Proceedings of the
1999 ASME Design Engineering Technical Conferences
September 12-15, 1999, Las Vegas, Nevada

DETC99/DTM-8779

LEARNIT: A SYSTEM THAT CAN LEARN AND REUSE DESIGN STRATEGIES

Thomas F. Stahovich
Mechanical Engineering Department
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213
Email: stahov@andrew.cmu.edu

ABSTRACT

We describe a system called LearnIT that can observe an iterative solution to a parametric design problem and learn the design strategy employed. The program represents the strategy as a set of rules, which it then uses to automatically generate new design solutions when the design requirements change. Because the rules are learned from the original designer, these new solutions reflect the original designer's engineering judgment and knowledge of implicit constraints. LearnIT's approach is based on the observation that often iterative design is actually a form of debugging: each iteration is an attempt to repair a particular flaw in the design. Thus, a program can learn the designer's strategy by observing what actions are taken in response to each kind of flaw. We have found that the state of the design constraints (satisfied or not satisfied) is a good indicator of what flaws are being addressed at any given time. Because of its ability to capture and reuse the original designer's understanding of the problem, LearnIT's primary use is as a design documentation system. However, because it can learn and reuse a design strategy, it can also be considered a design automation tool.

1 Introduction

Traditionally, design rationale management tools have been intended to help store, index, and retrieve human generated design documentation (e.g., [2], [7], and [11]). The services that these tools provide have proven quite useful, but researchers have continued to look for ways to reduce the cost of managing documentation. A recent trend is the creation of methods for directly computing some of the documentation itself, thus relieving the designer of this burden (e.g., [6], [8], and [14]). While all of these tools and methods reduce the up front cost of *creating* documentation, they are comparatively less effective at reducing the downstream cost of *using* it. The human designer must still sort through the documentation, find the relevant information, interpret its meaning, and apply it to the problem at hand. Of course current tools do provide assistance in sifting through the information, however our goal is to reduce this downstream cost even further by creating a documentation system that not only can generate useful documentation, but also can apply it to new problems, without human intervention. We have developed a system called LearnIT that can observe an engineer's design activities, learn the design strategy governing his or her decisions, and then automatically apply this strategy to new design problems.

LearnIT's domain is the restricted, but important, domain of parametric design. In this domain, a design problem is characterized by a set of parameters and a set of variables. Consistent with the optimization literature (e.g., [13]), we define a parameter as a quantity that can be directly modified while a variable is a quantity that is computed from the parameters. For example, if the length, cross-sectional area, and density of a beam are the parameters, the total mass of the beam would be a variable. In all real design problems, some of the variables are subject to constraints, which may be either equality or inequality constraints. The designer's task is to find a set of parameter values that satisfy the constraints.

LearnIT sits between the designer and the modeling and analysis software as shown in Figure 1. LearnIT observes the sequence of design modifications and learns the designer's strategy for satisfying the constraints. At the implementation level, the

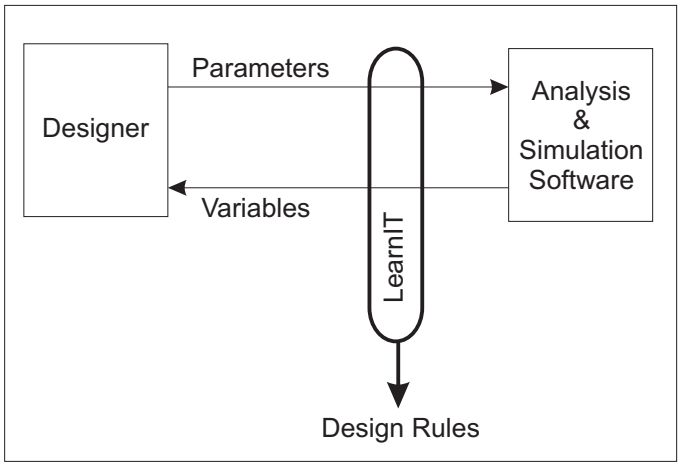

Figure 1. LearnIT observes the interaction between the designer and the modeling & analysis software and learns the design strategy employed. LearnIT records the strategy in the form of rules which it then uses for automated redesign if the design requirements should change. Parameters are quantities that can be directly changed. Variables are computed from the parameters.

program learns the conditions that determine when the designer would choose to modify any particular parameter. LearnIT expresses the design strategy as a set of if-then (production) rules. When the design requirements change, LearnIT uses these rules to automatically generate a new design solution. Because the rules are learned from the original designer, this new solution reflects the original designer's engineering judgment, knowledge of implicit constraints, and overall familiarity with the problem.

2 Example: Circuit Breaker

We use the design of a circuit breaker to illustrate LearnIT in operation. Figure 2 shows a parameterized model of the device. Figure 3 defines the parameters. In normal use, current flows from the lever to the hook. When there is a current overload, the bimetallic hook heats and bends, releasing the lever and interrupting the current flow. After the hook cools, pressing and releasing a push-rod (not shown) resets the device.

The designer's objective is to find parameter values for which the circuit breaker will trip when a 15 amp overload is applied for 5 seconds. The requirements can be expressed in terms of three constraints: (C_1) The final deflection, δ, of the hook must exceed h, the initial overlap between the hook and the lever, thus allowing the hook and lever to disengage. (C_2) The time, ΔT, at which they disengage must equal 5 seconds. (C_3) The hook stress, σ, which is the only critical stress, must be less than 100 MPa. Figure 4 shows the sequence of iterations the designer used to obtain a satisfactory solution.

Now imagine that sometime in the future the design requirements change: instead of tripping at 15 amps, the device must now trip at 25. LearnIT's task is to examine the trace of the original design session (Figure 4), identify the "strategy" employed, and use this strategy to find new parameter values that will satisfy the new requirements.

Assuming the original designer was experienced, the design trace in Figure 4 will be more like a purposeful march to a good solution than a random search through the parameter space.[1] The decisions would have been based on knowledge of similar problems, good engineering judgment, and knowledge of implicit constraints. The designer would have translated this knowledge

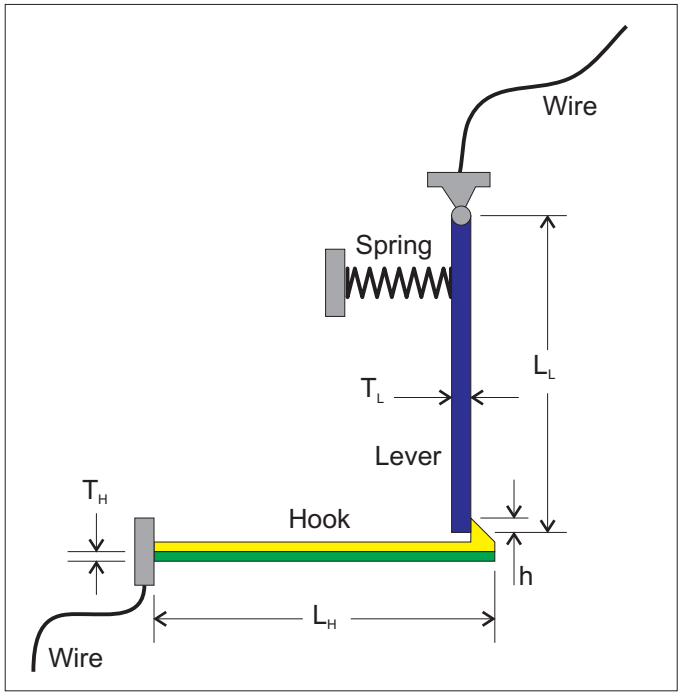

Figure 2. Parameterized circuit breaker model. Both layers of the hook have the same thickness (T_H). The width of the hook (W_H) and the width of the lever (W_L) are perpendicular to the page.

Symbol	Meaning	Initial Value
h	Initial overlap of hook and lever	1.0
T_H	Thickness of each bi-metal layer	0.2
L_H	Length of hook	10.0
W_H	Width of hook	2.5
T_L	Thickness of lever	2.0
L_L	Length of lever	10.0
W_L	Width of lever	1.5

Figure 3. Parameters for the circuit breaker. Values in millimeters.

[1] This assumption is discussed in Section 5.

Step	Parameter	$C_1: \delta > h$	$C_2: \Delta T = 5$	$C_3: \sigma < 100$
1	$T_H \rightarrow 0.1$	$<$	$>$	SAT
2	$h \rightarrow 0.7$	$<$	$>$	SAT
3	$T_H \rightarrow 0.07$	$<$	$>$	SAT
4	$L_H \rightarrow 13.0$	$<$	$>$	SAT
5	$T_H \rightarrow 0.05$	$<$	$>$	SAT
6	$h \rightarrow 0.5$	$<$	$>$	SAT
7	$T_H \rightarrow 0.04$	$<$	$>$	SAT
8	$W_H \rightarrow 1.44$	SAT	$>$	SAT
		SAT	SAT	SAT

Figure 4. Trace of circuit breaker design session. The 2^{nd} column is the parameter that was changed and the new value in millimeters. The last three columns are the states of the constraints before the parameter was changed. "$<$" = variable too small, "SAT" = constraint satisfied, and "$>$" = variable too large. δ is the final deflection of the hook. ΔT is the trip time. σ is the stress in the hook. e.g., on step 3, the third constraint was satisfied but the hook deflection was too small and the trip time too long. Each "step" may be composed of multiple iterations all changing the same parameter. e.g., in step 8, the designer changed W_H to 1.44mm in 4 iterations. Step 8 resulted in all constraints being satisfied.

into a set of preferences about how to modify the design to satisfy the constraints. For example, the designer's preferred means of making the circuit breaker trip faster was to reduce the thickness (T_H) of each layer of the bimetallic strip. This method was preferred, and thus was used first, because it is quite effective at reducing trip time: it both increases the electrical resistance so that the hook heats faster, and decreases the bending resistance so that a smaller amount of thermal expansion is needed to bend the hook. However, the designer also preferred to avoid making the layers too thin and fragile. Thus, after reducing the layers to 0.1mm, the designer chose to start reducing the initial overlap (h) between the hook and the lever. This parameter also is quite effective at reducing trip time, but it was not the first preference because the smaller h is, the more likely it is that small vibrations will accidentally trip the device.

If we continued to examine the designer's reasoning, we would find that each of the modifications in Figure 4 has a good reason behind it, just as the modifications to T_H and h did. Thus if we can examine the design trace and learn how to reproduce the designer's decision making process, we will have identified a good design strategy. Note that our goal is to be able to duplicate the designer's decision making process, not to understand the reasons behind the decisions; only the designer knows those. Said differently, we want to know what the designer's preferences are, not why they are the preferences.

With this in mind, our research task was to identify the properties of the design, or of the trace of the design session, that best indicate the designer's preferences. We explored as possible indicators: the values of the variables and parameters, the magnitudes of the gradients (partial derivatives of the variables with respect to the parameters), the order in which the parameters were modified, and the states of each of the constraints.

We found that often the best indicator appears to be a combination of the latter two properties. It appears that the kind of modifications the designer prefers to make depends on the state of the constraints, i.e., the parameter that the designer chooses to modify depends on which constraints are satisfied and which are not. Our explanation for this is that in iterative parametric design, the designer may actually be thinking in terms of repairing flaws. For each different kind of flaw, the designer chooses a specific set of remedies. The flaws themselves are manifest as unsatisfied constraints. Thus the state of the constraints determines what changes the designer will make. In the circuit breaker for example, if the hook deflection is too small – the deflection constraint is not satisfied – the designer will take actions to increase the deflection; if the stress is too large – the stress constraint is not satisfied – the designer will take other actions to reduce the stress.

Given this observation, our approach to learning the designer's strategy is based on the following heuristic: *The modifications the designer makes for any given state of the constraints are the preferred modifications for that state. If there is more than one preferred kind of modification for any given state of the constraints, the order in which those modifications were made indicates the preferred order of use.*

In the specific problem shown in Figure 4, we can identify two distinct kinds of flaws and corresponding sets of remedies. The first kind of flaw is when constraints C_1 and C_2 are not satisfied while C_3 is. This case corresponds to the hook not tripping at all. The three preferred remedies, in decreasing order of preference, are: reducing T_H, reducing h, and increasing L_H. (The first two of these preferences are described above.) The second kind of flaw is when C_1 and C_3 are satisfied while the trip time is too long (C_2 not satisfied). In this case, the design requirements are almost satisfied and just a little fine-tuning is necessary. The hook width is the preferred parameter for fine-tuning because the trip time is not overly sensitive to it.

Using our heuristic for identifying the designer's strategy, our program turns the trace of an iterative design solution into a set of if-then rules describing the designer's preferences (Figure 5). Each rule has three parts: The *antecedent* ("if" part) describes the state of the constraints for which that rule is applicable. The *consequent* ("then" part) indicates which parameter to modify and in which direction. It also indicates the *rule limit*: the maximum or minimum value, depending on whether the parameter is increased or decreased, that the parameter should have as a consequence of applying the rule. The "*expected outcome*" part of the rule describes how the modification affected the constraints in the original problem. This is used to find a suitable rule in situations where none of the rules is an exact match (see Section 3.2.1). Each of the rules is given a number to indicate the

Rule 1:
If:
C_1: δ too small
C_2: ΔT too large
C_3: σ less then yield (SAT)
Then:
decrease T_H with a limit of 0.1mm
Expected outcome:
δ will increase
ΔT will not change
σ will increase

Rule 2:
If:
C_1: δ too small
C_2: ΔT too large
C_3: σ less than yield (SAT)
Then:
decrease h with a limit of 0.7mm
Expected Outcome:
δ will increase
ΔT will not change
σ will not change

Figure 5. Sample rules from circuit breaker design.

order in which it was encountered in the original design problem. In a redesign situation, when more than one rule is applicable, the one with the lowest number is applied first.

After LearnIT has generated the set of rules describing the original circuit breaker design session, it is ready to solve our new problem in which the trip current has been increased to 25 amps. The final parameter values that LearnIT selected are shown in Figure 6. We discuss LearnIT's redesign process in detail below, here we point out a few noteworthy properties of this new solution. First, the new solution reflects the preference for reducing trip time by reducing the thickness of the bi-metal strips rather than reducing the initial overlap of the hook and lever. In the original problem, the designer reduced T_H from 0.2mm to 0.04mm and reduced h from 1.0mm to 0.5mm. In the new problem, LearnIT used most of the range of T_H while using only a little more than half of the range of h. Second, once LearnIT obtained a design that could successfully trip, it used the strategy of adjusting the hook width to fine-tune the trip time. Thus, the design strategy LearnIT used to create this new design is a faithful reproduction of the original designer's strategy.

h	T_H	L_H	W_H	T_L	L_L	W_L
0.7	0.0597	13.0	1.44	2.0	10.0	1.5

Figure 6. The final values of the parameters for the redesigned 25 amp circuit breaker. Values in millimeters.

3 How LearnIT Works

The previous section described LearnIT's approach and provided an example of the program in action. This section provides a more detailed description of the program's rule learning and redesign approaches.

3.1 Learning the Rules

As LearnIT observes a design session, it records all of the information that it will later need to generate the rule set describing the underlying design strategy. For each iteration, the program computes and records the state of each of the constraints.[2] The program uses an empirically determined 5% tolerance on the constraints to compensate for the fact that designers do not always exactly solve the constraints, but consider them to be solved when they are "close enough." This is particularly true in the early stages of the solution when the work done to exactly satisfy a constraint might be undone by later iterations intended to satisfy other constraints. By using a tolerance, LearnIT gets a reasonable estimate of when the designer would consider any particular constraint to be satisfied.

At the completion of the design session, LearnIT lumps together all consecutive iterations for which the same parameter was modified and the state of the constraints was unchanged. The result is the representation shown in Figure 4. Each of these "combined steps" describes a piece of the design strategy and is directly transformed into an if-then rule as was described above (see Figure 5 for sample rules).

The only special case rules occur when two consecutive combined steps have different constraint states. Steps 7 and 8 in Figure 4 are an example. During step 7, T_H was changed so much that the constraint on the hook deflection changed from being unsatisfied to being satisfied. We assume that changing T_H was a preference under both of these states, and thus, step 7 is turned into two rules. In general, if n constraints change state, there will be 2^n rules.

3.2 Automated Redesign

The automated redesign process is an iterative one. LearnIT finds the rule that best matches the current state of the design and changes the corresponding parameter. After making the change, LearnIT calls the analysis program to recompute the variables.

[2]The analysis program provides the values of the constrained variables; LearnIT compares these values to the desired values to determine if the constraints are satisfied.

LearnIT repeats this process until all of the constraints are satisfied or there are no applicable rules left. In the former case, the program has successfully satisfied the new requirements. In the latter case, the redesign is not completed, however, by this point the program may have already done much of the work, thus making the new designer's task much easier.

We usually start the redesign process from the parameter values used at the start of the original iterative design solution. For example, the values in Figure 3 are used as the starting point for new circuit breaker designs. This, in effect, allows the program to re-think all of the original decisions. If instead we started from the final state of the original design, the starting point might already be based on decisions that are not preferred in the context of the new problem.[3]

The next two sections describe in detail the means by which LearnIT selects the best rule and decides how much to change the corresponding parameter.

3.2.1 Finding the Best Rule The process of selecting the best rule is illustrated in Figure 7. Before the program can start the process, however, it must first evaluate the current states of the constraints. Here again, the program uses a small tolerance on the constraints; this time the tolerance compensates for numerical round off errors (we have used both a 0.5% and a 5.0% tolerance and found that both provide satisfactory results.)

If any of the rules match exactly, and they have not yet reached their limits, the one with the lowest rule number is the best.[4] Recall that a lower rule number indicates a higher preference. If all of the rules have exceeded their limit, more analysis is necessary to find the best rule.

Imagine that at a particular point in the redesign process, rule *X* is the only rule that exactly matches the state of the constraints. Furthermore, assume that when rule *X* is applied, it ordinarily increases parameter *P* up to a limit of say 10mm. If, at this point, *P* is already 12 mm (i.e., larger than *X*'s limit), should we still apply rule *X*? To answer this question, we must know if the original designer was working against a hard constraint preventing *P* from being larger than 10mm, or if this was just a preference. If there is some other rule that also increases *P*, to say a limit of 20mm, we would have evidence that 10mm was only a preference.[5] In this case, it would make sense to apply rule *X* until a limit of 20mm, if necessary.

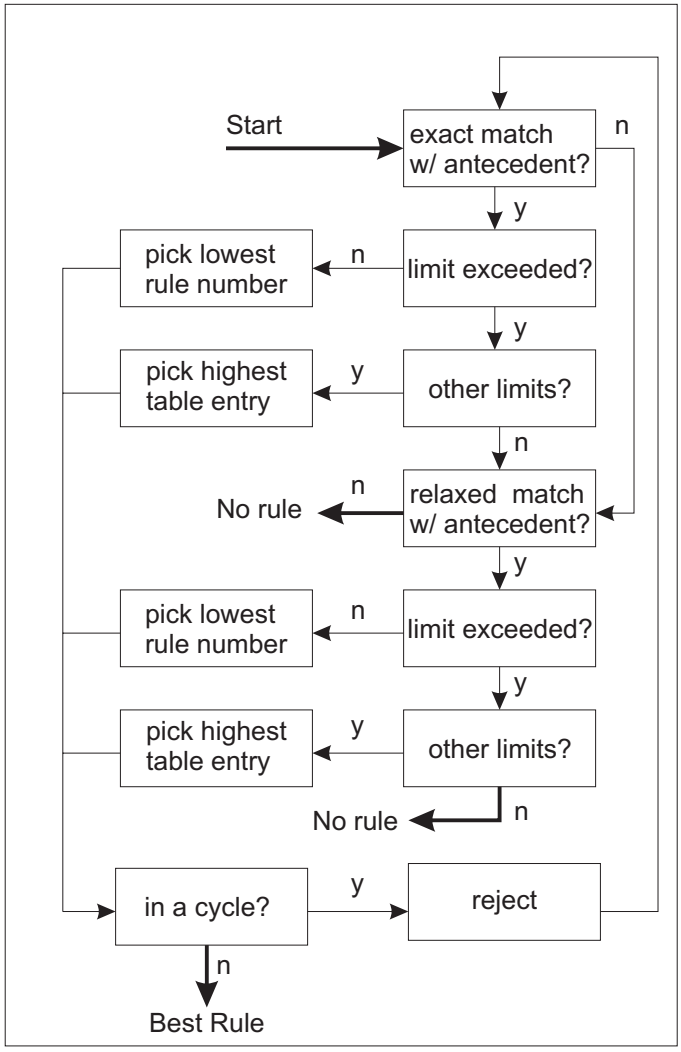

Figure 7. The rule selection process.

To facilitate this kind of analysis, LearnIT generates a table of limits. Figure 8 shows the table for the circuit breaker problem. The last 8 entries are generated directly from the rules: Each entry contains the parameter name and limit for a particular rule. The entries are listed in order of increasing rule number, with the entry for rule 1 higher in the table. Note that these entries correspond directly with the 8 steps in Figure 4. The first 4 entries in the table are the initial values of the parameters from the original design session. These are included because they help bound the range of parameter values that the original designer considered acceptable. They are listed in the order in which the parameters were first changed during the original design session.

If all of the rules matching the constraints have exceeded

[3]Starting the new redesign from the final parameter values of the original design would likely require reversing the reasoning behind each rule. This would allow the program to undo decisions that were not appropriate for the new problem. For example, a rule that decreases a particular parameter in order to decrease the trip time would be transformed into a rule that increases that parameter in order to increase the trip time.

[4]As defined above, the "rule limit" is the maximum or minimum value, depending on whether the parameter is increased or decreased, that a parameter should have as a consequence of applying a rule.

[5]This other rule would be applicable for some other state of the constraints. Otherwise, this analysis would be unnecessary and we could directly apply this other rule.

Number	Parameter	Limit Value
1	T_H	0.2
2	h	1.0
3	L_H	10.0
4	W_H	2.5
5	T_H	0.1
6	h	0.7
7	T_H	0.07
8	L_H	13.0
9	T_H	0.05
10	h	0.5
11	T_H	0.04
12	W_H	1.44

Figure 8. Limit table for the circuit breaker. The first 4 entries are the initial values for those parameters that were changed. The final 8 entries correspond to the 8 steps in Figure 4.

their limits, the best rule is the one whose parameter's next limit is highest in the table. The reason is that if the next limit is very low in the table, the designer must have tried a number of other alternatives before modifying the parameter. Thus, modifying the parameter was probably not a preferred action. Conversely, if the next limit is high in the table, the designer was probably quite comfortable modifying the parameter.

If no rule is applicable because every limit has been exceeded, LearnIT relaxes the criteria for a positive match. The first exception occurs if some of the constraints do not match the rule, but the "expected outcome" portion of the rule indicates it will help satisfy those constraints. In this case the rule is still considered applicable. For example, if a rule normally applies when the stress constraint is satisfied (e.g., when the stress is less than yield), the rule can be applied in situations when the stress constraint is not satisfied, so long as the rule tends to reduce the stress. The second exception occurs if some of the constraints do not match the rule because those constraints are already satisfied. In this case, the rule is also still considered applicable. For example if a rule normally applies when both the stress and temperature are too high, the rule can be applied to situations in which only the temperature is too high and the stress constraint is satisfied. The rule was intended to fix two flaws, but it is still likely to be useful if one of the flaws has already been fixed.

As before, if multiple rules match under the relaxed criteria, the best is the one with the lowest rule number. Similarly, as before, if the limits of each of the matching rules have been exceeded, the best rule is the one whose next limit is highest in the table. If all of the limits in the table have been exceeded, then there are no applicable rules, and the redesign process terminates.

We have found that there are some problems for which the relaxed rule matching criteria captures the designer's preferences better than the more strict criteria. This appears to be the result of learning the designer's strategy from too few examples. For instance, if the designer made a particular modification when there were two flaws (two unsatisfied constraints), LearnIT will assume that this strategy applies when exactly those two flaws exist. It is possible that this is also the preferred strategy when only one of the flaws exists, however, LearnIT will be unable to determine this without seeing an example of what is done when only one flaw exists. Clearly, without an adequate set of examples, LearnIT can generate overly specific rules. This is why we are currently working to extend our approach to enable learning across several design examples. (See Section 5.) In the meantime, however, our program provides an option to redesign using the relaxed rule matching criteria, even if there are rules that are a strict match.

After picking the best rule, LearnIT checks to make sure the rule will not setup a cycle in the solution. We have found that occasionally a pair of rules can interact in such away that the solution oscillates rather than converging. The first rule will change a parameter in one direction until the states of the constraints change and the rule is no longer applicable. At that point the second rule becomes applicable; it changes the parameter in the opposite direction, making the first rule once again applicable, and thus beginning the next cycle. To break this kind of cycle, LearnIT monitors the sequence of parameter changes. If a parameter oscillates (e.g., its value increases then decreases), LearnIT marks the second rule in the cycle as the culprit and removes it from further use. In our experience, this technique has been entirely successful in breaking the cycle and allowing the program to converge to a final solution.

3.2.2 Determining the Parameter Change Once LearnIT has found the best rule to apply, it must determine how much to change the corresponding parameter. As a first estimate, LearnIT computes the maximum amount the parameter can change before the constraints change state. (If the constraints did change state, the rule would no longer be applicable.) LearnIT uses numerical first derivatives to make the estimate. However, because this is only an estimate, we do not want to change the parameter too much in any given iteration. Thus, if the change would be substantially more than 1% of the parameter's current value, LearnIT uses a fraction of the change ($1/7^{th}$).[6] However, in all cases, LearnIT selects a new parameter that does not exceed the next parameter limit – either the limit contained in the rule itself, or if that has already been exceeded, the next applicable limit from the limit table.

LearnIT can handle discrete parameters just as well as continuous ones. As part of the problem definition, a list of legal values is attached to each discrete parameter. To determine how

[6] These values have been found to work well, but have no special significance. Using a larger fraction would tend to speed up the solution, however, it would increase the risk of using a rule inappropriately.

much to change a discrete parameter, the program uses the same procedure as for continuous ones and then rounds to the nearest legal value.

4 Related Work

4.1 Design Rationale Management

There is a large and growing body of work in design rationale capture and construction. [2], [7], and [11] offer good overviews of work relevant to the work described here. However, much of that work is focused on tools for managing documentation that is human generated whereas our goal is to automatically compute and reuse rationales.

There have been a number of recent efforts aimed at the first half of our task, automatically computing design documentation. For example, Gautier and Gruber describe a system that uses component-connection models to compute the purposes of the parts of a device [6; 8]. Similarly, Raghavan and Stahovich describe a system called ExplainIT that uses numerical simulation to compute the purposes of the geometric features on the parts of a device [14]. However, both of these systems express the purpose in natural language (English text), rather than in a machine usable form.

More similar to our task is the work of Garcia and de Souza [5]. They describe an Active Design Documentation system that computes rationales for iterative parametric design problems. Their system works from an initial design model that describes both the artifact and the decision making process for selecting parameter values. Their system compares the parameter values predicted by the decision making model with those actually selected by the designer. If the prediction and selection do not match, the designer is prompted for the rationale, which can then be added to the program's knowledge-base. Their system works from a decision making model constructed by a knowledge engineer, while our system directly infers the decision making model by observing a design session.

Perhaps most similar to the work here is a system called SketchIT [18] that can transform a rough sketch of a mechanical device into a Behavior Ensuring Parametric Model ("BEP-Model"): a parametric geometric model augmented with constraints that ensure the geometry will produce the desired behavior. The BEP-Model serves as a form of design documentation: as long as future design efforts do not violate the constraints, the device will still produce the originally intended behavior. The SketchIT and LearnIT systems are complementary: SketchIT transforms the designer's intent into constraints, LearnIT learns how the designer chooses parameter values to satisfy the constraints.

4.2 Machine Learning

Machine learning has been applied to a wide variety of different problems in design and engineering (see [4] for an overview of recent work). However, we are aware of no previous work that specifically addresses our task of learning the designer's design strategy and reusing it for automated redesign. Nevertheless, there have been a number of research efforts that touch upon issues related to our task. We summarize a sampling of those here.

Ivezic and Garrett describe a machine learning system for simulation-based support of early collaborative design [9]. However, their goal is to use statistical neural networks to learn the mapping from form to behavior, whereas our goal is to learn the strategy by which the designer modifies the design to obtain the desired behavior. Similarly, Jamalabad and Langrana describe a system that can record sensitivity information from an optimization run and use it to speed up later optimization runs [10]. Their task it to learn the shape of the design space in terms of the sensitivities, whereas our task is to learn how an expert designer would navigate the design space. Bhatta and Goel describe a model-based method for learning generic teleological mechanisms such as cascading, feedback, and feedforward [1]. Their method learns how design fragments can be assembled to achieve a satisfactory design, while our method learns how the parameters of a design should be adjusted to achieve a satisfactory design. Also, their method works from structure-behavior-function models while our method does not require a model of the device.

The work of Schwabacher *et al*. is perhaps the most directly related machine learning application [16]. They use an inductive learning algorithm to learn how to select a starting prototype for a numerical optimization problem. Thus, our systems are complementary: theirs learns how to select initial parameter values, ours learns how an expert designer gets from the initial values to a final solution.

4.3 Iterative Design

Many believe that iteration is central to design. (See [15] for a classification of iteration). For example, iteration is emphasized in introductory design texts (e.g., [17]) and it is the basis of numerical optimization techniques (e.g., [13] and [20]) and multi disciplinary optimization (e.g., [3]). Our work is related to work aimed at exploring the computational aspects of iteration. Here we summarize the most closely related efforts.

The DOMINIC II system of Orelup *et al*. was designed to automate iterative parametric redesign [12]. Our problem statements are the same: find parameter values that satisfy the constraints. However, we use very different approaches: DOMINIC II relies on hill climbing while LearnIT works from design rules learned from the designer. DOMINIC II monitors its own performance and switches from one form of hill climbing to another, as needed. Thus, it does have a notion of selecting appropri-

ate strategies at run time. However, it switches strategies based on how fast the solution is converging, whereas we attempt to duplicate the strategy of an expert designer. In fact, this is the fundamental difference between the two systems: we use problem specific information learned from an expert designer, while their goal is to solve the problem using only domain independent techniques.

The Engineous system [19] is similar to the DOMINIC II system except that it provides the ability to use problem specific rules to guide the solution process. However, these rules are manually created and entered into the system, while LearnIT automatically learns its redesign rules by observing a sample design session.

5 Discussion & Future Work

LearnIT, as its name suggests, is a machine learning system. As with all machine learning systems, the rules LearnIT learns are only as good as the lessons it is taught. Our program can only learn a good design strategy if it is shown an example of one. Thus, LearnIT works best on problems for which the designer has reasonable experience. If the designer is simply exploring the design space without direction, the strategy our program learns will also suffer from a lack of direction.

Our current solution to this problem is to provide the designer with a backtrack option. If the designer makes a few steps of unguided exploration, he or she can backtrack to the last purposeful step and then take new purposeful steps to implement the insights gained from the exploration. In future generations of the LearnIT system, we plan to develop techniques to distinguish between purposeful design iterations and unguided exploration. For example, if a sequence of design iterations returns the design to an earlier state, that sequence may have been an unsuccessful exploration. Similarly, if an iteration has detrimental effects on all of the constraints, it is possible (but not guaranteed) that the step was an unguided exploration.

Currently our program attempts to learn the designer's strategy by observing a single design session. Testing has indicated that this approach does work well: the learned rules are adequate to solve a reasonably large range of problems. However, we are currently working to increase this range by developing techniques that can learn across multiple design sessions. This will provide our program with a more complete picture of the designer's strategy, and thus will allow it to compute a more complete set of design rules. Considering multiple different training examples will also help to prevent creating rules that are either overly specific or overly general (see Section 3.2.1).

Although the rules that LearnIT produces can usually solve a wide variety of problems, it is always possible to create a new problem that is so different from the original that LearnIT is unable to solve it. In this case, the program will get as close to a solution as it can, but will stop when there are no applicable rules left. At that point the designer can take over and manually complete the problem. Once the designer has succeeded in completing the problem, LearnIT can then perform its usual analysis and add the new steps to its rule-base. The new rules can then be used to help solve other problems. Furthermore, if the designer's manual iterations change the design to a state where LearnIT has applicable rules, it can continue on with the automated redesign.

Even if the designer does need to perform some manual iterations, the designer's task is usually easier than it would otherwise be because LearnIT will almost always get part way to a solution. However our goal is to reduce, if not to completely eliminate, the need for human intervention. Towards this end, we are exploring the use of traditional numerical optimization to supplement the rule-base. We would use the rule-based approach to the extent possible, and then fall back to numerical optimization to complete the problem if necessary. The optimizer might be used to find a solution that minimizes the deviation from the learned strategy.

Numerical optimization may also help to overcome another of LearnIT's limitations. Currently our program assumes that there is the same set of constraints in both the new problem and the original one. For example, if the original problem had an equality constraint on the trip time, so must the new problem, although the desired trip time need not be the same. If the new problem has a different set of constraints, we might again be able to use a form of numerical optimization to supplement the rules. The challenge would be to make design changes which best satisfy the new constraints while still being consistent with the original rules.

For the purposes of design, large-scale systems such as airplanes and automobiles are, by necessity, decomposed into smaller modules that are manageable by individual designers. LearnIT is intended to learn how each individual module is designed, so that if necessary it can be redesigned automatically. In a related project, we are developing tools to manage the redesign of the larger system to which the modules belong. These tools use qualitative physical reasoning to determine which modules need to by modified in order to satisfy new system level design requirements. LearnIT will then use its rule-base to redesign those modules.

6 Conclusion

We have described a system called LearnIT that can observe an iterative solution to a parametric design problem and learn the design strategy employed. The program represents the strategy as a set of rules, which the program then uses to automatically generate new design solutions when the design requirements change. Because the rules are learned from the original designer, these new solutions reflect the original designer's engineering judgment, familiarity with similar problems, and knowledge of implicit constraints.

LearnIT's approach is based on the observation that in iterative design problems, designers often work in a debugging mode. At each iteration, the designer identifies the unresolved flaws in the design and chooses a design action to eliminate those flaws. We have found that the state of the constraints – whether they are satisfied or not – is a good indicator of which flaws the designer is working on at any given time. LearnIT's approach to inferring the designer's strategy can be summarized by the following heuristic: The design modifications the designer makes for any given state of the constraints are the preferred modifications for that state. If there is more than one preferred kind of modification for any given state of the constraints, the order in which those modifications were made indicates the preferred order of use.

We have tested LearnIT on a variety of problems[7] and found that our approach does accurately capture the designer's strategy. Furthermore, we have found that the rules LearnIT produces can solve a relatively large range of problems. For example, the rules learned from the design of a 15 amp circuit breaker were suitable even when the trip current was doubled to 30 amps.

Our work was motivated by the desire to reduce the cost of creating and using design documentation. From this perspective, our program provides useful capabilities. With a minimum of human intervention, our system helps to ensure that future design modifications are consistent with the reasoning behind the original design. However, our system is also useful when viewed as a design automation aide. We have indeed discovered that LearnIT is a handy tool for exploring a design space. After providing an initial training example (the solution of one design problem) one can easily change material properties, dimensions, etc., and LearnIT can automatically generate new solutions. Thus, LearnIT can both help the original designer find a satisfactory solution, and can help future designers successfully adapt the design to new requirements.

ACKNOWLEDGMENT

Thanks go to Michael Paisner who helped in the early work leading up to the development of LearnIT and to Hrishikesh Bal who helped in the implementation.

REFERENCES

[1] Sambasiva R. Bhatta and Ashok Goel. Learning generic mechanisms for innovative strategies in adaptive design. *The Journal of the Learning Sciences*, 6(4):367–396, 1997.

[2] Paul Chung and René Bañares-Alcántara Editors. Special issue: Representation and use of design rationale. *Artificial Intelligence for Engineering Design, Analysis, and Manufacturing*, 11(2), 1997.

[3] Evin J. Cramer, J. E. Dennis, Jr., Paul D. Frank, Robert Michael Lewis, and Gregory R. Shubin. Problem formulation for multidisciplinary optimization. *Siam J. Optimization*, 4(4):754–776, 1994.

[4] Alex H. B. Duffy, David C. Brown, and Ashok K. Goel Editors. Special issue: Machine learning in design. *Artificial Intelligence for Engineering Design, Analysis, and Manufacturing*, 12(2), April 1998.

[5] Ana Cristina Garcia and Clarisse Sieckenius de Souza. Add+: Including rhetorical structures in active documents. *Artificial Intelligence for Engineering Design, Analysis, and Manufacturing*, 11(2):109–124, 1997.

[6] Patrice O. Gautier and Thomas R. Gruber. Generating explanations of device behavior using compositional modeling and causal ordering. In *Eleventh National Conference on Artificial Intelligence*, 1993.

[7] Thomas Gruber, Catherine Baudin, John Boose, and Jay Weber. Design rationale capture as knowledge acquisition trade-offs in the design of interactive tools. Technical Report KSL 91-47, Stanford University, Knowledge Systems Laboratory, 1991.

[8] Thomas R. Gruber and Patrice O. Gautier. Machine-generated explanations of engineering models: A compositional modeling approach. In *1993 International Joint Conference on Artificial Intelligence*, 1993.

[9] Nenad Ivezic and James H. Garrett, Jr. Machine learning for simulation-based support of early collaborative design. *Artificial Intelligence for Engineering Design, Analysis, and Manufacturing*, 12(2):123–140, 1998.

[10] V. R. Jamalabad and N. A. Langrana. A learning shell for iterative design (L'SID): Concepts and applications. *Journal of Mechanical Design*, 120(2):203–209, June 1998.

[11] Thomas P. Moran and John M. Carroll, editors. *Design Rationale: Concepts, Techniques, and Use*. Lawrence Erlbaum Associates, Publishers, Hahwah, New Jersey, 1996.

[12] M. F. Orelup, J. R. Dixon, and M. K. Simmons. DOMINIC II: More progress towards domain independent design by iterative redesign. In *Intelligent and Integrated Manufacturing Analysis and Synthesis, Winter Annual Meeting of ASME*, pages 67–80, 1987.

[13] Panos Y. Papalambros and Douglass J. Wilde. *Principles of Optimal Design: Modeling and Computation*. Cambridge University Press, 1988.

[14] Anand Raghavan and Thomas F. Stahovich. Computing design rationales by interpreting simulations. In *ASME Design Engineering Technical Conference*, Atlanta, GA, September 1998. DETC98/DTM-5652.

[15] Michael J. Safoutin. Classification of iteration in engineering design processes. In *ASME Design Engineering Technical Conference*, Atlanta, GA, September 1998.

[7] We have considered three devices: a gear reducer and two versions of a circuit breaker. We learned the strategy used in about 6 sample designs and produced about 15 new designs.

DETC98/DTM-5672.

[16] Mark Schwabacher, Thomas Ellman, and Haym Hirsh. Learning to set up numerical optimizations of engineering designs. *Artificial Intelligence for Engineering Design, Analysis, and Manufacturing*, 12(2):173–192, 1998.

[17] Joseph Edward Shigley and Charles R. Mischke. *Mechanical engineering design*. McGraw-Hill, New York, 1989.

[18] Thomas F. Stahovich, Randall Davis, and Howard Shrobe. Generating multiple new designs from a sketch. *Artificial Intelligence*, 104(1–2):211–264, October 1998.

[19] Sui S. Tong. Engineous explores the design space. *Mechanical Engineering*, 114(2):49, February 1992.

[20] Garrett N. Vanderplaats. *Numerical Optimization Techniques for Engineering Design: with Applications*. McGraw Hill, 1984.