Case-Based Reasoning: A New Technology for Experience Based Construction of Knowledge Systems

Klaus-Dieter Althoff    Michael M. Richter
Wolfgang Wilke
TR-97-008
March 1997
Case-Based Reasoning:
A New Technology for Experience Based
Construction of Knowledge Systems

Klaus-Dieter Althoff, Michael M. Richter, Wolfgang Wilke
Case-Based Reasoning:  
A New Technology for Experience Based Construction of Knowledge Systems

Klaus-Dieter Althoff, Michael M. Richter, Wolfgang Wilke  
Centre for Learning Systems and Applications  
University of Kaiserslautern

Abstract

We will discuss the role of case-based reasoning – a new emerging technology that contributes to solving the well-known problems of software maintenance, reuse, and quality improvement by storing, retrieving and adapting similar past cases – in this new light. Case-based reasoning, which has proven to be of practical importance by a large number of industrial/business applications, is a flexible approach to software development that has overcome the indicated difficulties to a large extent. We will point out in which way case-based reasoning takes up the separation issue by a certain decomposition idea in order to offer a useful flexibility required to adapt software production in a changing world. One important contribution of case-based reasoning technology is that it allows to reduce the „update complexity“ to a smaller dimension. We will show for which kinds of application tasks case-based reasoning is more flexible than other approaches and we will illustrate this using the introduced general structure of a case-based reasoning system. From a software engineering perspective future research on case-based reasoning will deal with the analysis of which kind the „invariants of case-based reasoning“ are. These invariants need to be standardised, as well as the corresponding methods. As a conclusion we will draw the attention to some points which seem to be important for future directions in research on and applications with case-based reasoning technology.

1. Introduction

The classical view on software products is mainly static. In this view a piece of software remains mainly unchanged after its production and maintenance problems are primarily considered as unpleasant additions (despite the fact that they constitute a major financial factor). The static view on software products comes from the underlying idea that a software product is the implementation of an algorithm, which solves the given problem once and for all. Many such products have their merits anyway not due their algorithmic ingenuity but due to the fact that a huge accumulation of relatively simple steps is organized in a clear structure. Within this scenario there is no need for any modification of the product. There are basically three reasons for considering software products as dynamic objects which undergo various modifications during their lifetime:

1. Changes are quite normal in real-life problems. Many things may change over time: External entities, wishes, requirements and, last but not least, the knowledge of the problem solver who
understands the problem better and better (cf., e.g., Schumeyer, 1995). We call this the maintenance problem (in a somewhat generalized sense).

2. One wants to use experience gained in the development of similar past products. Because the actual situation differs somewhat from the previous one the product cannot be taken over without changes in order to meet the actual requirements. Such experience is up to now mainly in the heads of humans and therefore not accessible to other persons. We call this the reuse problem.

3. One wants to improve the quality of previous products.

All three problem types result in the desire for regarding software products as very flexible objects. There is, however, an increasing level of difficulty in these problems. In (1) the possibility of modification is needed. In (2) there is an additional retrieval problem involved because the most suitable experience has to be found. In (3) there is an additional evaluation and learning problem in order to detect weaknesses of the past solution and learning how to do it better. This is the quality control problem.

As a consequence, in past years this perspective on software products has radically changed. Software products – built with methods of modern software engineering – are treated as dynamic objects, similar to living organisms. This has a parallel in the treatment of business and general economic and technical processes and here is in fact one of the driving forces to the new developments located. There are various attempts (cf., e.g., Scheer, 1994) to handle problems arising here.

The demands on flexibility essentially mean that the necessary changes of the system can be obtained with relatively small effort. This implies that most of the work spent on the development of the system can be reused in a new problem situation.

The three problems mentioned above, in particular the maintenance problem are not new and have been attacked often in the past. In an abstract setting one major difficulty is to keep consequences of modifications under control: If an entry A at some position is replaced by B, what else has to be changed? If we denote these consequences by Con(A,B) then Con(A,B) has to be controlled. One way to express this is that one wants to separate Con(A,B) clearly from the rest of the system.
A major attempt in this direction was the development of expert systems. The separation of knowledge base and inference engine seemed to indicate that this was a key step: Changes had to be made in the knowledge base only and their control should be easy because of the declarative character of the knowledge representation. The separation idea, however, worked only in relatively simple situations. This was mainly due to two reasons. Firstly, the knowledge base contains often procedural knowledge in an implicit way and secondly the inferences drawn by the system depended not only on the meaning of the knowledge units but also on the particular representation. An example is the early expert system R1 which was finally discontinued due to maintenance problems.

In this article we will discuss the role of case-based reasoning (CBR) – a new emerging technology that solves all three problems by storing, retrieving and adapting similar past cases – in this new light. CBR is a flexible approach to software development that has overcome the indicated difficulties to a large extend. It has proven to be of practical importance by a large number of industrial/business applications (Allen, 1994; Houlder, 1995; Schult, 1995; Althoff, Auriol et al., 1995; etc.)

The advantages of CBR in this context are that the involved processes as well as the representation formalisms can be decomposed which allows that they are easily and independently adapted to new purposes (see section 7).

In particular we will point out in which way CBR takes up the separation issue by a certain decomposition idea in order to offer a useful flexibility required to adapt software production in a changing world.

We will describe what is fixed within the CBR approach to software development and what can be changed. In this context we will introduce the main parts of a CBR system.

The leading aspect of this article is not to show that CBR as a problem solving and learning method has e.g. a better classification accuracy than competitive approaches from clustering,
neural nets, statistics, or machine learning tailored for particular applications although this may happen for certain application tasks. We will rather motivate that CBR is a generic methodology for software development and to prove this using software engineering criteria.

CBR is a technological means to put a generally accepted view into practise, namely that the development of software products should not be understood as a creative act but as an engineering activity that needs to be industrialised such that it can be produced with minimal costs (Quack, 1996, cf. also sections 5.2 and 6.2). One important contribution of CBR technology here is that it allows to reduce the „update complexity“ to a smaller dimension.

We will show for which kinds of application tasks CBR is more flexible than other approaches and we will illustrate this using the introduced general structure of a CBR system.

From a software engineering perspective future research on CBR will deal with the analysis of which kind the „invariants of CBR“ are. These invariants need to be standardised, as well as the corresponding methods.

2. Case-Based Reasoning Technology

We will include some basic elements and terminology of CBR in order to refer to it although it is familiar to most readers.

2.1 General Technological Issues

CBR is the most recent approach of the machine learning and knowledge acquisition fields to overcome the well known knowledge acquisition bottleneck in the development of knowledge-based systems. Sustained learning from experience enables a steadily extension and improvement of the knowledge-base (cf. Aamodt, 1995). E.g., by focusing on feedback in the basic knowledge acquisition: Quality knowledge-bases can be updated by learning from protocols that have been acquired on-line (cf. Pfeifer, Grob & Klonaris, 1995: section 4.3.1; Waeschkowsk, Schahn & Forchert, 1995). CBR is also a helpful means during the testing of the knowledge-base while using special case-data to validate the knowledge-base with respect to the underlying requirements.

CBR is a technology that offers learning from test data (cf. Althoff, 1992), learning from feedback, storing of case-data, retrieving of information, documentation of useful experiences, and constructing of up-to-date knowledge-bases for a variety of problems, especially those usually occurring in enterprises during the heavily interleaved processes of product development, production, and quality management. CBR technology offers solutions for the integration of data, information, and knowledge (cf. Aamodt & Nygård, 1995) and by this creates the opportunity to build integrated (knowledge-based) information systems for modelling, e.g., various kinds of business and general economic processes.

In the following we will as a first step contrast CBR technology with the well known database technology for a simplified – but still real-life – problem. Then we introduce basic characteristics of CBR as far as it is necessary within this article. In section 2.3 we will give an overview on real-life applications being developed using CBR. These applications demonstrate the practical relevance of CBR technology for a number of different application areas.
2.2 Exemplary Comparison between the Database and the CBR Approach

We now give a short motivation of CBR technology using the domain of „sales support for one-year-old cars“ (cf. Table 1), where the underlying goal is to give decision support for the car dealer who must offer a car to his client that meets the given requirements and wishes as much as possible.

<table>
<thead>
<tr>
<th>Kind of car</th>
<th>Client's query</th>
<th>Database query</th>
<th>CBR query</th>
</tr>
</thead>
<tbody>
<tr>
<td>Horsepower output</td>
<td>about 100</td>
<td>{80..120}</td>
<td>100</td>
</tr>
<tr>
<td>Piston displacement (cm³)</td>
<td>about 2000</td>
<td>{1800..2200}</td>
<td>2000</td>
</tr>
<tr>
<td>Price (DM)</td>
<td>cheap</td>
<td>&lt;30.000</td>
<td>&lt;30.000</td>
</tr>
</tbody>
</table>

Table 1 – Comparison of database- and case-based retrieval for used cars

We assume that a client wants to buy a used car that is about one year old that allows to carry him, his wife, and his three children. Therefore, he decides to look for a caravan. He also has some expectations with respect to horsepower output, piston displacement and, of course, the price. Table 1 shows how the car dealer could search his relational database by transferring the user’s request to a set-oriented query. Since a CBR system bases on similarity instead of exact matches only, the car dealer could more easily generate a query for the CBR system (there is no necessity to specify sets of values). By contrast, the CBR system could also retrieve „cars“ based on exact matches (caravan) as well as relations (<30.000).

<table>
<thead>
<tr>
<th></th>
<th>Database approach</th>
<th>CBR approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set of results</td>
<td>not sorted</td>
<td>sorted</td>
</tr>
<tr>
<td>Quality of the result</td>
<td>by chance</td>
<td>guaranteed</td>
</tr>
<tr>
<td>Near miss</td>
<td>possible</td>
<td>impossible</td>
</tr>
</tbody>
</table>

Table 2 – Comparison of database- and case-based retrieval

In addition, the CBR system supports the car dealer in further aspects (cf. Table 2). The set of results is sorted by similarity. He is guaranteed to find the most similar case because all attributes are used for computing the similarity including different values for the respective attributes (inexact matches). By this, the effect of near misses, namely that very interesting cars are not found because of a non-matching dimension, can be excluded. Here the car dealer is a decision maker who needs access to all relevant information without being able to specify this exactly. Hence there is a database as well as some (inexact) specification problem involved. The following aspects in the example are typical:

- Both numeric and symbolic attributes can be used.
- The values for some attribute can be weighted by a local similarity measure and combined with respect to their respective importance for the global similarity between the query and retrieved car.
For realizing the search issue data base technology can also be an effective means to implement a CBR system (cf., e.g., Shimazu, Kitano & Shibata, 1993). In this sense databases and CBR are complementary technologies. The above advantage of CBR systems stems from the fact that CBR bases on nearest neighbour search. Concerning this aspect comparison of CBR with other classification methods from statistics, pattern recognition, or machine learning would adequate. We will, however, restrict ourselves here to software engineering issues.

2.3 Basic Characteristics of CBR

CBR distinguishes several activities by storing, retrieving and adapting similar past cases. CBR appeals to those professionals who solve problems by recalling what they did in similar situations.

![Diagram of the CBR cycle]

Fig. 1 – The CBR cycle

The activities in this cycle are related to the maintenance, reuse and quality control problem as we will discuss below. At the highest level of generality, a general CBR cycle may be described by four tasks (Aamodt & Plaza, 1994): *Retrieve* the most similar case or cases, *Reuse* the information and knowledge in that case to solve the problem (a process called „solution transfer“), *Revise* the proposed solution, and *Retain* the parts of this experience likely to be useful for future problem solving (cf. Fig. 1).

Note that the tasks referred to here are internal reasoning tasks, and different from application problems tasks like diagnosis, planning, scheduling etc. referred to throughout the article. The four CBR tasks each involve a number of more specific sub-tasks. An initial description of a
problem (top of Fig. 1) defines a new case. In the Retrieve task this new case is used to find a matching case from the collection of previous cases. The retrieved case is combined with the input case - in the Reuse task - into a solved case, i.e. a proposed solution to the initial problem. The Revise task tests this solution for success, e.g. by applying it to the real-life environment or have it evaluated by a teacher, and repaired if failed. This task is important for learning, since the system needs a feedback of how successful its proposed solution actually was. Retain is the main learning task, where useful experience is retained for future reuse, by updating the case-base and possibly also the general domain knowledge.

As indicated in Fig. 1, general knowledge usually plays a part in this cycle, by supporting the CBR processes. This support may range from very weak to very strong, depending on the type of CBR method. By general knowledge we here mean general domain-dependent knowledge, as opposed to the specific domain knowledge embodied by cases. For example, in diagnosing a patient by retrieving and reusing the case of a previous patient, a model of anatomy together with causal relationships between pathological and other physiological states may constitute the general knowledge used by a CBR system. A set of rules may have the same role.

2.4 Knowledge representation and knowledge container

A common distinction is the one between explicit (or declarative) and implicit knowledge representation. It is clear that the desires for modification favors an explicit representation. In order to get changes under control structured programming suggests a strong modularization. This is domain and task dependent and relies on the assumption that they allow a decomposition. The idea of a knowledge container is somewhat orthogonal to this. Here one identifies parts which can carry knowledge for one and the same problem such that

(i) each container can be manipulated independently from the other part,
(ii) knowledge can be shifted from one container to another one.

For a CBR system we have four containers in which one can store knowledge (cf. Fig. 2): the vocabulary used, the similarity measure, the solution transformation, and the case-base. In principle, each container is able to carry all the available knowledge (but this does not mean that this is advisable). The first three containers include compiled knowledge (with „compile time“ we mean the development time before actual problem solving, and „compilation“ is taken in a very general sense including human coding activities), while the case-base consists of case-specific knowledge that is interpreted at run time, i.e. during actual problem solving. For compiled knowledge the maintenance task has similar difficulties as in knowledge-based systems in general. An important point is that the consequences of a modification are still inside of the same container. However, for interpreted knowledge the maintenance task is easier because it results in an updating of the case-base only.
We will illustrate the role of the containers in the example of used car selling from above.

(a) The maintenance problem: A change in the cars on stock results in a change of the case-base. A different user type with modified preferences leads to different weight functions, i.e. a change in the measure. It may also lead to introduce the new predicate "price per piston replacement" if this is wanted.

(b) The reuse problem: Firstly, the system might be reused by other car dealers with changes as in (a). Secondly, the system may be reused by dealers of trucks, boats or related merchandise with also appropriate modifications.

(c) The quality control problem: A quality measure here would use the satisfaction of the customers and an investigation of how often serious mismatches occurred. An improvement could result in a better similarity measure. In general, also the case-base can be improved by selecting more typical cases (in this example the cases are just all available cars).

In contrast to this we will shortly mention the knowledge containers for a knowledge-based system: The vocabulary, the knowledge base and the inference engine. The major point is that the last two cannot be manipulated independently of each other; changes in one container may very well result in changes in the other one.

In our opinion a main attractiveness of CBR comes from the possibility to modify the content of each container independently and as a consequence from the flexibility to decide pragmatically which container includes which knowledge and therefore to choose the degree of compilation in order to optimise the cost function CF that can be (straight forward) associated with the respective distribution of knowledge (C=Compilation costs; P=Problem solving costs; n=number of problems solved):

$$CF = C + n^*P$$

A general strategy for developing CBR systems is to compile as little knowledge as possible and as much as absolutely necessary. The structure of cases being essentially an ordered pair <problem,solution> has another modularization effect. The solution of one problem can be changed without effecting the solution of other problems. In addition, such solutions can range from vary simple hints to the user to arbitrarily complicated objects. In this light solutions are
components and one sees that CBR includes aspects of componentware. Learning strategies should be used to automate the compilation process as far as possible, in particular for the quality control issue (cf. Richter, 1995b).

2.5 Applications of CBR Technology

To underline the relevance of CBR for real-life application development we give a short overview to which kinds of business and industrial processes CBR already has been successfully applied (of course, we can only provide an incomplete and subjective selection). All of these applications have in common that knowledge-based systems or even traditional algorithms would also solve the problem, but their development would be very (and mostly too) cost-intensive.

A prominent application field for CBR is the area of help-desk applications (Wess, 1996), e.g., for diagnosing printer faults at the end user (Nguyen, Czerwinski & Lee, 1993), faults of software components (e.g., SAP R3 service management, cf. Bartsch-Spörli, 1996a), or for supporting call-centres where an answer must be given in a certain period of time and a problem needs to be solved on the very first call (Wess, 1996).

Another important field is maintenance support and after-sales service like, e.g., injection moulding robot diagnosis (Auriol & Manago, 1995), troubleshooting of jet engines (Auriol, Manago & Guiot-Dorel, 1995), or troubleshooting of flexible manufacturing systems (Gausemeier & Leschka, 1996; Wincheringer, 1995). An additional dimension along which CBR can be applied is on analysis tasks like failure mode and effects analysis (cf. Pfeifer & Zenner, 1995; Pfeifer & Faupel, 1993; Pfeifer, Grob & Klonaris, 1995) or top management fraud detection (Curet & Jackson, 1996). More difficult are critical decision support tasks in medicine like, e.g., therapy selection of poison cases based on psychotropes (Althoff, Bergmann et al., 1996; Malek, Daniel & Rialle, 1996), or in engineering applications (Laczkovich, 1990: type III applications, p. 103; cf., e.g., Lees, 1996).

CBR technology can also be applied to quite a wide range of assessment tasks as, for example, in credit assessment (Reinartz & Wilke, 1995), in real estate property appraisal (Lenz & Ladewig, 1996), or in inherent audit risk assessment (Bienderra & Ehrenberg, 1996), or for organisational support like, e.g., case-based software development for store management (Czap & Grimme, 1996), controlling material flow in production (Vitiello & Dangelmaier, 1996), scheduling (Schmidt & Meyer, 1996), forestry management (Althoff, Wess et al., 1995), travel agency support (Lenz, 1994), or sales support (Schild, 1991).

An important area is the reuse of complex components. For instance, basic electronic components, object-oriented software (Bergmann & Eisenecker, 1995), software modules from previous successful designs by integrating CBR and quality function deployment (Lees, Hamza & Irgens, 1996), or design requirements for information systems (Krampe & Lusti, 1996). Here CBR could be used as a means for the development of business information systems, e.g. by using the experiences collected with toolsets like ARIS (cf. Scheer, 1996).

As an addition, however, the authors want to mention that they do not consider CBR as something which is in general superior to all other methods and which is always advantageous to employ. In fact, in each application one must carefully select the solution approach; the given list is intended to give indications where CBR might be useful.
In the following we want to focus on parameters that influence the usefulness of a software product like the complexity of the problem(s) to solve, the usefulness of potential solutions depending on the complexity of the underlying problems, the development of the system's (problem solving) competence over application development time, the development of the distribution of computational effort on compile versus run time over application development time, and the system maintenance over application development time.

3. Different Kinds of Problem Complexity

As mentioned in the introduction, the static view on software products stems from the idea that pieces of software are implementations of algorithms. Algorithms are solutions to clearly defined problems (in the sense of mathematics). Once such an algorithm is found and the context does not change it can be used for ever. In real-life applications most problems are not so widely stated (cf. Fig. 3) and we encounter a complexity hierarchy of problems (not in the sense of complexity theory) which we will list now. We present each type together with the expected solution approach, an example, and a suitable application situation.

- **Atomic problem**: It is not decomposed and solved by a single algorithm.
  - Expected solution: algorithm.
  - Example: retrieving all those clients from the client database who shall receive the new marketing brochure; retrieving all clients from the client database that are best-suited to some specified criterion based on a given evaluation function.
  - Application: sort-a-z/0-9: sorting a list of references according to the alphabetic order, any sorting application.

- **Complex problem**: Due to the underlying complexity of the given problem, it cannot be reasonably solved using a single reasoning task, but it needs to be modelled using a complete reasoning meta-level.
  - Expected solution: a number of algorithms that are organised according to the given reasoning task hierarchy.
  - Example: workflow management, management of large organisations.
  - Application: software product.

- **Classes of (atomic or complex) problems**: The class of problems is indexed with some parameter, which may be structured. Each parameter value defines a particular problem. Such a class requires a class of algorithms (or software products). This class of solution algorithms may not be uniform. Some meta-level is also involved on which the question is decided which method is used for which problem, respectively.
  - Example: fault diagnostic system that may be used for various kinds of CNC machines;
  - Application: generic software product.
4. Different Degrees of Solution Usefulness

While a problem has associated the respective problem data, solutions depend on the selected software environment. A problem normally is not identical with the problem data that can, e.g., be processed by any kind of appropriate algorithm. More complex problems need more encompassing solutions that can only be delivered building a complete software product (cf. Fig. 3). This software product must meet all the given problem requirements and gets the associated problem data as input.

Obviously there exists a decreasing degree of solution usefulness (cf. Fig. 4):

- Explicit solution:
  - Applied technology explicitly achieves the solution to the given problem.
  - Example: statistics based classifier, sort algorithms, search algorithms.
• **Adaptable solution:**
  - Applied technology achieves an approximation of the solution which can be transferred into an explicit solution of the given problem by use of explicit knowledge and/or ascertaining additional information.
  - Example: case-based planning/diagnosis/decision support system.

![Diagram]

*Fig. 4 – Decreasing degree of solution usefulness*

• Solution hint:
  - Applied technology obtains information that is useful according to the semantic criteria of the user.
  - Example: information retrieval system, case retrieval system.

• Any kind of information:
  - Applied technology obtains information by browsing through a huge amount of data.
  - Example: employees of a company that uses the wrong technology for the management of information.

5. **Selected Qualities of a Software Product**

In this section we will shortly introduce four selected qualities which a software product should have, namely the usefulness of a presented solution depending on the complexity of the underlying problem, the development of the problem solving competence of a software product over application development time, the development of the distribution of the computational effort on compile time versus run time over application development time, and the maintenance effort for the software product over application development time. In section 6 we will analyse the contributions of CBR technology with respect to these qualities. Additional qualities of a software product and the respective contribution of CBR will be discussed in section 7.

5.1. **Solution Usefulness**

For precisely stated atomic problems usually only precise, correct, and complete solutions are accepted. With increasing complexity such solutions are often not expected and can often not be given. The best one can do is to look for solutions with a lower degree of usefulness. Fig. 5 reflects roughly these relations.
A consequence for the solution approach is here that it should be able to react over time on a decreasing problem complexity with an increasing degree of solution usefulness.

5.2. Problem Solving Competence

When people solve problems their competence increases over time until it reaches a level of saturation as indicated in Fig. 6.
There are two ways to react here. For many (in particular very critical) situations there is no other choice then to wait until a minimal degree of competence is reached. Often, however, an immediate response is expected, even on a low level of competence. This requires ideally for the software support an incremental increase of the solution power, reflecting at each time the available competence (cf., e.g., Quack, 1996).

Higher solution power does not only mean that more problems can be solved but also that problems can be solved better. Here „better“ can mean that a higher „degree of optimality“ can be reached, and it can also mean that less mistakes are made. In many situations such mistakes can also be measured by some cost function. Then the increase of the competence indicated in Fig. 6 implies a decrease of this cost function. The natural requirement for the software development over time is to aim at such a cost reduction.
5.3. Distribution of the Computational Effort on Compile versus Run Time

This point is similar to the above one but is concerned with run time effort. The more time can be spent for the system development (i.e. at compile time), the more should be the actual run time reduce, as indicated in Fig. 7.

The consequence for the solution approach is again that the compilation effort should smoothly be transformed to run time efficiency, rather than in new versions of radically different character.

5.4. Maintenance Effort

It is well known how maintenance effort develops over time. We base on an analogy to the product life cycle, as for instance described in Wöhe (1986, p. 627), and list in short the main points (cf. also Fig. 8) by differentiating between four basic phases:

I) Removing bugs and meeting requirements that are not yet covered leads to a steadily increasing maintenance effort directly after introducing the software product to the market (corresponds to the introductory phase in the product life cycle; cf. Wöhe, 1986: p. 626).

II) The maintenance effort increases slower and then starts to go down (corresponds to the growth phase in the product life cycle; cf. Wöhe, 1986: p. 627).

III) The maintenance effort steadily decreases but after some time it again starts to increase because of (more and more) changes of the underlying domain requirements (corresponds to the maturity phase in the product life cycle; cf. Wöhe, 1986: p. 627).
IV) From then on the maintenance effort steadily increases (corresponds to the saturation phase in the product life cycle; cf. Wöhe, 1986: p. 627f). From a certain point of time on, in particular when requirements or the context changes, maintenance costs may go beyond certain thresholds, i.e. the benefit is smaller than the costs. So, maybe a new system must be developed, or no service can be given any more (corresponds to the degeneration phase; cf. Wöhe, 1986: p. 628).

Fig. 8 indicates that there are two possibilities to reduce maintenance costs. Techniques for lowering the first maximum are beyond the scope of this article. The increase at the end of the curve is again due to external changes and asks for flexible approaches like the previous arguments.

6. The Contribution of CBR Technology

In the previous sections we have listed some properties of problem types, difficulties arising from them, and demands on solution approaches, in particular from a practical point of view. All the difficulties have in common that the task and the context changes, but one still wants to make use of the labour investigated so far. Since the task is not continuously performed by single persons, it is not sufficient that the persons make use of their experience but one wants to make use of the developed software as well. Therefore, the software should change smoothly and it should not be too difficult to perform changes.

In this section we will discuss the flexibility of CBR with respect to the arising challenges. The flexibility of CBR seems to be a consequence of its basic structure. Usually a case is described as an ordered pair (problem description, solution). According to sections 3 and 4 a first step would be to allow changes of a „problem“ and a „solution“. For instance, instead of a solution an arbitrary piece of information could appear.

This would reduce the adaptation of the whole system to the adaptation of the entities in „problem“ and in „solution“, leaving the overall structure invariant.

A second degree of freedom becomes apparent when one looks where the knowledge in a CBR system is located. In section 2.2 we introduced the four containers in which one can store knowledge (cf. section 2.2). In principle, each container is able to carry all the knowledge. The point is now that one can shift knowledge from one container to another one, again without changing the overall structure of the system. E.g., in the beginning one would use many cases; many of them can be removed when understanding of the problems grows and results in a more sophisticated similarity measure.

We will now discuss such possibilities along the lines indicated in section 5.

6.1. Solution Usefulness

CBR can contribute to all three kinds of solutions: explicit, adaptable, and hints. If all the knowledge is distributed on the three containers for terminology, similarity measure, and case-base, the CBR system can deliver an explicit solution (in case the represented knowledge is sufficient to solve the problem). If the application is more in a way that a large number of solution variants exists and the necessary knowledge for solution transfer is included in the respective
container, the CBR system can present adaptable solutions that can be inferred automatically or stepwise in co-operation with the user. Therefore, CBR is applicable to a wider range of problems. If the query can only be roughly described and the user's point of view may change during his work with the system with the consequence that (s)he incrementally builds (or radically changes) her/his query, the CBR system can still deliver some support in the form of solution hints by retrieving a sorted list of cases that reflects the knowledge available in the respective containers as well as the degree of precision of the query.

6.2. Problem Solving Competence

In the beginning where the competence is low one has

- cases with only moderately good solutions,
- a poor solution adaptation, and
- a simple similarity measure.

Nevertheless, one gets certain solutions of the problem when urgently needed. Increasing competence can smoothly be obtained by improving the solution adaptation, improving the similarity measure, introducing better cases, and removing bad ones (cf. Fig. 9). All these activities can result in an improvement of the associated cost function indicated in section 5.2.

![Image](image.png)

Fig. 9 – Expected versus CBR problem solving competence

6.3. Distribution of the Computational Effort on Compile and Run Time

The knowledge in the vocabulary, the solution adaptation, and the similarity measure is dealt with at compile time, while the case-base is used at run time. The treatment of the case-base is a source of inefficiency (cost for searching cases, computing similarity values etc.). An improvement of a CBR system will usually lead to lower run-time costs because one has fewer, better structured, and more easily accessible cases in the (possibly smaller) base. Therefore, CBR offers the
possibility of deciding how much computational effort is spent during system development (corresponds to compilation effort) or during problem solving time (corresponds to interpretation effort) (cf. Fig. 7).

6.4. Maintenance Effort

The CBR maintenance effort is nearly constant during application development, maybe slightly increasing (cf. Fig. 10; cf. also Bartsch-Spörl and Manago, 1996; Bartsch-Spörl, 1996).

![Graph showing maintenance effort over application development time]

Fig. 10 – Expected maintenance effort versus CBR system maintenance effort

7. The Potential Role of CBR Technology in Software Development

Though there exist prominent approaches for the knowledge engineering of knowledge-based systems like, e.g., CommonKADS (cf. Wielinga, Van de Velde et al., 1993), the potential usefulness of CBR has not been recognised by the authors of such approaches. For CommonKADS this is partially reflected by the fact that CBR technology is not covered in the CommonKADS library (cf., e.g., p. 159 in Breuker & Van de Velde, 1994, on the assessment task that is at most comparable to CBR where no link exists between case descriptions, i.e. the case-base, on the one side, and the measurement system on the other side, which in fact is an essential part of CBR and not just a matter of terminology).

We do not compare CBR with neural nets on the classification task, with information retrieval on the retrieval task, or with fuzzy sets on the matching membership task (which we could in fact do because there is a certain degree of overlapping; cf., e.g. Althoff, 1996; Althoff, Bergmann et al., 1996), but we compare CBR only with technologies that cover the whole range of tasks necessary to be a software engineering methodology (or at least a non-trivial part of it). By this we want to demonstrate that CBR is a generic software development methodology.
As a completion to sections 5 and 6 we will demonstrate this for the following aspects of software product quality (cf., e.g., Meyer, 1988; Nagl, 1990):

**Correctness:** The precise fulfilment of requirements that are given by the specification:
- CBR allows to approximate correctness by including more and more correct cases.

**Robustness:** Keeping workability also for exceptional cases:
- Exceptional cases can be easily included if they are known, e.g., directly when they occur during testing.

**Extendibility:** The possibility to easily adapt a software product to a modified or extended specification (e.g., by simplicity of the design, autonomy and decentralisation):
- CBR can be easily extended by new cases.

**Reusability:** The possibility to completely or partially reuse a software product for a new application (influences other aspects to a high degree):
- The case-base could be reused for a different task, e.g., based on a different similarity measure, and/or different levels of description/abstraction (Wilke, 1995; Bergmann and Wilke, 1995; Bergmann, 1996).
- „Reuse“ is an explicit phase in the CBR cycle according to Aamodt and Plaza (1994). CBR applied to a case-base on explicit designs of software products leads to situation that the reuse process of CBR is a contribution to software reusability (cf. Krampe & Lusti, 1996; Bergmann & Eisenecker, 1995; and Scheer, 1992, for an interesting application field).

**Compatibility:** Ease with which different software products can be connected with one another (e.g., uniformity of the design, standards):
- We refer to Althoff, Auriol et al. (1995) or Bartsch-Spörl and Woltering (1996) for the openness of current industrial CBR tools and their compatibility with commercial standard software. An object-oriented case description language (CASUEL) that is supported by a number of tools is described in Manago, Bergmann et al. (1994).
- In general a CBR system can be a black-box pre-processor for any kind of problem solver to speed up the overall problem solving process. The generic problem solver is then used to „fill“ the solution transformation container of the CBR system, i.e. the problem solver is only needed if a retrieved solution needs to be adapted. The case-base is filled with successfully solved problems (together with their solutions).

**Modular decomposability:** This criterion is fulfilled if the used design method supports the decomposition of the problem into sub-problems that can be processed independently:
- In general the knowledge is decomposed into the four knowledge containers: terminology, similarity measure, case-base, solution transformation.
- The problem solving process is decomposed into the main phases retrieve, reuse, revise, and retain (cf. Aamodt & Plaza, 1994).
- The problem solving process is based on the case-base that is easily decomposable into the respective cases.
Moreover, reasoning can even be decomposed onto sub-cases (cf. Holz, 1996), especially if the underlying representation is object-oriented as it is, e.g., the case with the CASUEL case representation language (cf. Manago, Bergmann et al., 1994).

Modular combinability: This criterion is fulfilled if software elements can be freely combined (goal: reusability; toolbox style):
- For every container different content can be used and freely combined with the realisations of the other containers.

Modular understandability: A module should be easy to read and, to understand it, only small knowledge about neighbouring modules should be necessary (problem of maintenance):
- Each case can be understood as such (the other containers include compiled knowledge which might be not so easily understandable)

Modular continuance: Small changes of the specification of the problem should effect only one or a small number of modules:
- Cases can be directly changed, added, or removed (changing parts of the other containers is more costly).

8. Discussion

CBR is an integrative methodology for software product development. The basic structure of a CBR system can be described by the four container model (cf. section 2.2). The development of a CBR system is oriented towards an incremental development of prototype systems with a stepwise validation (Bartsch-Spörl & Manago, 1996). This corresponds to the well-known spiral model (Böhm, 1988). Several ongoing project activities explicitly deal with building a concrete and detailed methodology for CBR application/systems (Althoff, Bergmann et al., 1995; Manago, Althoff et al., 1995; Althoff & Bartsch-Spörl, 1995; Althoff & Bartsch-Spörl, 1996; Althoff & Aamodt, 1996; Bartsch-Spörl & Manago, 1996).

CBR technology offers the flexibility of solving problems of different complexity without changing the underlying technology. By this CBR often allows to improve the underlying cost-benefit ratio for the development of real-life applications.

CBR can be used as a means to approximate expert reasoning. This means that reasoning from cases is understandable for the expert, though in practice he may rely on other kinds of reasoning and use CBR only when reasoning from exceptions (e.g., Althoff & Wess, 1991; Larichev, 1995).

CBR technology offers the flexibility of solving problems of different complexity on different levels of abstraction in the description languages (cf. Bergmann, 1996). Different views on the same case-base can be used to deal with different aspects of a problem. Depending on the user and the respective context, the degree of detail of the information used to retrieve and display the most similar cases can vary (cf. Wilke, 1995).

In the sections above we have indicated how and why CBR systems can meet modern demands of software engineering. As mentioned in the beginning, such possibilities are not in competition with tailored solutions for special problems. The research and applications of CBR technology in the past years shows, however, that the general expectations on CBR are well founded, both in a
theoretical and in a practical sense (Richter, 1995a; Althoff, Auriol et al., 1995; Althoff, Wess et al., 1995; Bartsch-Spörl and Wess, 1996; Bartsch-Spörl, 1996b). For the practical side we mention as widenesses for such a claim the applications cited in section 2.3.

We are concerned about three issues connected with the distribution of knowledge:

- Semantics: "What is the meaning of the representation language" used in the containers described in section 2.2.
- Knowledge engineering: What must be done in order to distribute the knowledge (in the best way)?
- Maintenance: How to react on dynamic changes of the knowledge and the context?

There is the demand of developing terminology maintenance (cf. Richter, Schmidt & Schneider, 1995), similarity maintenance (cf. Althoff, Bergmann et al., 1996, for a practical example), case-base maintenance (e.g., by varying or extending the IBL algorithms: Aha, Kibler & Albert, 1991), and the solution transformation maintenance. All these issues are not independent. Maintenance relies heavily on knowledge engineering and both must focus on what is semantically relevant. For all these issues we should make further advantage of the flexibility of CBR based on the four-container concept.

9. Conclusion

As a conclusion we will draw the attention to some points which seem to be important for future directions in research on and applications with CBR technology.

- The invariant structures of a CBR system should be improved and standardised.
- The tools and methods for improving a CBR system over time should be structured and the tools should again be standardised.
- The standardisation should be accompanied by a clear structure for a cost analysis.
- For important areas of applications more or less fixed rules should be developed for the application and the improvement of a CBR system.

The authors believe that CBR has a promising future and will become a major method in software engineering.

10. Acknowledgement

The work described has been partially funded by the „Stiftung Rheinland-Pfalz für Innovation“ within the project WiMo (Modelling and Acquisition of Knowledge for Case-Based Reasoning). The authors would like to thank Martin Bräuer, Stefan Dittrich, Christoph Globig, Hector Muñoz-Avila, and Stefan Wess for many intensive discussions and, in addition, Brigitte Bartsch-Spörl, Ralph Bergmann, and Ralph Traphöner for comments on earlier versions of this article.
11. Literature


Aamodt, A. and Nygård, M. (1995). Different roles and mutual dependencies of data, information, and knowledge - an AI perspective on their integration. Accepted for publication in Data and Knowledge Engineering


Althoff, K.-D. and Bartsch-Spörl, B. (1995). Call for Participation: Decision Support for Case-Based Applications. Initiative for systematically collecting information about existing CBR systems and applications, BSR Consulting (Munich) and Centre for Learning Systems and Applications (LSA) at the University of Kaiserslautern

Althoff, K.-D. and Bartsch-Spörl, B. (1996). Decision Support for Case-Based Applications. Wirtschaftsinformatik 1/96, special issue on case-based decision support (edited by D. Ehrenberg), 8-16


