Critics for Knowledge-Based Design Systems

H. Liu,* C.D. Rowles *and* W.X. Wen
{c.rowles, w.wen}@trl.oz.au
Artificial Intelligence Systems
Telecom Research Laboratories
Clayton, Vic. 3168, Australia

Abstract

Expert critics have been built to critique human performance in various areas such as engineering design, decision making, etc. We suggest that critics can also be useful in building and use of knowledge-based design systems (KBDSs).

Knowledge engineers elicit *knowledge* from domain experts and build a knowledge-based design *system*. The system generates *designs*. The amount of knowledge the system possesses and the way it applies the knowledge directly influence the performance of its designs. Therefore, critics are proposed to assist (1) acquiring *sufficient* knowledge for constructing a desirable system, and (2) applying *proper* knowledge to generating designs. Methodologies of equipping a KBDS with critics are developed. Our practice in building and using a KBDS shows the applicability and capability of these critics.

Key Words: Knowledge-based Systems, Design, Critiquing, Knowledge Acquisition

1 Introduction

Engineering design is a domain where a knowledge-based approach has often been adopted due to the very large number of factors that must be considered and the difficulty in accurately characterizing them [17]. A brute-force approach is out of the question for *configuration design tasks* due to its computational complexity. Experts design by following general rules, applying their knowledge, and using contextual sense and empirical insights. A knowledge-based design approach attempts to mechanically generate designs with reference to the behavior of experts. This work is about how to build a better knowledge-based design system to perform configuration design where the choices in a design can be enumerated exhaustively. Given a domain and problems to be solved, knowledge engineers first

*This author is currently with Department of Information Systems & Computer Science, National University of Singapore, 0511. liuh@iscs.nus.sg

acquire expert knowledge, *codify* the knowledge in a machine usable form (rule, predicates, etc.), and *build* a knowledge-based design system. Ideally, when the system is in operation, it can *apply* proper knowledge to problems accordingly to produce "good" designs that meet the requirements and criteria. Otherwise, more knowledge needs to be acquired in order for the system to generate good designs. A cycle is formed by these four processes (*acquiring, codifying, building*, and *applying*) that heavily depend upon domain experts (how they choose and summarize the knowledge) and knowledge engineers (how they elicit, codify and use the knowledge). In general, like all models, knowledge bases are selective, based on assumptions, and prone to failure [2]. Biases, irrelevant or incomplete knowledge, misconceptions, and wrong assumptions are likely to be introduced into the system through these four processes and therefore affect the quality of the system's designs. Some effective means must be employed to reduce the likelihood of these factors being incorporated in a system.

A *critic* is a system in which a knowledge-based design system (KBDS) is a component. A critic takes, as its inputs, a problem description and a proposed solution to the problem and gives, as its output, a critique, i.e., what is wrong and where it is wrong. It detects errors or possible improvements, and provides the user with directions for correction and improvement. A set of critics is proposed for knowledge-based engineering design. It accounts for aspects to be covered by critics, their functions, and their usage in building and applying knowledge-based systems. The critics are aimed at assisting knowledge acquisition and improving system's performance.

In the following, the need for critics is elucidated for a knowledge-based approach; a set of critics is then suggested; and each critic is described in turn by giving algorithms that implement the critics. In Section 3, an example application is defined, and a KBDS, called *DS*, is described. Section 4 illustrates how the set of critics is applied to improve DS's performance. In Section 5, some related work is discussed to place this work in literature. We conclude with a summary of the proposed critics, and with a discussion of future directions for research and development on this work.

2 Critics of KBDS

In this section, it is first shown, by identifying four problems, that critics are necessary in building and using a KBDS. Second, a set of critics is proposed to attack these problems by critiquing designs of a KBDS and by using domain knowledge. Then the critics are presented with the algorithms that implement them.

2.1 The Necessity of Critics

The necessity of critics can best be seen in the four problems identified in KBDSs. The first two problems pertain to eliciting and applying knowledge, and the last two concern the designs a system produces.

(1) Acquiring adequate knowledge

Acquiring knowledge for a KBDS is a subjective process: knowledge engineers and domain experts could well be biased in their viewpoints or from their experiences. Too often, useful knowledge is omitted – *the adequacy problem*. A critical question is how they can know if the knowledge obtained is sufficient for a given domain. Perfect knowledge is impossible to obtain, but sufficient knowledge is essential for a KBDS to succeed [6, 8]. In a moderately complex domain, it is difficult to find out what's missing, based on a handful of test cases. Increasing the number of cases studied can be time-consuming and is often impractical. Hence, a method is sought to alleviate the adequacy problem without increasing the number of cases to study. Such a method is expected to provide clues and directions to search for more knowledge and to reorganize knowledge.

(2) Choosing proper knowledge

As the knowledge accumulates in a particular design domain, we ultimately encounter the problem of choosing the most suitable knowledge for a particular case – *the application problem* [7]. Not knowing all cases beforehand makes this problem extremely difficult. Heuristics employed by a KBDS may bring about global optimization. Carefully designed heuristic search may generally produce the expected results, but, it may also generate some unexpected (or poor) results in some cases. Therefore, it is sometimes necessary to try some other knowledge or a new combination of heuristics in order to ensure proper knowledge is used in design.

(3) Correctness of designs

Correctness is a minimum requirement for engineering design in that a solution should satisfy what a problem description specifies – *the correctness problem*. Any unsatisfied requirement indicates mistakes in the structure of the system, and/or misconceptions in a knowledge base. Whether a design is correct can be tested directly by checking if all the requirements stated in the input (a problem description) are met by the output (a design).

(4) Consistency of designs

A problem description specifies particular requirements to be satisfied, whereas a domain theory contains general knowledge about the domain. As a problem description can be employed to check correctness of a design, a domain theory can be used to check consistency of a design. That is, designs should be consistent according to a domain theory – *the consistency problem*. Design consistency means two things: (1) consistency among the components of a design; and (2) consistency between the components of a design and the domain theory. Lack of consistency indicates logical errors in the system structure, misconceptions, or possibly, wrong assumptions in a knowledge base.

2.2 Critics

The four problems described above are obstacles in building a successful KBDS, and must be treated systematically. One way of doing so is to critique a design, because (1) designs are in general a good indicator of how well a design system is built and knowledge is used; and (2) errors found in designs usually reflect problems in the process of either acquiring, organizing, or applying knowledge. In addition, domain theories and problem descriptions can be used in critiquing. Therefore, the use of critics is suggested as follows:

i. these four problems are the minimum set that should be considered;
ii. there is information available for critiquing, such as problem descriptions, domain theories and produced designs;
iii. the use of critics should be either independent (of a knowledge base) or deductive (from a part of the knowledge base);
iv. formal and systematic use of critics should be pursued; and
v. a KBDS is a part of critics.

Four critics are proposed here to critique solutions generated by a KBDS. The aim of these critics is to assist (1) acquiring *sufficient* knowledge for constructing a desirable system, and (2) applying *proper* knowledge to generating *good* designs.

2.2.1 Expertise completion critic

An expertise completion critic is required to deal with the adequacy problem, and to help a knowledge engineer reorganize and generalize the knowledge base of a KBDS. One way of implementing the critic is to build a strong model of a domain-trained knowledge engineer and to apply it to knowledge acquisition [15]. In general, we should try as strong a model as possible, to pool all available experts together, and to combine their expertise. When a domain is complex, building a strong model is a problem in itself. Even if a strong model is obtainable, the adequacy problem still cannot be solved. When a KBDS produces correct designs consistent with the domain theory, how could one know whether a better design has been missed or not? A system cannot determine, by itself, its own knowledge adequacy. A critic can help, after a strong model has been built, to make the model stronger. When only a few known cases are available, the critic uses an *independent* system to check the knowledge adequacy by ensuring that the KBDS produces at least as good as any designs by the independent means. By independent we mean that two systems are different in their implementation mechanisms, in their search algorithms, and in their knowledge used. For example, a neural network system can be a choice of independent system, but another KBDS is not[1]. The critic discovers missing knowledge by applying both systems to producing designs, and then studying their differences to find out why the designs are different, and to locate what's missing. Of course obtaining an independent system can be difficult. It must (1) adopt a different mechanism and a different search algorithm; (2) apply different knowledge; (3) use the same representation for the input and output as the KBDS; and (4) be feasible to implement and run. Often these requirements can be met by using an algorithmic design system, which is capable of producing good designs, but too computationally intensive for routine use in design.

Algorithm 1 implements our first critic. This critic is used during system construction phase when computing time is relatively affordable, and experts are available for detailed cross examination of a *few known* cases. Suppose we are building a knowledge base, KB for a design system, DS; IS is an independent design system that does not use the knowledge in KB; d_d and d_i are designs generated by DS and IS respectively; and $diff$ stores the differences between d_d and d_i. This critiquing process continues until every individual case, p in P - a set of known cases, is tested and $diff$ is empty or it can be terminated by a knowledge engineer if a certain degree of satisfaction is reached.

```
--- Algorithm 1 ---
For (each p ∈ P) do
  d_i = Apply(IS, p);
  KnowledgeAcquisition(p, KB, d_i);
KnowledgeAcquisition(p, KB, d_i)
  d_d = Apply(DS, p);
  diff = Compare(d_d, d_i);
  If (diff <> NULL)
     ModifyKB(KB, diff);
     KnowledgeAcquisition(p, KB, d_i);
```

Above, Compare(d_d, d_i) gives the differences between d_d and d_i in terms of critical decision points defined by the domain. In ModifyKB$(KB, diff)$, a knowledge engineer first analyzes $diff$, figures out whether KB is capable to diminish $diff$ by rearranging the order of rules or adding new knowledge, and modifies KB accordingly.

2.2.2 Alternative solution critic

There may exist redundant and *ad hoc* knowledge for handling special situations. For a given design problem, it is not always obvious which piece of knowledge is most suitable

[1] Otherwise, why can't we combine the two KBDSs to build a more powerful KBDS in the first place?

to achieve an optimal solution. However, it would become more obvious when a design is generated. Which parts of the design are not good or reasonable can be used as feedback to try out a suitable body of knowledge. The feedback can be obtained automatically or from a design expert, and guide a KBDS to generate alternative designs. By finding the best design in terms of cost or expert judgment, the critic helps identify the most proper knowledge. The critiquing process continues until no better solution is found or until design experts are satisfied. Below two algorithms are given for an alternative solution critic: one is automatic and the other allows experts to be involved in critiquing.

Suppose T is a threshold for the maximum number of designs; c_t is a cost for d_t; C is a set of various costs; d_t is a design at time t; and p and P are defined above. Every design has a $WorstParts$ stack. Initially, it is empty. *DS* generates a design d_t that is evaluated to have a cost c_t and what is in *WorstParts* depends on applications.

```
--- Algorithm 2 ---
For (each p ∈ P) do
  numOfDesign = 0;
  WorstParts = NULL;
  AutoReDesign(p, numOfDesign, WorstParts);
AutoReDesign(p, numOfDesign, WorstParts)
    SetInitial(WorstParts);
    d_t = Apply(DS, p);
    numOfDesign++;
    c_t = Evaluate(d_t);
    WorstParts = Report(DS, d_t);
    If (numOfDesign > T OR c_t ∈ C)
       Stop;
    else
       C = Append(c_t, C);
       AutoReDesign(p, numOfDesign, WorstParts);
```

Here SetInitial($WorstParts$) finds the most possible responsible decision points in d_t, resets the strategy (heuristic) there. At each decision point, a few competing heuristics are available. They are selected in a round robin fashion. Evaluate(d_t) calculates the cost of d_t based on a cost function. Report(DS, d_t) gives a list of worst parts in d_t. An algorithm is given in Appendix A to show how this can be implemented in telephone cable network design. Append(c_t, C) includes c_t into C. Unless *numOfDesign* reaches the threshold T or *DS* does not produce designs with the same costs, the algorithm tries to reset some strategies at the critical decision points (determined by applications) responsible for *WorstParts*. A new design will be generated with different heuristics at the identified decision points. By varying the heuristics various designs can be generated. Thus, the above algorithm carries out automatic redesign.

Obviously, the performance of this critic is dependent upon the utility of the cost function as metric for design quality. If an accurate metric is difficult to generate, it would limit the degree to which designs can be improved. As an alternative, the following algorithm involves expert in determining which parts of a design can be improved. Suppose D is a set of designs; and *suggest* is the input by human experts after examining a design, d_t. Suggestions are solely based on the expert's judgment of a graphically displayed design. SetInitial(*suggest*) resets heuristics at suggested decision points. ExpertExamine(d_t) presents d_t, with all d in D if necessary, graphically to design experts and returns suggestions by experts for possible improvement to d_t.

```
--- Algorithm 3 ---
For (each p ∈ P) do
    suggest = NULL;
    D = NULL;
    InteractiveReDesign(suggest, D);
InteractiveReDesign(suggest, D)
    SetInitial(suggest);
    d_t = Apply(DS, p);
    D = Append(d_t, D);
    suggest = ExpertExamine(d_t, D);
    If (suggest <> NULL)
       InteractiveReDesign(suggest, D);
```

Algorithms 2 & 3 implement the alternative solution critic, which can be used either automatically or manually. Their use in DS is shown in Figure **??**. A user can choose either one, automatic or manual. When the latter is chosen, the user will be asked to input critical decision points s/he thinks are responsible for an unsatisfied design based on a graphical display.

2.2.3 Correctness and consistency checking critics

These two critics are relatively close to each other since they rely on the contextual information of design such as domain theories and problem descriptions. Correctness of a design is defined as satisfaction of the design requirements specified in a problem description. Consistency is defined as correct interrelationships (no conflicts) between the design and the domain theory, and between different parts of a design. Two critics are used during *both* system construction *and* design generation. Only when no incorrectness or inconsistency is found can the two critics stop monitoring. While correctness and consistency of a design must be checked to ensure that it is a valid solution to the design problem, the role of these critics in the phase of system construction is mainly to identify the sources of problems in the knowledge base and help a knowledge engineer correct them.

(Insert Figure 1 here)

The algorithm for correctness and consistency checking is as follows. Suppose we are constructing a system DS, P is a set of design problems, T a domain theory, d a design, Sys a version of DS, $S_{incorrect}$ and $S_{inconsistent}$ sets of reported errors. The algorithm is terminated when $S_{incorrect}$ and $S_{inconsistent}$ are empty, i.e., Sys produces designs for P without incorrectness and inconsistency. The domain theory and input information are used. A nonempty $S_{incorrect}$ or $S_{inconsistent}$ starts refining DS. In Create(T) of Algorithm 4, knowledge engineers generate an initial version of DS using all knowledge available, such as all domain experts, specifications, manuals, etc. in forms of production rules and procedures. In Apply(Sys, p), Sys is employed to construct a design for p. In Refine($Sys, d, T, S_{incorrect}$, and $S_{inconsistent}$), knowledge engineers modify Sys based on those variables and produce a new version of DS. CorrectnessCheck(d, p) examines d against p and reports any errors to $S_{incorrect}$. ConsistencyCheck(d, T) examines d against T and reports any errors to $S_{inconsistent}$. SystemDesign(Sys, p) is defined recursively to refine DS until a version of DS does not have incorrect and inconsistent parts in its design for p.

```
--- Algorithm 4 ---
Sys = Create(T);
For (each p ∈ P) do
    SystemDesign(Sys, p);
SystemDesign(Sys, p)
    d = Apply(Sys, p);
    S_incorrect = CorrectnessCheck(d, p);
    S_inconsistent = ConsistencyCheck(d, T);
    If (S_incorrect <> NULL OR S_inconsistent <> NULL)
       Sys = Refine(Sys, d, T, S_incorrect, S_inconsistent);
       SystemDesign(Sys, p);
```

Re-examining the above four algorithms, we notice that a KBDS is always a component of a critic (e.g., either Apply(DS, p) or Apply(Sys, p) is called), but not vice versa. All the four try to improve a KBDS by examining designs the KBDS produces. Moreover, *Modify*, *Refine* and *ExpertExamine* require the involvement of experts and/or knowledge engineers. Machine learning could be applied to knowledge base modification [3], but it is outside the scope of this paper.

In the following, we will show the use of the critics in the construction and application of a practical KBDS called DS. Critics are used to detect errors in a design, provide directions to enrich a knowledge base, improve designs, and reduce subjectivity in knowledge acquisition. Improved DS can result in better designs.

3 An Example Application of Critics

We will now describe an engineering design application to show how critics are built and used. The application is that of telecommunications network design. As a commonplace design task, this has attracted the development of KBDSs for automatic network design.

3.1 An application: telecommunications network design

The specific area tackled by DS is that of telephone distribution network design. This is that part of the telephone network between telephone exchanges (switches) and customers' premises. For simplicity, the design of this type of network is divided into distribution areas, each served by a large cable from the exchange. The distribution network consists of cables and joints, with cables running in underground pipes, and joints in underground pits. The problem is partly one of connectivity to all customers, but also one of efficiency in that larger cables are used to minimize costs, so the topological distribution problem becomes one of non-linear, discrete optimization where longer cable routes are traded-off against the costs of various cable sizes. Network design must also consider maintenance costs, which leads to minimal use of joints.

Input *(problem description)*: (1) The topological and cadastral information of a district that consists of streets, intersections, and houses; and (2) service requests for each house (i.e., location for provision of telephone network access for a residence and the amount of the provision.)

A sample problem: there are 118 houses in a district as shown in Figure **??**(a) that consists of eight intersections and eleven street segments (polygons), etc. (provided in a cadastre file with the details such as how they are located and related, their coordinates and so on), and each house requires one telephone connection.

Output: A tree structure $\mathrm{T}(j, c)$ where j are joints that connect different sized cables, and c cables that are telephone lines in groups. Joints are nodes, and cables are links in $\mathrm{T}(j, c)$. The root of a tree is a special joint, the pillar, as a major distribution point that is linked to a larger network. Cables connect the joints to the pillar.

Criteria: to achieve the minimum number of joints, minimum number of street crossings, and minimum use of cables in terms of length and size, and minimum cost of $\mathrm{T}(j, c)$ when the first two minima are obtained.

3.2 Domain theory

A domain theory consists of assumptions, facts and inference rules in the given domain. An assumption is something that is hard to prove but supposedly true. Facts are what are actually done in practice. Inference rules are more abstract facts and can be used to derive new facts. Given below is a sample theory for the chosen domain:

Assumptions:

1. In general, service requests are evenly distributed in a district.
2. All service requests are located along the streets.
3. Every service request should be served by just one cable.
4. The center of a district is where a pillar is located.

Facts:

1. Telephone cables should run only along the streets, should not go through the central areas of the street intersections and any other properties such as houses, parks.
2. There should be no loops in the network (the definition of $\mathrm{T}(j, c)$).
3. Telephone cables are available in discrete sizes: 10, 20, 30, 50, 70, and 100.
4. A joint can split a cable into cables with smaller sizes.
5. A joint's minimum capability is 10.
6. Every service request should be included in the network, $\mathrm{T}(j, c)$.
7. Cables are inside pipes.

Inference Rules:

1. If a pipe is allocated for a cable, the pipe shares some identical information with the cable, such as length, position and joints they connect with.
2. The number of joints is proportional to and less than the number of service requests because of Facts 3 and 5.
3. The cable sizes taper from the root to the leaves of $\mathrm{T}(j, c)$.

Heuristics used by experts are summarized in Table 1. Implementation cost is the major measure of optimality.

3.3 A KBDS for network design

DS is a KBDS for telecommunications network design. It is rule-based, and it models the design problem as search tree with solutions as the leaf-nodes. The design process then, is a matter of exploring the search tree for the optimum solution to each design problem.

As the search space is large, both breadth and depth wise, we use knowledge-based design techniques to narrow the search space. While this results in a system that can quickly design networks, we are reliant on the quality of the knowledge used to build the system, and exposed to the possibility that designs are of local optima.

aspect	*content*
costs	minimize all material costs
reliability	minimize number of joints
cable layout	run from boundaries to the center
cable routes	shortest possible
parts usage	maximize to their capacities
future develop	25% extra in capacity
others	how to choose an intersection/polygon how to position a joint how to run cable crossings; and how to connect pits, etc.

Table 1: The aspects and contents of heuristics in telephone cable network design.

```
--- Algorithm 5 ---
GenerateDesign
   For (all_inter)
      CalculateOD(inter);
   DesignJoint;
   AssignCable(root);
DesignJoint
   S = LeastOD(all_inter);
   If (S <> NULL)
      inter = FurthestInter(S);
      polygon = ChoosePoly(inter);
      AssignJoint(polygon);
      UpdateOD(polygon);
      DesignJoint;
```

DS's algorithm is described in Algorithm 5. *OD* stands for outgoing degree for each intersection[2], *inter* an intersection, S a set of intersections. Without loss of generality, a Euclidean distance is used (one of the alternatives is the polygon distance[3], e.g.). The following describes the automated design process. It calculates OD's for every intersection.

[2] A one-way intersection has its *OD*=1, a two-way intersection *OD*=2, and an n-way intersection *OD*=n.

[3] A polygon is a part of a street between two closest intersections on the street. The polygon distance is the number of polygons between two intersections.

Next it calls DesignJoint that implements a general heuristic employed by human experts – assigning joints from the boundary of a district towards the pillar. DesignJoint finds the intersections, S, which have the least OD's at the time. If S is not empty, it chooses an intersection, $inter$, which is the furthest away from the pillar in S. Then it selects *polygon* according to some criterion (see below) among all unserved polygons connected to $inter$. After AssignJoint(*polygon*) and UpdateOD(*polygon*), DesignJoint continues until all intersections' OD is zero. An illustrative example is shown in Figure **??**.

(Insert Figure 2 here)

CalculateOD($inter$) computes OD for all $inter$ in the district. LeastOD(all_inter) finds intersections that have the least OD's among all_inter. FurthestInter(S) chooses an intersection among S, which is the furthest away from the pillar. ChoosePoly($inter$) selects one polygon, which starts at $inter$ and has not been assigned joints, based on some criterion (a polygon is either furthest away, or closest to the pillar, or just picked at random). AssignJoint(*polygon*) allocates joints according to the services required on *polygon*. If its previous polygon has some joints that have some spare capacity, these joints should be considered to use first. If its previous polygon has some service requests that have not been served, these requests should be included to the ones of *polygon*'s. Some balancing is usually needed if *polygon* has multiple previous polygons. UpdateOD(*polygon*) decreases by one OD of the two intersections that enclose *polygon*. AssignCable(*root*) allocates cables for joints and determines the cable sizes as well.

This algorithm implements a design strategy taken commonly by experts, i.e., laying cables and assigning joints from the boundary of a district toward the pillar. Since a cable tapers from the pillar and a joint at an intersection on a polygon may serve requests on other polygons around the intersection, the purpose of this strategy is twofold: achieving the minimum number of joints and lowest cost of cables.

4 The Use of Critics for DS

A basic requirement for a KBDS, and thus for DS, is that it produce designs that not only satisfy the specifications, in accordance with the domain theory, but also are comparable to the designs produced by human experts. Other requirements are: efficiency – cost and time saving; convenience – allowing users to easily try different designs; and flexibility – allowing users the choice of involving themselves by permitting manual instruction for changes to designs produced by a KBDS. DS aims to satisfy these requirements. In building DS, the correctness and consistency critics are designed to check each version of DS in order to produce correct and consistent designs. The expertise completion critic is used to obtain sufficient knowledge – a critical part for DS's success. The alternative solution critic offers the opportunities to have various designs with or without expert's intervention.

4.1 Using correctness and consistency checking critics

These critics can be implemented by giving functions in Algorithm 4 physical meanings. In system design, two critics check the designs produced by DS: (1) CorrectnessCheck(d, p) uses the information contained in the problem description (input) to check the counterparts included in the design (output). For the sample problem described in example 1, there are 118 houses that are equivalent to 118 requests. One task CorrectnessCheck performs is to sum up all the requests that the design d (i.e., $T(j, c)$) serves. If the sum is either greater or less than 118, an error message is added into $S_{incorrect}$. Another check is that the cable network should have no loops, i.e., starting from a joint, going down it's child links, the joint will not be visited again. (2) ConsistencyCheck(d, T) works less straightforwardly. The domain theory T is used to check if there is any conflict among the design components based on the inference rules in T and/or inconsistency between d and the facts in T. For example, the number of joints should be close or equal to $\lceil 118/10 \rceil$ using inference rule 2 based on fact 3. Other consistency checking includes: #(cable segments) = #(joints) - 1, where # reads number of; #(pits) $\approx$ households/2; all pits are connected by pipes; pipe routes are overlaid on the cable network, etc. A complete check should examine all components in d with the facts and rules in T. $S_{inconsistent}$ contains the complaints found in the examination. (For the meanings of symbols and functions, refer to Algorithm 4.) Please note that in design generation, function Refine() in Algorithm 4 is replaced by ReportError() since a user of a system normally has no right to modify a system or its knowledge base. Algorithm 4 can be viewed as a simple version of *constraint-based reasoning* in which the goal is to discover some problem state that satisfies a given set of constraints.

4.2 Using expertise completion critic

Only a limited number of test cases can be studied during knowledge acquisition and system construction, due to the time constraint and number of cases available. Algorithm 1 tries to make full use of these test cases in knowledge acquisition, and to improve *DS*'s generality[4]. In principle, the more problems studied, the more general the system could become.

As is suggested in Algorithm 1, *IS* should be independent of *DS*. Our choice of *IS* is Simulated Annealing [5] that relies on an energy (cost) function, instead of the heuristics used in Algorithm 5 that are shown in Table 1 of section 3.2. This independent design system is called SA. Algorithm 1 is implemented by calling Apply(DS, p) and Apply(SA,

[4]By generality we mean that the system can perform at a comparable level for the unseen cases as it does for the seen ones.

p), here IS is replaced by SA. Both DS and SA have the same basic representation for streets, polygons, intersections, joints, cables, etc. It is shown in Figure **??** how these two systems are used in critiquing. Two sets of designs, {design1} and {design2}, are generated. Any *difference* between the two sets can be a good clue for knowledge base modification, while which design is better is not an issue concerned.

(Insert Figure 3 here)

Human expertise is used in SA to determine an appropriate cost function in equation (1). SA searches for an optimal design in terms of the cost function that is about the total cost of a design. There may be many local minima of the cost function. SA tries to avoid getting stuck at a local minimum by increasing the "temperature" so that a globally optimum solution may be found. The main features of SA are:

1. The uphill moves are controlled by a parameter T – the "temperature".
2. A higher T means a higher uphill moving probability.
3. Initially, a sufficient high T is given to allow a lot of uphill moves.
4. As time passes, T decreases and eventually arrives to a frozen point. After that, only downhill moves are allowed until the global minimum is reached.
5. State space are all possible legitimate designs.
6. A scheme s_j is a neighbor of scheme s_i if s_j can be obtained from s_i by a single change: change of either an intersection or a polygon.
7. Cost (or criterion) function (simplified):

$$\mathcal{C} = \frac{\alpha \cdot J + \beta \cdot L + \gamma \cdot X}{N} \tag{1}$$

where J – Number of Joints, L – Total length of cables, a weighted sum of the lengths of different cables by itself, X – Total length of crossing cables from one side of a street to the other side, and N – Number of services. And α, β, and γ, are constants selected by the designer to reflect the importance of each cost parameter. The optimal design should, of course, have a minimal cost factor $\mathcal{C}$. The choice of such a $\mathcal{C}$ is due to its simplicity and to that it reflects the nature of the underlying application [9]. It is noted that a linear cost factor is acceptable due to the presence of rules that have already selected between various possible designs such as minimum number of joints, etc.

DS is rule-based, and SA is neural-network-based. Knowledge used in DS is summarized in Algorithm 5, DS is basically a greedy algorithm that looks one step ahead by choosing a proper intersection and polygon locally; while knowledge used in SA is expressed in minimizing equation (1), SA is, in essence, searching for a global optimum of C. DS and SA do share the representation for their inputs and outputs, so that their designs can be compared.

The two designs are generated by using DS (Algorithm 5) and SA, depicted in Figure **??**(a) and (b) respectively. Recall that in Algorithm 5, when function ChoosePoly($inter$) is called, a criterion is needed in order to make a selection. A heuristic derived from the general rule of laying cables from a boundary to the pillar is, naturally, choosing a polygon among those available that is furthest away from the pillar. The effect of this heuristic is that a cable at the top in Figure **??**(a) goes around and along the arc towards the pillar instead of going directly to the pillar at the first possible turning intersection. If a more complex district is presented, more such (similar and subtle) problems could occur. Manually finding these problems are both difficult and impractical.

The alternative approach, SA, is used to look for different configurations of the network. SA generates one shown in Figure **??**(b). The discrepancy between the two networks provides clues to modify the heuristic. The *Comparison* module in Figure **??** checks each route in design1 against all routes in design2 until all routes in design1 are checked. By comparing designs (a) and (b) in Figure **??**, we notice that the heuristic does not give intuitively good choice at the intersections from which the pillar is one polygon away, although it works at the other intersections. As a result, when such intersections are reached, another heuristic must be used to choose the polygon that connects the intersection to the pillar. This example shows the limitation of the heuristics used, as well as the value of an independent approach in improving a knowledge base.

(Insert Figure 4 here)

4.3 Using alternative solution critics

DS takes a very small amount of time to produce a design (order of seconds), compared to the time used by experts (order of days). Although quick design is a desirable feature, so is the quality of designs. Designs generated by DS should be comparable to the designs by human experts. One way to measure the quality is to compare the design costs; another way is for designs made by DS to pass examination by experts. In either case, a correct and consistent design is a minimum requirement. We shall see if a design can be improved based on either a cost evaluation or an expert opinion.

The generation of alternative designs is a process of guided redesign based on previous designs. Two types of redesign implemented in DS are *automatic* and *interactive* redesign. They are two forms of the alternative solution critic. The former relies on a cost function, and the latter depends on expert judgment.

As was mentioned in Algorithm 5, a cable route can be altered at an intersection. Hence the responsible decision points are street intersections here. Choosing which polygon to lay cables at an intersection is based on the strategy assigned. SetInitial($WorstParts$)

in Algorithm 2 finds the responsible intersections in d_t, and resets the strategy there. There are three strategies for choosing a polygon to continue to lay cables at an intersection: furthest, closest in terms of a polygon position away from the pillar, or random. They are chosen in a round robin fashion. The default strategy is selecting the closest polygon. The *WorstParts* stack is filled with some joints which connect cables[5]. These cable routes are relatively costly, compared to the other part of the cable network. Unless $numOfDesign$ is greater than a threshold T or *DS* does not produce designs with the same costs, the algorithm tries to reset some strategies at the intersections related to joints in $WorstParts$. A new design will be generated with different strategies at the specified intersections.

Two cases are used to illustrate how Algorithms 2 and 3 work. Along with the graphic display for each case, a state (design) transition graph is also used to show the repetitive process of alternative design generation. In Algorithm 2 a critical point is set automatically (in short, automatic setting), but in Algorithm 3 it is set manually by experts (in short, manual setting).

(Insert Figures 5 and 6 here)

Case I: The previous example revisited

Take example 2 again. DS generates the design in Figure **??**(a). Automatic redesign does not create any new design in this case. However, if we manually reset at the top middle intersection (grey one in the figure), DS creates another design (b). Although design (b) has a higher cost, in this case DS cannot automatically reduce the cost to that of design (a). That is, as shown in the transition graph, automatic redesign does not bring about any new design. In most cases tested, automatic redesign did produce best designs; in some cases, manual resetting was necessary for DS to change designs from one to another. We have shown by this example that the sole reliance on the automatic setting for redesign may not necessarily produce a better design; a better design may be missed.

Case II: A non-standard pillar position

In this case, DS generates the design in Figure **??**(a). When it's in the automatic redesign mode, it produces design (b), next design (c), then back to (b), forms a loop. Now, if we manually set the bottom middle intersection (grey one in the figure (d)) after design (a), a new design (d) is constructed. From there, a series of automatic redesigns bring (d) back to (b). Judged by their costs, designs (b) and (c) are the best. *DS* oscillates between generation of (b) and (c). This oscillation could be settled at (b), since design (b) costs the least. This case is more complex than case I, for the pillar is located at the boundary instead of the center. There are more variations, as seen in the transition map.

[5] An algorithm of finding worst parts of design based on a cost function is given in Appendix A.

4.4 Remarks

For the telecommunications network design project described here, roughly two thirds of the time was spent on the traditional development of DS, one third on the development of critics, critiquing and revision. Implementing SA did not take much time (half a day) since the algorithm is simple and easy to implement. Obtaining a proper cost function C, however, took longer time with respect to SA implementation, since obtaining C involved the consultation with domain experts and experiments. By implementing these critics, the completed project has been proven to be beyond what was originally expected, and to be very useful in assisting design to expert designers.

Nevertheless, a few more points are worth mentioning regarding the use of these critics. First, it is not always necessary to implement two independent systems. Since building an independent system may be costly, we suggest the use of this approach when the following occur: (a) There are only a few known test cases; (b) Some heuristics cannot be proven generally useful; (c) There exists a simple cost function; and (d) It is too complex and costly for experts to consider *all* choices of a design. Second, without the time constraint, these two approaches (knowledge intensive versus computationally intensive) compensate each other. However, a computationally intensive approach such as simulated annealing cannot satisfy the time constraint in a practical design application, a KBDS is practical to give a "satisficing" solution within a reasonable time. Third, design experts, in trial use of automatic and manual redesigns, found these redesigns very convenient and useful for them to generate various designs to compare. Two guidelines on using these redesigns are: (a) When it is suspected that the evaluation function is out of tune, it is strongly suggested to implement the Interactive Redesign critic so that a domain expert can be involved in evaluation. (b) If a greedy algorithm is taken in generating a design, there are reasons to believe that *some* non-local optima may result in the global optimum, then it is worth of implementing the AutoReDesign critic. These two critics are not suitable for tasks such as planning, control in real time. Last, a KBDS is only a component of each critic. The input and output of a KBDS are used in critiquing. The knowledge employed by the KBDS is not used in the system evaluation.

5 Related Work

Although this work stems from the practical need, it is influenced and nurtured by some work in the community. Among many, an *expert critic* is one, which is a computer program that critiques *human-generated* solutions [10]. A survey of expert critics by Silverman [14] provides up-to-date work in the field. These critics have found many applications such as decision making, engineering design, word processing, knowledge base acquisition, and software engineering. Expert critics scrutinize the solutions (e.g., designs) produced by humans in terms of clarity, coherence, correspondence, and workability tests. As defined in [14], *clarity* means that all statements should be unambiguous; *coherence* deals with abstract truth, or the logical structure of statements; *correspondence* concerns the agreement of statements with reality; and *workability* means pragmatically verifying and validating a body of knowledge. Expert critics were built to improve the performance of users [16].

Critics suggested by Fischer *et al* [4] are an important component of *cooperative* problem solving systems, especially when they are embedded in integrated design environments. A critic, by their definition, is a system that presents a reasoned opinion about a product or action *generated by a human*. These critics detect inferior designs, provide explanations and argumentation for their "opinion" and suggest alternative solutions.

The critics proposed in this paper share many similarities with the above two groups of critics in their functions, objects they deal with, and objectives they want to achieve. But they have their own distinct features: (1) they are designed to critique the solutions generated by KBDSs, instead of by human designers; (2) they are used to assist knowledge engineers improve the KBDS and to make the KBDS generate alternative designs, not to help human designers produce alternative solutions; (3) two stages of using these critics are clearly specified - first is system designing, second is design generating by the system. Not all the critics are used in the two stages. The expertise completion critic is active in the first stage, which helps clarify knowledge and enrich the knowledge base. Correctness and consistency checking critics are functioning in both stages. The alternative solution critic is only working in the second stage. The last three are used in a similar manner as are Fischer *et al*'s critics, that is, being embedded in integrated design environments.

Intelligent tutoring systems [18] also work on solutions and problems together, but they are for people who learn a new trade. These systems are designed and developed based on a relatively good understanding of domains and mastered skills. For example, the computer tutors of Anderson *et al* [1] were based on a set of pedagogical principles derived from ACT* theory of cognition. In other words, the correct solutions are always known beforehand. In design, such knowledge is not available. The proposed critics cannot tell if the best design exists but only if a better one is accomplished. These critics also differ from intelligent tutoring systems in the way they generate critiques. The critics try to locate the differences between designs, and to find incorrectness or inconsistency of designs. Tutoring systems begin always with deviations of a solution from the standard one.

Other relevant fields are the refinement and verification of knowledge-based systems. Refining [13] is the process of fine tuning rules that discriminate between alternatives and help assure the validity of the resulting system *within the model*. When the knowledge is incomplete, it is not suitable. Critiquing shares some similarities with verification [11]. Both attack the problems of knowledge redundancy and incompleteness. Knowledge verification, however, focuses on the *demonstration* of logical correctness of the rules, wherein checks are performed for superfluous, incorrect, or missing rules, which would eventually impair system performance.

6 Conclusion and Future Work

Knowledge-based approaches are error-prone due to the subjective nature of knowledge acquisition. Studying the processes from knowledge acquisition to design generation, we identify four main problems, i.e., the *adequacy, correctness, consistency, and application* problems. A set of critics has been proposed: expertise completion, correctness and consistency checking, and alternative solution critics. They are called "critics" because all of them accomplish their tasks by critiquing designs produced by a KBDS. Algorithms that implement the critics have been given, and these critics have been used in a practical design system in two phases (designing a system and generating a design using the system).

The expertise completion critic helps a knowledge engineer reorganize and generalize the knowledge base of a KBDS. In order to obtain its critique, the critic uses the KBDS and a second, independent system, which might be much slower, less *ad hoc* than the KBDS, to run test cases. By comparing the designs of the two systems, problems can be detected in the knowledge base of the KBDS. The alternative solution critic computes alternatives to system generated designs based on information about sub-optimal parts of the original design. The heuristics which caused the sub-optimal parts are replaced by other heuristics for generating alternative designs. The sub-optimal parts of the design are identified either by evaluating the design parts using a cost function, or by displaying the current design interactively to an expert who can then point at parts of the design which he regards as sub-optimal. The correctness checking critic checks whether the designs generated by the system satisfy the original design specification. Detected errors are passed to the knowledge engineers who can modify the knowledge base to remedy the errors. The consistency checking critic examines whether the designs generated by the system satisfy a given set of consistency rules. Any inconsistency will be notified to the knowledge engineers to modify the knowledge base. These critics are a necessary set for a working knowledge-based engineering design system. The systematic application of these critics has shown, in telephone network design, promising results in knowledge acquisition, heuristics selection, and design quality control.

This work shows our effort in search of methodologies that consistently and systematically guide knowledge engineers to perform their tasks objectively and thoroughly, in order to mitigate the bottleneck problem of knowledge acquisition and the subjectivity of a knowledge-based engineering design approach. It has been shown the usefulness of the critics in this limited application domain. However, more work is needed to extend this critiquing approach to other domains, and to expand the set of critics. Future work will also be on how to find suitable independent models to test against the knowledge-based model, and how the critiquing approach can be combined with other approaches such as verification, refining, intelligent tutoring, etc. to have a systematic, usable tool for knowledge-based systems design. Another line of research will be on the objectivity in knowledge acquisition. Since the subjectivity in knowledge acquisition is difficult to avoid, we start to investigate, for some domains, using unsupervised learning algorithms to induce production rules (decision tree induction has been shown effective in a supervised environment), and acquiring knowledge at raw data level.

Acknowledgments

The suggestions made by anonymous reviewers on an early version of this paper are very helpful and highly appreciated. The permission of the Director of Research, Telecom Research Laboratories Australia, to publish this paper is hereby acknowledged.

References

[1] John R. Anderson, C. Franklin Boyle, and Brian J. Reiser. Intelligent tutoring systems. *Science*, 228:456–462, April 1985.

[2] W.J. Clancey. Viewing knowledge bases as qualitative models. *IEEE Expert*, Summer, 1989.

[3] E.A. Feigenbaum and B.G. Buchanan. Dendral and meta-dendral: roots of knowledge systems and expert system application. *Artificial Intelligence*, 59(1 - 2), 1993.

[4] G. Fischer, A.C. Lemke, T. Mastaglio, and A.I. Morch. Critics: an emerging approach to knowledge-based human-computer interaction. *Int. J. Man-Machine Studies*, 35(5):695–721, 1991.

[5] S. Kirkpatrick, C. D. Gellat, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.

[6] D.B. Lenat and E.A. Feigenbaum. On the thresholds of knowledge. *Artificial Intelligence*, 47(1-3), January 1991.

[7] H. Liu, C.D. Rowles, and W. Wen. Design, evaluation and redesign. In *IEEE Conference on AI Applications*, March 1992.

[8] H. Liu, W. Wen, and C.D. Rowles. Optimizing knowledge based system design. In *IEEE Conference on AI Applications*, February 1991.

[9] D. G. Luenberger. *Introduction to Linear and Nonlinear Programming.* Addison-Wesley Pub. Comp., Reading, Massachusetts, 1973.

[10] P.L. Miller. Attending: Critiquing a physician's management plan. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 5:449–61, September 1983.

[11] D.L. Nazareth. Issues in the verification of rule-based systems. *Int. J. Man-Machine Studies*, 30:255–71, 1989.

[12] C.D. Rowles, C. Leckie, H. Liu, and W. Wen. Automating the design of telecommunication distribution networks. In *International Conference on Artificial Intelligence in Design*, June 1991.

[13] D.L. Schmoldt. Refining rule bases for classification knowledge-based systems. *AI Applications*, 3(3):31–41, 1989.

[14] Barry G. Silverman. Survey of expert critiquing systems: Practical and theoretical frontiers. *Communications of the ACM*, 35(4), April 1992.

[15] B.G. Silverman. Critiquing human judgment using knowledge-acquisition systems. *AI Magazine*, pages 60–79, Fall 1990.

[16] B.G. Silverman and T.M. Mezher. Expert critics in engineering design: Lessons learned and research needs. *AI Magazine*, 13(1), Spring 1992.

[17] C. Tong and D. Sriram. Introduction. In C. Tong and D. Sriram, editors, *Artifical Intelligence in Engineering Design, I,II,III.* Academic Press, Inc., 1992.

[18] E. Wenger. *Artificial Intelligence and Tutoring Systems.* Morgan Kaufman, Los Altos, 1987.

Appendix

A Finding worst parts of a design

To find out the worst part or parts of a design, the evaluation routine maintains a list of joints, at which the subtrees are rooted, having the highest cost factors that are determined by equation (1). This list is called the *worst joint stack* (WJS). Initially, the WJS is empty. As was mentioned above, while the evaluation routine accumulates the cost of the subtree of T(j,c), the cost factor $\mathcal{C}'$ of the subtree is also calculated according to equation (1), and $\mathcal{C}'$ is also compared with each cost factor $\mathcal{C}_i$ in the WJS. If there exists a $\mathcal{C}_i$ in the WJS such that $\mathcal{C}' > \mathcal{C}_i$, then insert the current joint at the position before the joint with cost factor $\mathcal{C}_i$ in the WJS. If the WJS overflows after insertion, then delete the last joint in it, i.e., the one with the least cost factor. Having obtained the WJS, it is possible to choose a joint according to the hints provided by the WJS and modify the design of the subtree rooted at this joint. In this work, the WJS' size is set at nine, but it is adjustable.

Intuitively, if we can locate an intersection from which the worst route has been established, we could then try alternative choices at that decision point. Although the pillar is the center of the network that extends to every customer towards the boundary of a distribution area, routing starts from the boundary towards the pillar, to reduce the number of redundant joints [12]. The leaves of a tree are the joints on each cable route furthest away from the pillar. The algorithm is implemented as follows:

1. Choose the top joint in WJS as the worst one;

2. Find a cable route, *CR*, below the worst joint in T(j,c), which has most number of joints in WJS;

3. causal_joint (CJ) = the furthest joint (FJ) in the WJS and on the *CR*;
/* FJ is defined as a joint on CR but furthest away from a pillar */

```
4. set_intersection(CJ)
     If CJ is the last joint that leads
        a cable to an intersection
     Then
        selectOtherStrategyAt(intersection)
     Else
        PJ = findPreviousJointOf(CJ)
       set_intersection(PJ).
```

The rationale of finding the worst causal joint in the above algorithm is that the worst cable route should have its most joints appearing in the WJS on a single cable route under the first worst joint.