The Roles of Adaptation in Case-Based Design*

Thomas R. Hinrichs and **Janet L. Kolodner**
School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

Abstract

Many design tasks have search spaces that are vague and evaluation criteria that are subjective. We present a model of design that can solve such problems using a method of plausible design adaptation. In our approach, *adaptation transformations* are used to modify the components and structure of a design and constraints on the design problem. This adaptation process plays multiple roles in design: 1) It is used as part of case-based reasoning to modify previous design cases. 2) It accommodates constraints that arrive late in the design process by adapting previous decisions rather than by retracting them. 3) It resolves impasses in the design process by weakening preference constraints. This model of design has been implemented in a computer program called JULIA that designs the presentation and menu of a meal to satisfy multiple, interacting constraints.

Introduction

In Artificial Intelligence, design has typically been classified as being either the *selection* of components to instantiate a skeletal design [Ward & Seering, 1989], the *configuration* of a given set of components [McDermott, 1980], the fixing of numerical *parameters* [Brown & Chandrasekaran, 1985, Mittal & Araya, 1986], or the *construction* of designs from scratch [Tong, 1988]. While useful for routine sorts of design, these rigid classifications do not begin to capture the flexibility that real designers exhibit [Goel & Pirolli, 1989]. For high-level conceptual design especially, these tasks are often inseparable.

In addition, many design tasks are *ill-defined*; they have search spaces that are vague and evaluation criteria that are subjective. This is in part because design categories may be defined not in terms of necessary and sufficient conditions, but instead by experience and expectations. For such tasks, it is unreasonable to assume that a designer can systematically enumerate possible designs. Architectural design, for example, cannot easily be reduced to searching a discrete problem space of alternative configurations or components.

Moreover, real-world designs may have requirements that change over time. In fact, for relatively complex problems such as architectural or aerospace design, specifications are updated constantly as the solution is refined. A problem solver should be able to accommodate such changes with minimal disruption. Part of the solution is to use dependency-directed backtracking [Stallman & Sussman, 1977], but even that can be too heavy-handed if decisions have many consequences.

In this paper, we present a strategy for automating high-level design that addresses these three critical issues:

1. How can a design architecture transcend the rigid classifications of the design task (i.e., selection, configuration, parameter fixing, and construction)?
2. How can a design problem solver avoid searching problem spaces that are vague and ill-defined?
3. How can the introduction of new constraints late in the design process be accommodated in a computationally efficient way?

Our solution to these problems is a model of design that incorporates case-based reasoning [Kolodner *et al.*, 1985, Hammond, 1989], constraint posting [Stefik, 1981a], and problem reduction. Central to this model is a process of plausible *design adaptation*. Design adaptation is a heuristic debugging process that takes as input a source concept, a set of constraint violations and a set of adaptation transformations and returns a new concept that satisfies constraints. This process plays multiple roles in our model of design: 1) It is used as part of case-based reasoning to modify previous design cases. 2) It accommodates constraints that arrive late in the design process by adapting previous decisions rather than by retracting them. 3) It resolves impasses in the design process by weakening preference constraints.

We have implemented and tested our model of design in a computer program called JULIA [Hinrichs, 1988, Hinrichs, 1989] that interactively designs the presentation and menu of a meal to satisfy multiple, interacting constraints. Meal planning entails the *selection*

*This research was funded in part by NSF Grant No. IST-8608362, in part by DARPA Grant No. F49620-88-C-0058

of dishes, the *configuration* of those dishes and courses within a meal, the *parametric manipulation of quantities* such as cost and calories, and occasionally the *construction* of new dishes to meet idiosyncratic constraints. Design is done in JULIA at the request of a client who both approves selections as the design of a meal progresses and often adds information late in the design process.

In the next section, we present a series of examples that demonstrate the use of adaptation in JULIA. Section 3 describes the different roles that adaptation plays in our model of design. Section 4 presents an overview of how JULIA works and section 5 summarizes an experiment that shows the effect of adaptation on different types of problems.

Examples from JULIA

In the first example, we see JULIA dealing with an *under-constrained* problem. It is asked to plan a meal for which there are enormous numbers of satisficing solutions. In this example, JULIA uses adaptation for several different tasks: to specialize the structure of the meal, to change the structure of the meal in response to a contradiction, and to change a component in the meal, again in response to a contradiction. JULIA deals with the enormous search space by using a suggestion made by its case-based reasoner as a starting point and adapting it to fit the constraints of the new situation.

Problem 1

Plan a meal that is cheap, easy to prepare and includes tomato and cheese in its main dish. JULIA begins by choosing a cuisine. It remembers several cases, some of which are Italian and some Mexican. It makes both suggestions to its client and asks for a preference. The client chooses Italian.

A meal is generally composed of an appetizer, salad, main course, and dessert, but an Italian meal is different. Its appetizer is antipasto, and rather than a salad course, it has a pasta course. JULIA adapts the normal (default) meal structure it assumed in the beginning to conform to the structure of an Italian meal.

JULIA now concentrates on coming up with its main course (the part of the meal that provides the most constraint for the remainder of the meal). JULIA is reminded of an Italian meal it once planned. In that meal, lasagne, garlic bread, and red wine were served. It proposes this to its client, who accepts it.

Lasagne, however, violates an important meal constraint: main ingredients of dishes are not allowed to be repeated across courses. JULIA uses an adaptation transformation called SHARE-FUNCTION to combine the pasta course and the main course.

Now the client adds a new constraint: *The meal should be vegetarian.* This introduces a contradiction that is resolved by adapting lasagne to vegetarian lasagne using the SPECIALIZE transformation. JULIA proceeds to propose vegetarian antipasto, spumoni-messina and coffee.

In the next example, adaptation is used in a different way. JULIA solves an *over-constrained* problem by adapting the structure of the meal to allow conflicting constraints to be solved independently. It recognizes that the problem is over-constrained when it reaches an *impasse* in its reasoning.

Problem 2

Plan a meal that is inexpensive, easy to prepare, and uses eggplant. Furthermore, it should satisfy Guest1, a vegetarian, and Guest2, a 'meat and potatoes' person who requires meat in his meal. In general, JULIA attempts to find single solutions to each of its subgoals. In this situation, however, it fails to find a single dish that will satisfy both guests. JULIA examines the constraints that are responsible for this impasse. One adaptation transformation it has available, SPLIT-FUNCTION, specifies that if conflicting constraints derive from different sources, the conflict can be alleviated by increasing choice, i.e., using two items to achieve the goal instead of just one. Since the conflicting constraints arise from two different guests, JULIA suggests baba-ghanouj for guest-1 and skewered-lamb and eggplant for guest-2. It supplements this with hummus, pita-bread, retsina, greek-salad, and baklava.

In the next example, adaptation is used a third way. Here, the problem solver reaches another impasse. While above the impasse was resolved by adapting the structure of the meal, here it is resolved by adapting one of the conflicting constraints.

Problem 3

Plan a brunch for a group of 5 people. Some sort of eggs should be the main dish. One of the guests is on a low cholesterol diet. In attempting to solve this problem, JULIA reaches an impasse. It cannot serve a meal with eggs as the main dish that is also low in cholesterol. It resolves this problem by relaxing the constraint that eggs should be a main ingredient of the main dish. It allows eggs to be a secondary ingredient. It then chooses high-protein pancakes as its main dish. Eggs is only a secondary ingredient and very little egg is used.

The last example shows JULIA anticipating a failure and taking steps to avoid it. When the case-based reasoner uncovers a potential failure, JULIA adds the necessary constraints to its problem description that

allow it to avoid the problem. It then continues to solve the problem by remembering another meal whose menu is appropriate to the client. No adaptation is done.

Problem 4

Plan a Mexican meal for my research group that includes tomatoes. JULIA is reminded of a similar meal that failed when someone wouldn't eat spicy chili, and inquires whether this will be a problem. When the client says yes, JULIA adds a constraint to rule out spicy dishes. It remembers another meal, and based on its menu, suggests guacamole, tacos, coffee and flan for dessert.

Roles of Adaptation

The examples above suggest some of the roles that adaptation can play in design:

1. Adaptation facilitates making decisions in under-constrained problems by allowing a problem solver to re-use an 'almost right' plan rather than re-solving from scratch.
2. Adaptation resolves over-constrained problems by serving as a alternative to retraction.
3. Adaptation relaxes preference constraints by minimally weakening them.
4. Adaptation extends the vocabulary of a problem solver by designing new components as variations of known components.

Adaptation and Under-constraint

An under-constrained problem is one for which there may be many possible solutions, but the problem constraints do not help to deduce or construct a solution. Informal tasks such as meal planning tend to be under-constrained and to have large search spaces. Because these search spaces may be ill-defined, it is important for a problem solver to avoid trying to exhaustively search them.

One strategy that can be used is to retrieve previous solutions that are nearly adequate and *adapt* them to fit the current problem. This is called *case-based reasoning.* Case-based reasoning zeros in on a part of the search space that has proven relevant in the past, and in effect, searches only in the neighborhood local to the solution provided by the remembered case. JULIA uses case-based reasoning to propose plausible solutions, and in so doing, trades off guaranteed correctness for search efficiency. In this light, adaptation can be viewed as switching to a smaller, more tractable search space.

Adaptation is a kind of heuristic search in which transformations are applied to a source concept in order to repair constraint violations. The transformations are indexed by constraint violation and type of source concept, and are ranked by their expected cost of application. In JULIA, adaptation transformations are divided into two basic types: those that retain the structure and substitute the components of a concept, and those that retain the components of a concept but alter its structure. Examples of adaptation transformations are:

- SPECIALIZE substitutes a component with a more specific variant that satisfies constraints.
- GENERALIZE substitutes a component with a more general variant that satisfies constraints.
- SUBSTITUTE-BY-FUNCTION substitutes a component with one that is functionally-identical.
- SUBSTITUTE-SIBLING substitutes a component with a taxonomic sibling.
- SHARE-FUNCTION re-structures a concept such that one component serves two functions.
- SPLIT-FUNCTION re-structures a concept such that two components serve a single function in tandem.

The first four are applicable when some feature of a selected design component violates a design constraint. SHARE-FUNCTION is applicable when two components with the same function have been inserted into the design. SPLIT-FUNCTION is chosen when there is an impasse caused by a failure to be able to solve a conjunctive set of constraints. These transformations are described in more detail in [Hinrichs, 1989, Hinrichs, 1991].

Adaptation and Over-constraint

An over-constrained problem is one that has no known solution. There are two types of overconstraint: A *contradiction* denotes a situation in which the problem solver is in an inconsistent state. Problem 1 illustrates several of these. An *impasse* is a situation in which the solution is incomplete and the problem solver can make no further progress. Problems 2 and 3 illustrate impasses. Two classical techniques for dealing with overconstraint are (a) to switch to a different context and (b) to backtrack to a previous decision and retract all of its consequences.

Our approach shows that a contradictory decision need not always be retracted completely to solve a problem. Instead, the decision can often be *adapted* by finding a similar value that will fulfill all of the constraints while preserving the consequences of the previous decision. For example, in Problem 1, the decision to serve lasagne is adapted by specializing it to vegetarian lasagne. This makes it unnecessary to retract other decisions that depend on lasagne, such as garlic bread and red wine.

Another way to look at this role of adaptation is that while dependency-directed backtracking is intended to retract just those *decisions* of a problem that are relevant to a contradiction, the adaptation approach is designed to modify only those *features* of a decision that are contradictory. This can involve substituting one known concept for another, or it can involve creating an entirely new concept on the fly.

There are several reasons why adaptation is an appropriate strategy for dealing with both overconstrained and underconstrained problems:

1. Satisficing solutions are often 'nearby' in the search space. If a component of a solution 'doesn't quite' fit, it is likely that one of its neighbors or relatives will.
2. Replacing one decision with a similar one often leaves consequences of the original decision intact.
3. Minimally altering the solution makes it easier for a human client or user to keep track of what is going on.

Adapting Constraints

When it is not possible to adapt a value or the structure of a problem to satisfy constraints (as in problem 3), it is sometimes possible to relax the constraints on the decision. Typically, constraints are relaxed by ordering them in terms of their importance or utility, and then simply retracting the least important constraint.

Our approach is to *adapt* an offending constraint in order to weaken it just enough so that the problem can be solved. For example, consider the following constraint that the main ingredients of a dish must contain cheese:

```
(contains (?dish main-ingredients) cheese
          relaxable)
```

In this constraint, the first argument defines its scope, or *domain*, and the second argument, **cheese**, defines its *range*. Constraints such as this can be adapted by enlarging or contracting their domain and range.

The constraint adaptation routine resolves an overconstrained decision by first examining the values that have been considered and ruled out, and then sorting them by the number of relaxable constraints they violate. It then attempts to establish the ruled-out value by relaxing each of its violated constraints. An individual constraint is relaxed by applying one of four transformations:

1. ENLARGE-DOMAIN substitutes the scope of the constraint with one that is higher on the partonomic hierarchy.
2. REDUCE-DOMAIN substitutes the scope of the constraint with one that is lower on the partonomic hierarchy.
3. GENERALIZE-RANGE replaces the range of a constraint with a generalization of the range (for nominal constraints) or increases the numeric range (for ordinal constraints).
4. DIMINISH-RANGE specializes the range (for nominal constraints) or contracts the numeric range (for ordinal constraints).

Each constraint type indicates in which direction it must be adapted in order to weaken it. For example, the constraint described above could be adapted by enlarging its domain:

```
(contains (?dish ingredients) cheese
          relaxable),
```

or by generalizing its range:

```
(contains (?dish main-ingredients)
          dairy-product relaxable).
```

In this way, values that almost satisfy constraints can be re-considered if no better solution is discovered. This is a less drastic approach than retracting constraints altogether, because it provides a means of partially satisfying a constraint.

Extending Vocabulary

Adaptation in JULIA does more than simply improve efficiency; it also permits some problems to be solved that would otherwise be impossible. It does this by opening up new search spaces in order to construct concepts that can serve as components of the overall solution. For example, consider the situation in Problem 1 again, in which a decision to serve lasagne contradicts a constraint that the meal be vegetarian. The given search space for this problem is the set of dishes that JULIA knows about. If JULIA didn't know of a vegetarian version of lasagne, then exhaustively searching for a dish wouldn't help.

The adaptation process can solve this problem by reasoning about which feature of lasagne violates the constraint and recursively adapting that feature. In this example, lasagne could be made vegetarian in two ways: either by substituting the meat with vegetarian burger (a functional substitution), or by applying the DELETE transformation to the set of secondary ingredients and eliminating the meat. When a feature is substituted, a new concept is created as a variant of the original. In this way, the problem-solving vocabulary is extended as a by-product of adaptation.

A critical step in this process is determining the legality and plausibility of proposed adaptations. For example, it would make no sense to delete noodles from lasagne, or to make a low calorie version of a dish just by changing the value of its **calories** attribute. JULIA restricts adaptations in two main ways. First, its representation of objects distinguishes between secondary features that are easy to change and primary features that are critical or definitional in some way. For example, secondary-ingredients can be deleted, while main-ingredients, such as the noodles in lasagne, can only be substituted, not deleted. This is one way of representing a partial domain theory. In other words, some kind of lasagne noodle is necessary but not sufficient for a dish to be lasagne.

The second way adaptations are restricted is by using heuristics to infer which features correspond to independent variables and what their ranges of variability are. Instead of directly altering dependent features such as the calories of a dish, it analyzes constraints internal to the concept and regresses back to independent variables (e.g., the ingredients of the dish) and

adapts them. Reasoning of this sort is essential if a problem solver is to create new concepts without the benefit of a strong domain theory.

It is this ability to construct new concepts that allows JULIA to overcome the rigid classifications of selection, configuration and parametric design. If the vocabulary were constant, then JULIA could be viewed as selecting dishes from a known set. However, the ability to adapt components and their structural configuration enables JULIA to do constructive design, but only in the highly circumscribed space of adaptation transformations.

Evaluation

One claim of this research is that adaptation can reduce the amount of work needed to solve problems. To establish this, we measured the number of constraints that JULIA checked as it solved three of the problems described above. We then ran each problem two more times, once with the adaptation capability turned off for contradiction resolution (*Modification Only*), and once with no adaptation at all. The results of this study are shown in table 1:

Problem	*With Adaptation*	*Modification Only*	*No Adaptation*
1	405	439	1138
2	821	–	–
4	1095	1095	600

Table 1: Number of Constraints Checked

Although these numbers are not especially informative by themselves, the intent of this experiment is to study the relative costs and benefits of adaptation. To this end, the example problems presented were chosen to illustrate three different effects. For problem 1, the problem-solving performance degrades monotonically as adaptation is turned off. This is because the contradiction introduced into the problem requires that lasagne be retracted rather than adapted. When the program cannot recall another case containing an acceptable solution, it is reduced to linearly searching all dishes it knows about.

Problem 2 cannot be solved at all without adaptation because there is no single dish in JULIA's knowledge base that satisfies the constraints. Problems that require structural modification cannot be solved by simply searching through categories.

While the results from problems 1 and 2 were as expected, problem 4 was a surprise. Since its solution did not rely on adaptation, we expected no effect on efficiency. As it turns out, however, there can be a hidden cost of adaptation. The process of *attempting* to adapt a value entails looking for substitutions and checking constraints on those values. If no satisfactory solution is found in this way, JULIA proceeds on to the next 'best' reminding and checks components from that case. What happened in problem 4 was that JULIA did not pick the most appropriate case right off the bat, and therefore spent some time chasing down blind alleys attempting to adapt dishes that could never suffice.

Problem 4 may simply be a pathological case. For the vast majority of cases we have run, adaptation appears to be beneficial. However, adaptation clearly involves a tradeoff between flexibility and efficiency. It opens up search spaces that can lead to otherwise inaccessible solutions, but it may also explore deadends. Because of this, models of problem solving that rely on relatively expensive deep reasoning for adaptation should take into account the *accuracy* of the indexing mechanism and the *density* of the solution space. For problem solvers that index cases accurately and for problems that have sparse solution spaces, it is worthwhile to expend a lot of effort in trying to adapt previous cases. For problem solvers that index cases less accurately and for problems with dense solution spaces, it is often better to expend less effort adapting cases since a better solution may be found in the next case considered.

In principle, the cost of a complex adaptation process could be amortized if the problem solver were to store and re-use its own reasoning steps. JULIA is limited in this regard because it only applies case-based reasoning to propose design solutions. However, if the cost of adaptation were more significant, then recursively applying case-based reasoning would be worth investigating.

Related Work

JULIA draws heavily on ideas from other work in case-based reasoning, such as CHEF [Hammond, 1989]. From the perspective of adaptation, CHEF implements modification of cases independently from repair of failures. In JULIA, they are both applications of the same process of adaptation, and rely on the same set of transformations. Also, CHEF does not adapt previous decisions as part of its control strategy.

Using adaptation to augment dependency-directed backtracking is not new to JULIA, and has been addressed in PRIDE [Mittal & Araya, 1986], where it is referred to as *modification advice*, and DONTE [Tong, 1988], where it is called *patching*. We extend the technique to cover the adaptation of structure (ie, configuration) and constraints. Also, both PRIDE and DONTE solve problems that have well-defined search spaces. Modification in PRIDE cannot design new components. In JULIA, adaptation can construct new components, and thus partially determines the class of problems that can be solved.

Like VEXED [Steinberg, 1987], JULIA is a design advisor. As advisory systems, the two systems differ in how they define the division of labor between the user and the program. VEXED assumes responsibility

for evaluating completeness and correctness of designs and delegates control decisions to the user. JULIA, on the other hand, makes its own control decisions, but leaves the user as the final arbiter of design adequacy.

Discussion and Conclusions

We have presented a model of design that employs adaptation in multiple roles. This model provides the capability to solve design problems that require the integration of processes for selection, configuration, parametric manipulation, and construction from scratch. By adapting similar known solutions, we trade a poorly defined design search space (meals in the case of JULIA) for the better-defined space of adaptation transformations. This technique is appropriate when solutions to similar problems are known, and when the criteria for judging similarity are well understood.

Another feature of our model is a control strategy that exploits adaptation as an alternative to retraction. While this idea has been explored in some previous work, we extend the idea in two ways: First, adaptation constructs new concepts as a side-effect, so that it permits the solution of problems that would otherwise not be possible. Second, the ability to adapt *constraints* unifies the ideas of backtracking and constraint relaxation. This technique is appropriate when decisions have many consequences and the solution space is dense.

References

D.C. Brown and B. Chandrasekaran. Expert systems for a class of mechanical design activity. In J.S. Gero, editor, *Knowledge Engineering in Computer-Aided Design*. Elsevier Science Publishers B.V., North Holland, 1985.

J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12(3), 1979.

V. Goel and P. Pirolli. Design within information processing theory: The design problem space. *AI Magazine*, Vol. 10, No. 1, 1989.

K.J. Hammond. *Case-based Planning: Viewing Planning as a Memory Task*. Academic Press, New York, 1989.

T.R. Hinrichs. Towards an architecture for open world problem solving. In J.L. Kolodner, editor, *Proceedings of the 1988 DARPA Workshop on Case-Based Reasoning*, pages 182–189, 1988.

T.R. Hinrichs. Strategies for adaptation and recovery in a design problem solver. In K. Hammond, editor, *Proceedings of the 1989 DARPA Workshop on Case-Based Reasoning*, pages 115–118, 1989.

T.R. Hinrichs *Problem Solving in Open Worlds: A Case Study in Design*. PhD thesis, Georgia Institute of Technology, 1991. forthcoming.

J.L. Kolodner. *Retrieval and Organizational Strategies in Conceptual Memory: A Computer Model*. Lawrence Erlbaum and Associates, Hillsdale, NJ, 1984.

J.L. Kolodner, R.L. Simpson, and K. Sycara. A process model of case-based reasoning in problem solving. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 284–290, Los Angeles, 1985.

S. Mittal and A. Araya. A knowledge based framework for design. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 856–865, Philadelphia, PA, August 1986.

J. McDermott. R1: An expert in the computer systems domain. In *Proceedings of the First National Conference on Artificial Intelligence*, pages 269–271, August 1980.

R.C. Schank. *Dynamic Memory: A theory of reminding and learning in computers and people*. Cambridge University Press, London, 1982.

R.M. Stallman and G.J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer aided circuit analysis. *Artificial Intelligence*, 9:135–196, 1977.

M.J. Stefik. Planning with constraints. *Artificial Intelligence*, 16(2):111–140, 1981.

M.J. Stefik. Planning and meta-planning. *Artificial Intelligence*, 16(2):141–170, 1981.

L.I. Steinberg. Design as refinement plus constraint propagation: The VEXED experience. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 830–835, 1987.

C.H. Tong. *Knowledge-Based Circuit Design*. PhD thesis, Stanford University, 1988. (Rutgers University Report No. LCSR-TR-108).

A.C. Ward and W.P. Seering. Quantitative inference in a mechanical design "compiler". memo AIM-1062, MIT AI Lab, January 1989.