kaufman.

Coulon, C-H, 1995. "Automatic indexing, retrieval and reuse of topologies in complex designs" In: PJ Pahl and H Werner (eds.), *Proceedings of the Sixth International Conference on Computing in Civil and Building Engineering* 749–754, Balkema, Rotterdam.

Cunningham, P, Finn, D and Slattery, S, 1994. "Knowledge engineering requirements in derivational analogy" In: S Wess, K-D Althoff and MM Richter (eds.), *Topics in Case-based Reasoning: First European Workshop, EWCBR-93, selected papers: Lecture Notes in Artificial Intelligence 837* 234–245, Springer-Verlag.

Haigh, KZ and Veloso, MM, 1995. "Route planning by analogy" In: M Veloso and A Aamodt (eds.), *Case-based Reasoning Research and Development: First International Conference, ICCBR-95* 169–180, Springer-Verlag.

Hanney, K, Keane, MT, Smyth, B and Cunningham P, 1995. "Systems, tasks and adaptation knowledge: revealing some revealing dependencies" In: M Veloso and A Aamodt (eds.), *Case-based Reasoning Research and Development: First International Conference, ICCBR-95* 461–470, Springer-Verlag.

Kerner, Y, 1995. "Learning strategies for explanation patterns: basic game patterns with application to chess" In: M Veloso and A Aamodt (eds.), *Case-based Reasoning Research and Development: First International Conference, ICCBR-95* 491–500, Springer-Verlag.

Kolodner, JL, 1993. *Case-based Reasoning*, Morgan-Kaufmann.

Muñoz-Avila, H and Huellen, J, 1995. "Retrieving cases in structured domains by using goal dependencies" In: M Veloso and A Aamodt (eds.), *Case-based Reasoning Research and Development: First International Conference, ICCBR-95* 241–252, Springer-Verlag.

Napoli, A and Lieber, J, 1994. "A first study on case-based planning in organic synthesis" In: S Wess, K-D Althoff and MM Richter (eds.), *Topics in Case-based Reasoning: First European Workshop, EWCBR-93, selected papers: Lecture Notes in Artificial Intelligence 837* 458–469, Springer-Verlag.

Netten, BD, Vingerhoeds, RA and Koppelaar, H, 1995. "Expert assisted conceptual design: an application to fibre reinforced composite panels" In: R Oxman, M Bax and H Achten (eds.), *Design Research in the Netherlands* 125–139, Faculty of Architecture, Planning, and Building Design Science, Eindhoven University.

Purvis, L and Pu, P, 1995. "Adaptation using constraint satisfaction techniques" In: M Veloso and A Aamodt (eds.), *Case-based Reasoning Research and Development: First International Conference, ICCBR-95* 289–300, Springer-Verlag.

Riesbeck, CK and Schank, RC, 1989. *Inside Case-Based Reasoning*, Lawrence Erlbaum.

Slade, S, 1991. "Case-based reasoning: a research paradigm" *AAAI Magazine* **12** (1).

Smith, I, Lottaz, C and Faltings, B, 1995. "Spatial composition using cases: IDIOM" In: M Veloso and A Aamodt (eds.), *Case-based Reasoning Research and Development: First International Conference, ICCBR-95* 88–97, Springer-Verlag.

Smyth, S and Cunningham, C, 1992. "Déjà Vu: a hierarchical case-based reasoning system for software design" In: B Neumann (ed.), *ECAI 92: 10th European Conference on Artificial Intelligence* Vienna. 587–589, Wiley.

Smyth, S and Keane, MT, 1995. "Retrieval and adaptation in Déjà Vu, a case-based reasoning system for software design" In: *Adaptation of Knowledge for Reuse: A 1995 AAAI Fall Symposium: working notes* 99–105, MIT Press.

Veloso, MM and Aamodt, A (eds.), *Case-based Reasoning Research and Development: First International Conference, ICCBR-95: Lecture Notes in Artificial Intelligence 1010* Springer-Verlag.

Veloso, MM, 1994. *Planning and Learning by Analogical Reasoning: Lecture Notes in Computer Science 886* Springer-Verlag.

Voss, A, Bartsch-Spörl, B and Oxman, R, 1996. "A study of case adaptation systems" In: J Gero and F Sudweeks (eds.), *AI in Design'96* Stanford. Kluwer.

Voss, A, 1996a. "How to solve complex problems with cases" *Engineering Applications of Artificial Intelligence* Special Issue AI in Design.

Voss, A, 1996b. "Towards a methodology for case adaptation" In: W Wahlster (ed.), *ECAI'96* Wiley.

Watson, I and Marir, F, 1994. "Case-based reasoning: a review" *The Knowledge Engineering Review* **9** 327–354.