# An Application of CBR in Design System Generation

**Ming He & David C. Brown**

Computer Science Department,
Worcester Polytechnic Institute,
Worcester, MA 01609, USA.
*mhe@cs.wpi.edu; dcb@cs.wpi.edu*

**Abstract.** In this paper we are concerned with generating parametric design systems based on cases from existing design systems. Issues that distinguish this particular application of CBR in design from those commonly seen are highlighted. In summary: we are trying to reuse past knowledge of the design process rather than design results; Adaptation could be non-trivial and crucial to accomplishing this goal; The reuse is carried on in a decomposition hierarchy with subcase reassembly involved; Various other kinds of knowledge such as design object and decomposition knowledge play an important part throughout each stage of the CBR process.

## 1. Introduction

The background of this ongoing research is in the area of parametric routine design problems [Brown 1996]. A parametric design problem can be characterized as one in which the design object is represented by a set of parameters and the result of the design is a set of value assignments to the parameters. By routine design problems we mean a type of design problem for which problem decompositions, design plans, as well as design control knowledge, such as failure recovery, are known. Such a routine design is a process of choosing and searching in a space delimited by the above available knowledge.

The goal of our research is to experiment with the ways in which a design system described above can be generated automatically in response to given design specifications. We manage the complexity in a divide-and-conquer way:

a. decompose the given design problem,
b. generate the design problem solvers for the decomposed subproblems and recompose them in the original decomposition hierarchy.

The problem decomposition part has been tackled by Liu [1993] in which a system called KDD (Knowledge-based Design Decomposition) is shown to be able to produce a hierarchy of subproblems for a given design problem. The decomposition KDD achieves is based on object knowledge and heuristics, with a goal of minimizing design-time backtracking. The output of the KDD system suggests a framework for the target system to be built. We need to add problem solving, or design knowledge into this framework to construct a complete design system.

The focus of this research therefore lies on the generation of design plan and steps, where design knowledge is embedded, in a form that is executable. In another words, our problem is essentially to extract design knowledge, which is expressed in an appropriate form and at an appropriate level, out of available sources of knowledge.

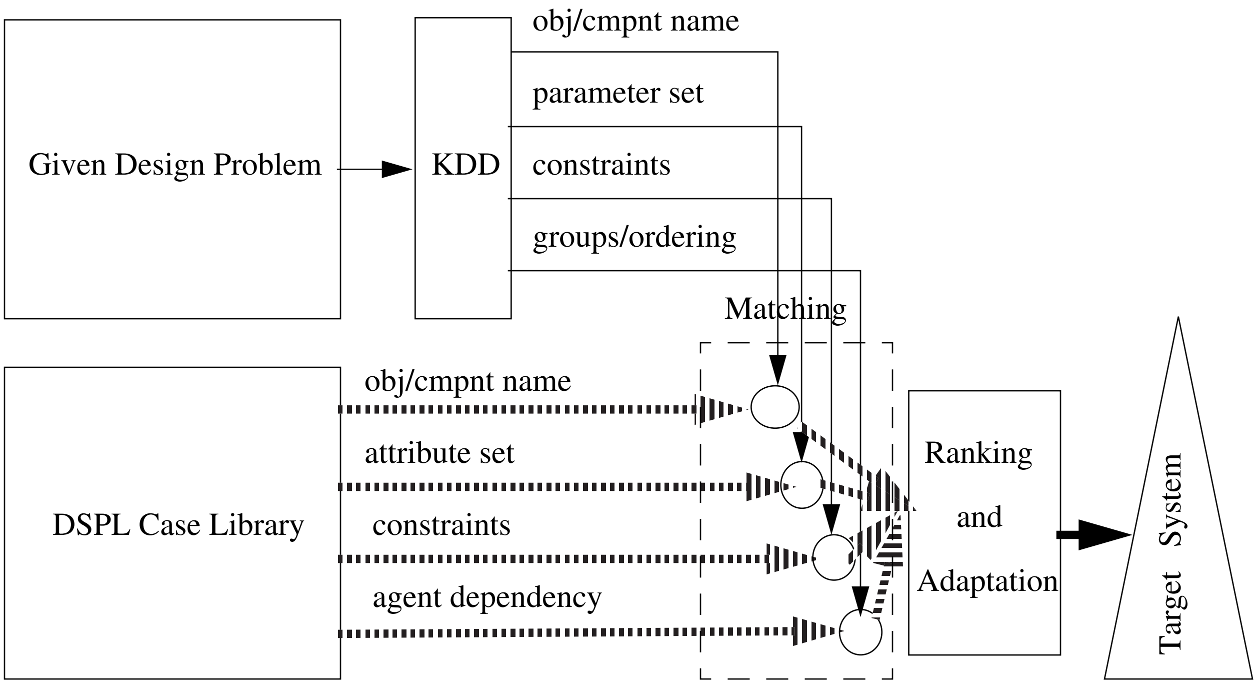
Correspondingly the knowledge needed for generating such a system can be divided into two classes: the knowledge about the design object, and knowledge about design problem solving. The object knowledge is descriptive, and is used mainly in decomposing the problem and validating the design result. The design(ing) knowledge, believed to be derived from the object knowledge and experiences, constitutes the rest of the target design system.

It is difficult to convert design object knowledge directly into the design knowledge we need in the target system. There is a big gap between their forms, and the gap can not be easily overcome with current techniques such as knowledge compilation. Hence, we turned to reuse of knowledge that has the same form as our target architecture, that is reuse of existing design systems or system components (i.e., pieces of "how to design" knowledge). Since the cases we are reusing are usually poorly justified by formal theory, CBR is a more suitable method than other types of reuse such as Explanation-Based Reasoning and Model-Based Reasoning.

The target language we use is DSPL (Design Specialists and Plans Language) [Brown 1989]. DSPL is devised for building routine design expert systems. It supports expressing the design knowledge in a problem decomposition hierarchy of design subsystems called agents. Different types of DSPL agents exist for different roles in the system. Some important types of agents are: *Specialists* that select a plans and refine the design; *Tasks*, used by specialists, are responsible for parameters of a component; *Steps* that produce a values for an individual parameters; and *Constraints* that are checked to detect incorrect design decisions.

## 2. Overview of the approach

For a given design problem, the retrieval module extracts a DSPL case from the case library. The case could be either a complete case, or a piece out of a complete case, depending on the best match of current situation. After evaluation of the fitness of the retrieved case, the case is passed to case-fixing module for adaptation, or to synthesis module if a new piece is needed. This process iterates until all the required cases are obtained. The assembly module assembles the cases according to the KDD output to produce the target system. See the figure below.



The application of CBR in this particular problem [Bardasz 1991] has the following features:

- The cases are plan-like. What we want to reuse are more like plans than ordinary results or objects represented as data. This implies that in many aspects it is different from conventional CBR concepts.
- Adaptation is important and difficult. We are going to have a relatively poor source of the cases. So for not perfectly matched cases we have to do adaptation.
- There might be times when adaptation or CBR cannot solve everything, and non-CBR methods will be used.
- Subcases and case reassembly is involved. We are reusing the cases in the context of problem decomposition knowledge.

## 3. Case Representation and Organization

### 3.1. Case Representation

Given the task of generating a design system in DSPL, we are interested in reusing existing DSPL systems or segments of such systems. So a case is composed of a DSPL agent and additional information that would help retrieval and adaptation.

An original DSPL agent consists of a hierarchy of subagents used by it, and optional pre-conditions for the agent to be applicable, and post-conditions describing the design status after the agent is done. All these are useful and will be kept in the case.

Each DSPL program has a design data base that serves as a repository for the design. The design object, its components, and attributes, are described structurally in the data base. From there we get the description of the object to be designed by the agent.

Design problem specifications, if available, are attached to the case to describe the range of problems that can be solved by the case.

Ideally we need more information, such as justifications for the design method and constraints used by an agent, in order to support better adaptation, but deriving this kind of information is hard. We will start with the basic knowledge initially, and experiment with adding some additional knowledge for a more powerful system.

In summary, a DSPL case is a DSPL agent hierarchy, with:

| | |
|---|---|
| *Object description:* | The object.<br>Its components<br>Attributes of the components. |
| *Agent description:* | Pre/post-conditions.<br>Parameters designed, range of values chosen.<br>Governing constraints.<br>Corresponding object/component. |
| *Problem specification:* | Constraints on a set of attributes. |

### 3.2. Case organization

Cases that contain complete DSPL systems are of the case grain size most suitable for reuse.

Whenever there is a complete case similar enough to the given problem, we try to use it. But we will have to use parts of some cases when there is no good match in the case library. The unit of reuse is then a DSPL agent that belongs to and exists in a complete case. There is no additional or different representation for the subcases, because all subcases or agents are stored and indexed in their original agent hierarchy or DSPL system. The retrieval of subcases will be the same as complete cases except that we only turn to agents at a lower level when there is no good case at their parent level.

Using complete cases could avoid problems with subcase reassembly, could avoid runtime inefficiency caused by poor coordination of agents that originate from different hierarchies, and should make the case retrieval more efficient and effective. Using partial cases may *sometimes* reduce adaptation complexity with each smaller case, and make better use of the case library. However, we will prefer complete cases to subcases at retrieving time.

The cases are organized in a flat memory structure, stored by component names for quick focus. This simple organization suffices to support the retrieval mechanism we are using.

## 4. Case Retrieval

### 4.1. Case Indexing

Without introducing justification/explanation information at this phase of the research, the best way to describe a DSPL program is by the design problem(s) it solves. Now that the design problem and subproblems are described by object knowledge, design requirements and KDD output, the case indexes are chosen to be:

| | |
|---|---|
| *Problem specification:* | Object (component) name/type<br>Parameter set<br>Design requirements<br>Agent pre/post-conditions |
| *KDD description:* | Parameter ordering<br>Parameter grouping |

Here the problem specifications are direct descriptions of the design problem. The KDD output provides a view of the problem in terms of decomposition. The degree to which the case matches the KDD output might be a confirmatory measurement of the closeness of the case to the problem. Moreover, since KDD output works as a potential framework of the target system, the closeness of the case to KDD output could also reflect the amount of adaptation required.

### 4.2. Case Retrieval

The retrieving procedure starts with top level cases or agents. If there is a match it is used. Otherwise, use the next level decomposition from KDD, try to retrieve a (sub)case for each subproblem in the KDD hierarchy, using the same strategy. Retrieval stops when a (sub)case has been found for each (sub)problem, or fails after trying the bottom level for certain subproblem.

For each retrieval, there are two steps:

a. *Quick selection:* exact matching by object/component/attribute name. This produces a set of cases for the current (sub)problem.

b. *Case matching:* measured and ranked by:

the overlapping between the parameter set of the case, and the (sub)problem;
the agents pre/post condition if any;
the number of parameters in the KDD framework that do not violate the design dependency required by the case;
the degree of agreement of KDD subproblem grouping with the subagents of the case.

Retrieving continues at a lower level of decomposition in the KDD hierarchy for those that fail to get a case match at the current level.

## 5. Case Adaptation

For a retrieved DSPL agent, some adaptation may need to be done to it before it is reassembled with other subcases and put into use. We are more concerned with adaptation that increases the applicability of the cases than adaptation that repairs failures that potentially occur when the system generated is *doing* design, because of the complexity of case evaluation/simulation and blame/credit assignment ability required for the latter. By "applicability" we mainly mean the syntactical applicability of a case, which in our case include:

a. the case matches the indexing features, an estimation of the applicability of the case, better.
b. the case has to be operational in the new context. More specifically, there should be no variable reference problem.

At the same time we have noted that adaptation at semantic level is necessary for a better validity of the system generated, and it is yet to be studied in the future.

**Adaptation type one** fixes the retrieved case in a direction such that it would match the given (sub)problem better in terms of the indexing features. In the indexing features, the name and parameter set matching are taken care of at retrieving phase, which leaves us mismatches of the following categories.

a. Mismatches in design requirements and agent pre/post-conditions, which are in the form of constraint mismatch:

   New constraints can be constructed and inserted at the point where all variables involved are available.

   Extra constraints can be discarded from the case, but it is hard to determine whether a constraint is derived from some other given constraints, that is, whether a constraint is really "extra".

   It requires more effort to deal with a constraint in the case that differs from that of the problem only by details such as a constant parameter. Not only should we modify the constraint, we have to propagate this change in the whole system. Here we need more knowledge about the origins and relationships of a constraint relative to the rest of the hierarchy.

b. Mismatches in parameter ordering and parameter grouping:

   If KDD framework is used, it must be adapted to fix any ordering it gives that violates the dependencies required by the matched subcases.

Except for the dependency violation orderings, ordering and grouping problems usually do not affect the functionality of the system. We would generally stick with the grouping given from the cases because it usually reduces the amount of redesign, while use ordering from KDD because it is supported by the KDDs decomposition knowledge and heuristics.

**Adaptation type two** solves the name reference problems in the case.

a. As with type one, extra parameters can be removed if they are proved to really be extra.

b. For a reference to a parameter that is not in the hierarchy, a new agent should be synthesized. Note that this is after we fail to find a case to borrow. We have to do this via some non-CBR method. Hopefully the use of CBR will limit this kind of situation to single DSPL steps. Under this assumption, we are filling some gaps in the target system hierarchy with the help of neighboring agents. Techniques such as symbolic manipulation and constraint propagation will be applied to simplify the constraints on the value of the parameter, and either a calculation or a selection method can be generated.

## 6. Conclusion

This paper presents a description of work in progress. The goal of the research is to develop methods to build knowledge-based design systems by completing a design decomposition "skeleton" using cases made up of smaller complete or partial knowledge-based design systems.

The CBR method is used to find cases to reuse and to guide adaptation. This work is distinguished by the need to assemble case fragments after adaptation. Much more work remains to be done, but we are confident that many interesting and challenging problems will be raised by this work.

## References

D. C. Brown & B. Chandrasekaran, *Design Problem Solving*, Pitman, London, 1989

D. C. Brown, Routineness Revisited, In: *Mechanical Design: Theory and Methodology,* (Eds.) M. Waldron & K. Waldron, Springer-Verlag, 1996, pp. 195-208.

J. Liu, *The compilation of Design Decomposition Knowledge*, Ph.D. Dissertation, CS Dept., Worcester Polytechnic Institute, 1993

T. Bardasz, *Mechanical Design Automation via CBR*, M.S. Thesis, ME Dept.,Northeastern University, 1991