Some Criteria for Evaluating Designs*

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

Abstract

Most non-trivial design tasks are *under-specified*, which makes evaluating designs subjective and problematic. In this paper, we address the evaluation criteria that are left implicit in problem specifications. We propose that these criteria evaluate designs in terms of specific types of consistency and completeness. In particular, we divide consistency into constraint, representational, and goal consistency, and we decompose completeness into the specificity, depth, and breadth of a solution. These distinctions are useful because they organize criteria for evaluating designs. This model of evaluation is largely implemented in a program called JULIA that plans the presentation and menu of meals to satisfy multiple, interacting constraints.

1 Introduction

Real world design problems are almost universally under-specified. Consequently, designers must augment formal specifications with their own expectations and criteria for evaluating designs. What are these criteria and where do they come from? One possibility is that they are simple syntactic checks on the consistency and completeness of a design. The problem with this approach is that it is insufficiently flexible, since the appropriate level of completeness may vary for the same problem in different contexts. For example, generating a shopping list for a meal requires a more specific menu than deciding whether that same meal will appeal to guests. In short, to evaluate a design, one must know the requirements of the design itself, as well as those of the artifact.

Previous research in design has emphasized the search strategies that designers use rather than their evaluation criteria and stopping rules. This may be because the tool of

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

choice in studying design is *protocol analysis*, which facilitates analyzing the time-course of problem solving, but provides little help in eliciting underlying decision criteria. Protocol studies have been reported in a variety of design domains, including architecture [Eastman 1969, Goel & Pirolli 1989], mechanical design [Ullman *et al.* 1986], software design [Malhotra *et al.*, 1980], algorithm design [Kant & Newell 1982], and meal planning [Byrne 1977]. In each of these studies, problem specifications were informal and incomplete, yet designers were able to evaluate their solutions. The unarticulated evaluation criteria which they used are the subject of this research.

In this paper, we define different types of consistency and completeness. These distinctions are useful because they serve to organize criteria for evaluating designs. Criteria for *consistency* enable the evaluation of partial designs and the coordination of multiple representations. Criteria for *completeness* provide contextual clues for determining when to stop refining a design, thereby permitting the design process to be iterative.

In the next section, we discuss criteria for evaluating the consistency of designs. In section 3, we elaborate the notion of completeness and present criteria for evaluating the specificity and scope of designs. These evaluation are being implemented in a computer program called JULIA [Hinrichs 1988, Hinrichs 1989], which plans meals to satisfy multiple, interacting constraints.

2 Evaluating Consistency

Consistency maintenance is the means by which a designer ensures that different parts of a solution agree with each other. Two factors conspire to make consistency maintenance difficult. First is the need to evaluate *partial solutions*. When a design is only partly refined, missing information may make constraints appear violated when they are not. For example, a global constraint that a meal contain eggplant may appear to be violated until a main dish is chosen. Alternatively, missing information may camouflage an actual inconsistency. For example, caviar may not appear inconsistent with a tight budget if the rest of the menu is not specified yet. To address this issue, we present an approach to constraint management that permits constraints to be incrementally formulated, evaluated under different policies, and strategically relaxed.

The second factor is called the *Logical Omniscience Fallacy* [Halpern 1986]. This is the mistaken assumption that it is reasonable for an agent to know all the implications of his beliefs. This assumption often shows up in Artificial Intelligence models of problem solving in which consistency is maintained via a single, uniform mechanism such as a Reason Maintenance System [Doyle 1979]. The problem with this is that it is both overly powerful and insufficiently flexible. In particular, it assumes that there is only a single type of consistency. However, design involves several different types of relationships that have independent and distinct criteria for evaluating consistency. To accommodate this, we partition design relationships into 1) *constrained* relationships between parts of a solution,

2) relationships between *representations*, and 3) relationships between *goals*. Consistency across each type of relationship is maintained by independent, dedicated mechanisms.

2.1 Constraint consistency

Most of the overt criteria for evaluating a design involve the satisfaction of constraints. These criteria determine *which* constraints are to be satisfied, to *what degree* they are satisfied, and what must be *changed* to maintain consistency. Constraint consistency is implemented in JULIA by a constraint poster that is built on top of a Reason Maintenance System.

Where do constraints come from?

To evaluate solutions to under-specified problems, a designer must implicitly fill in constraints that are missing. The process that generates these constraints is called *formulation* [Stefik 1981]. Constraint formulation is an incremental process in which the problem specification is refined as commitments are made in the solution. The constraint formulation process generates constraints from three sources:

- *Problem statement.* Many constraints are explicit in the original problem specification. For example, a typical problem that JULIA might solve is: "Plan a vegetarian Greek birthday dinner that contains eggplant."
- *General plans.* Some constraints are imported from general schematic knowledge of partial solutions. For instance, individual meals in JULIA inherit the constraint that side dishes should be compatible with the main dish.
- *Other constraints.* Some constraints are inferred from other constraints. For example, JULIA knows about the food preferences of individual people. These preference constraints are propagated to the meal being planned.

Strategies for evaluating constraints

A designer must be able to evaluate partial solutions, consequently it should be possible to check constraints when some information is missing. To do this, our model of design provides explicit strategies for evaluating constraints when some information is missing, or when approximate solutions are acceptable. These strategies include:

- *Optimistic evaluation.* Design constraints can be evaluated optimistically by accepting solutions for which constraints are not explicitly violated. JULIA adopts this policy when generating potential solutions.

- *Pessimistic evaluation.* Alternatively, potential solutions can be rejected whenever information with which to evaluate a constraint is missing. JULIA uses pessimistic evaluation when comparing two potential solutions, in order to determine which alternative satisfies constraints 'better'.
- *Approximate evaluation.* When constraints can be partially satisfied, a simple predicate is insufficient for evaluation. Instead, evaluation must indicate the degree to which constraints are satisfied. JULIA considers partial satisfaction when relaxing constraints on an over-constrained problem.

Strategies for resolving inconsistency

When constraints contradict each other, the inconsistency must be resolved. The ways in which inconsistencies are resolved can serve as evaluation criteria for the overall quality of a solution. For example, the fewer the constraints relaxed, the better the design. Three general strategies for resolving inconsistencies are:

- *Change the solution.* Inconsistent solutions can be corrected by either retracting a previous decision or by applying some new transformation to adapt it.
- *Change the problem.* Sometimes, it is not possible to solve a given problem. In this case, one strategy is to simplify the problem until it can be solved. This can be done by relaxing preference constraints.
- *Ignore it.* Some inconsistencies can be recognized and yet ignored, such as preference constraints that cannot be satisfied.

2.2 Representational consistency

The second type of consistency we distinguish is consistency across different *representations* of a design. Any problem solver that uses multiple representations must ensure that the representations agree with each other. JULIA maintains two types of representation, an *assertional representation* in the form of reason-maintenance nodes, and a *structural representation* in the form of frames. Representational consistency means that information represented in one form corresponds with, but does not contradict, information represented in another.

If the correspondence between representations becomes ambiguous, some mapping must be chosen. For example, if JULIA is interacting with a client to plan a meal, the client may suggest a particular dish. This suggestion is represented as an assertion that is ambiguous with respect to its role in the structure of the meal (*e.g.* Should it be an appetizer or a side dish?) To maintain representational consistency, JULIA must make an educated guess about the role that the dish is to play in the meal.

If representations conflict, it is necessary to either retract an assertion, or relax a representational assumption. For example, when JULIA asks a client to choose between two dishes, the client may answer 'both'. In this case, the structural representation assumes that there will be a unique dish in the role. JULIA must therefore change the structural representation to accommodate multiple dishes. Representational consistency is maintained by a module called a *structure maintenance system*, or SMS.

2.3 Goal consistency

The third type of consistency we distinguish is consistency across the *goals* of the designer. Unlike constraint and representational consistency, conflicting goals are an inescapable part of design. Designers must be able to entertain conflicting goals, because designs are often specified in contradictory terms. For example, the specification for an automobile may require that it be inexpensive, yet safe.

Nevertheless, some goal relationships *are* maintained. First, the designer's goals should track the structure of the evolving solution. To do this, when new structures are added, the problem solver posts new goals to refine the structure, and when structures are deleted, it abandons the corresponding goals. Second, when a goal is abandoned, its subgoals are also abandoned. In JULIA, these inferences are the responsibility of a *goal scheduler* that maintains an agenda and a network of goals.

3 Evaluating Completeness

The second main criterion for evaluating designs is *completeness*. How complete must a design be in order to be acceptable? Previous models of design have referred to *stopping rules* that terminate design refinement [Byrne 1977, Goel & Pirolli 1989], but they have been vague about just what these rules might be. In order to clarify the ways in which stopping rules may differ, we describe the completeness of designs in terms of their *specificity* and *scope*.

3.1 Criteria for specificity

The *specificity* of a design indicates the level of abstraction of the components of the design. For example, sometimes it is sufficient to indicate a hot soup, while at other times, it is necessary to specify 'Campbells tomato soup.' Criteria for determining the appropriate level of specificity include:

- *Convention.* Solutions should be of approximately the same level of specificity as previous similar designs. This criterion can be measured by reference to some level in

a taxonomy, such as a top level, a terminal level, or a designated basic level.

- *Determining constraint satisfaction.* Solutions should be specific enough that their constraints are explicitly satisfied. This criterion can be measured by the information content of the representation. For example, a requirement on the specificity of a dish might say that it must specify its ingredients. Under this rule, 'soup' would be inadequate, but 'tomato-soup' would suffice.
- *Cognitive load.* Solutions should be general enough to minimize the number of alternatives to be maintained in working memory, or to be communicated to a client. This can be measured by a bound on the size of the contrast set. For example, a meal might be described at the level of 'clear soup' *vs* 'cream soup' simply because the number of alternatives is small.

3.2 Scope

In addition to the specificity of solution components, a design can be evaluated in terms of the adequacy of its structure, or its *scope*. The scope of a design defines the set of variables the designer must solve. Some problems, such as *parametric design* problems, have fixed scope. Many design problems, however, do not have a fixed solution structure and require the designer to evaluate the scope of proposed solutions. The scope of a design can be further decomposed into its *depth* and *breadth*, which more precisely characterize its structure.

Some evidence for the distinction between specificity and scope can be observed in published design protocols. For instance, in [Byrne 1977] and [Kant & Newell 1982], the experimenters had to occasionally prompt their subjects to be more specific. However, there is no evidence of subjects being confused about the *scope* of their task. This suggests that the evaluation criteria for design scope and specificity are different and may be a function of different types of knowledge.

Criteria for depth

The *depth* of a design describes how finely it is decomposed. For a meal-planner, a shallow design might designate what courses are in a meal, or what dishes will be served in a course. A very deep design might decompose into discrete actions, such as going up to a buffet table, serving, eating, and so forth. Some criteria for determining the appropriate level of granularity of a design or plan are:

- *Simulation.* If there are constraints on the behavior of an artifact, then evaluation may involve simulating its operation. Simulation may require the design to be decomposed to a certain uniform depth.

- *Explanation.* In order to answer questions about a design, it may be necessary to decompose it to finer granularity. For example, to explain why a previous design failed, it may be necessary to simulate it.
- *Convention.* In the absence of other criteria, decompose a solution to some standardized level, based on similar designs or on generic design plans.

Criteria for breadth

The *breadth* of a design characterizes the range of considerations subsumed under the design problem. For example, the breadth of a design for an artifact may range from a simple structural description, all the way to an account of how it will be manufactured, the tools required to build it, and the procedures needed to operate and repair it. These peripheral aspects of a design provide constraints on the more central structure of the design. Criteria for determining the required breadth of a design include:

- *Standardized checklist.* Design tasks that are routine or well-understood often have explicit criteria for completeness in the form of standardized checklists. These serve as external aids to memory.
- *Avoiding failure.* One criterion for the breadth of a design is that it be broad enough to anticipate possible failure modes. To determine this, feedback from previous failed designs can be used to indicate peripheral features that led to goal failures. For example, a design that was impossible to maintain or test should clue the designer to consider these features in the future.
- *Determined by oracle.* An external agent, such as the client, may ask specific questions about peripheral aspects of a design, which would drive the designer to refine these features.

4 Discussion

We have presented a set of criteria for assessing the consistency and completeness of designs. An implication of these criteria is that to evaluate a design, one must know the requirements for the design itself, as well as those of the artifact. One way this idea can be operationalized is to associate evaluation criteria with explicit design goals, so that the requirements for the design can be inferred from the goal network. JULIA has a vocabulary of design goals that would facilitate this, though the completeness criteria have not yet been implemented.

This paper describes work in progress. JULIA exists and designs meals, and the partitioned approach to consistency maintenance is implemented and works. The criteria

for evaluating completeness are only partly integrated into JULIA and are our current focus of research. Some of the problems that remain to be addressed in this work are:

- *Partially satisfiable goals* complicate stopping rules. For example, if a goal of a meal is to minimize calories, then when is a meal sufficiently low-calorie? A design may be adequately consistent and complete, yet still be insufficient.
- *Representing completeness criteria.* Should specificity be represented as a level in a taxonomy, or as a constraint on information content? How should the required depth of a solution be represented? Design evaluation would appear to require a *meta-level* representation.

References

R. Byrne. Planning meals: Problem solving on a real data-base. *Cognition*, 5:287–332, 1977.

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

C.M. Eastman. Cognitive processes and ill-defined problems: A case study from design. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 669–690, 1969.

V. Goel and P. Pirolli. Motivating the notion of generic design within information-processing theory: The design problem space. *AI Magazine*, 10(1), Spring 1989.

J.Y. Halpern. Reasoning about knowledge: An overview. In *Theoretical Aspects of Reasoning About Knowledge*, pages 1–17, 1986.

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.

E. Kant and A. Newell. Problem solving techniques for the design of algorithms. Technical Report CMU-CS-82-145, Carnegie Mellon University, 1982.

A. Malhotra, J.C. Thomas, J.M. Carroll, and L.A. Miller. Cognitive processes in design. *International Journal of Man-Machine Studies*, 12(2):119–140, 1980.

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

D.G. Ullman, L.A. Stauffer, and T.G. Detterich. Preliminary results of an experimental study of the mechanical design process. Technical Report 86-30-9, Computer Science Department, Oregon State University, 1986.