Using Analogical Reasoning for Mechanism Design

Ghassan Issa, Pennsylvania College of Technology
Stewart Shen, Old Dominion University
Meng Sang Chew, Lehigh University

ANALOGICAL REASONING LETS EXPERT SYSTEMS USE PRIOR EXPERIENCES TO SOLVE NEW PROBLEMS IN DIFFERENT DOMAINS. RESEARCHERS ALREADY AGREE ON THE IMPORTANCE OF ANALOGICAL REASONING FOR SOLVING ENGINEERING DESIGN PROBLEMS IN PARTICULAR, BUT ONLY A FEW SYSTEMS ACTUALLY APPLY IT. THIS LACK OF ADOPTION IS DUE TO THE COMPLEXITY OF ANALOGICAL REASONING, WHICH INCLUDES DEDUCTIVE AND INDUCTIVE INFERENCES AND MACHINE-LEARNING STRATEGIES SUCH AS LEARNING BY OBSERVATION, LEARNING BY DISCOVERY, AND LEARNING BY EXAMPLE.

Our experience developing a "traditional" expert system motivated us to develop a practical system that applies analogical reasoning to the mechanism design. Our traditional design system uses kinematic structures, which separate a mechanism's structure from its function, reducing the complexity of the design process. However, the system's various limitations led us to develop a new system that can reuse existing solutions (apply analogical reasoning) to synthesize new mechanisms; use a more informative knowledge representation, so it could reason effectively and more meaningfully; and incorporate multiple reasoning capabilities, so it could generate new conceptual designs more intelligently. We called the system Smarts (Synthesis of Mechanisms using Analogical Reasoning Techniques).

Smarts consists of three modules in a hierarchical structure (see Figure 1). The highest level is the case-based reasoner, which contains a knowledge base of previously solved design cases and training examples. Based on a mechanism's design specifications (the target case), the case-based reasoner retrieves the closest analogous case (source case) and applies a set of transformation operators to reduce the difference between the source case solution and the target specifications. If the transformation is successful, the target case is solved. If not, Smarts calls on the background knowledge stored at the second level in the heuristic reasoner, a rule-based system with its own knowledge base and inference engine. This reasoner represents knowledge as application-oriented rules that are provided by experts in the mechanisms field. If Smarts cannot solve the problem with the heuristic reasoner, it turns to its third level and last resort: the first-principle reasoner. This database consists primarily of basic design knowledge about several mechanical components that are used as building blocks for many mechanical systems.

The overall operation is managed by the controller, which switches operation from one module to another (as indicated by the solid arrows in Figure 1). The controller maintains the flow of data and knowledge (the dashed arrows), and ensures that the knowledge is compatible before it is transferred between modules. The case-based reasoner eventually uses the knowledge trans-
Knowledge representation

The overall knowledge representation in Smarts makes it easier to search for and transform source cases. It uses a mechanism's kinematic structure to represent its conceptual design, and it includes frame-like structures that contain design specifications, graphs that represent the connectivity of the links (components) of mechanisms, design rules, and database tables that describe various mechanical components.

Figure 1 shows the representation of a design case for an eight-link variable-stroke engine. The mechanism's structure is represented by a connectivity graph (equivalent to the schematic diagram in the figure). The graph nodes represent the mechanical components in the diagram, while the edges represent the joints connecting the components. Each node includes information about the component itself and its adjacent components, and includes a set of rules that govern its connectivity to the mechanism. These rules play a major role in constraining and guiding the transformation. Due to the direct relationship between the graph and the schematic diagram, Smarts needs only the graph representation to complete its task and therefore does not use the schematic diagram. However, such a diagram can be easily produced from a graph given that all the components used have been predefined in some graphics library.

Case-based reasoner. The case-based reasoner's inference engine performs search and retrieval, source case transformation, classification, indexing, and storage. To facilitate the storage and retrieval of cases, Smarts contains a system index with features that are subsets...
of the initial specifications of the source cases. These features are organized into a hierarchy based on their degree of discrimination.  

Figure 3 shows an index with six cases. Each case is a level of abstraction. At the top level, a mechanism is a "black box" with a given input and an output. Two mechanisms are analogous at this level if they have similar input and output. Moving down the hierarchy increases the detail; the bottom level contains pointers to specific design cases. The number of levels is predetermined and depends on the number of design features used to create the indexes. However, the index's breadth will increase as Smarts learns new cases.

The heuristic reasoner. A typical rule is the heuristic reasoner contains a set of rules. As part of a rule, a certainty factor is assigned, which is used to understand the rule and reason accordingly. For example, a simple rule that describes the piston component in the variable-stroke engine application is:

1. If Comp. None Pn (Degree of Binary) Then: (Structure Legal) Else: (Structure Illegal)
2. Certainty: 1.0

Exclamation: The piston must be binary. It must be connected to only two other links. This requirement is to avoid high stress and friction, which is undesirable. The certainty factor reflects the rule's importance in rejecting or accepting a mechanism. In this case, a certainty factor of 1 implies that the mechanism must be rejected if it contains a piston that is not binary. Smarts calculates a mechanism's overall certainty factor (certainty) by combining the applicable rules' certainty factors. The heuristic reasoner is divided into several sets of rules, each covering a specific design application. If the proper rule set exists for a given application, it is activated. Figure 4 shows structural rules from several variable-stroke engine applications.

The heuristic reasoner's knowledge is dynamic and changes with repeated use of the system. The user can directly access the heuristic reasoner to add, delete, or modify existing rules. During training, the user can assign rules to describe components for which the heuristic reasoner has no rules.

The first-principle reasoner. The design knowledge in the first-principle reasoner is fundamental to most mechanisms and therefore does not depend on any specific design application. Domain experts enter this knowledge before the operation of Smarts, or users can enter it during reasoning when they require more information. The first-principle reasoner contains three tables describing each component's function, structure, and constraints. For example, Tables 1–3 represent a Gear. The function table describes the component's type of motion or its use when it is part of a complete mechanism. The structural table describes how the subcomponents connect. In Table 2, the first row shows that a Gear consists of two Gear Plates (Gear1 and Gear2) connected by a Gear Joint that has two degrees of freedom. When this table is accessed, all the rows concerning the component "Gear" are read and used to connect the component to a new
Applying analogy to engineering design

Analogy is important in the creative process. It facilitates the transfer of knowledge from domains with rich knowledge and experience to others with less knowledge. It is not surprising, therefore, that researchers in engineering design and especially creative design consider analogy to be a primary tool for the conception of new ideas. Design creativity focuses on developing and analyzing tools at the conceptual level.

Examples of the application of analogy to design include design by implication, design by inversion, and systematic design by analogy. Design by implication uses an existing optimal design to solve a new but similar problem that may or may not be in the same domain. Design by inversion solves highly difficult problems that have no previous or optimal solutions by examining existing results from a different, or opposite, perspective. For example, if a designer is trying to improve a robot arm so it can perform a given task, but is encountering such constraints as cost, space, or design complexity, he might instead focus on improving just the hard to satisfy the requirements. Finally, systematic design by analogy applies procedures across domains. For example, we can use electrical circuit diagrams to synthesize mechanical systems because the same differential equations govern both.

References

| Table 1 | Function table. 
| Gear | Rotation | The gear is a component consisting of five gear plates connected by a gear-pair joint. |

| Table 2 | Structural table. 
| Gear | Gear1 | Gear2 | Gear-Arm | R | 1 |
| Gear | Gear2 | Gear1 | Gear-Arm | R | 1 |
| Gear | Gear2 | Gear2 | Gear-Arm | R | 1 |

| Table 3 | Constraints table. 
| Gear | Must not be connected | 1 |
| Gear | Must be connected to Ground | 1 |
Let $T$ be the target case, $S$ be the source case.

Compute the difference between $T$ and $S$.

case difference of $T$:
- Replace $S.UO$ with $T.UO$.
- Describe new $UO$ components.
- First applicable rules in heuristic reasoner, first-principle reasoner, or user.

Application domain:
- If heuristic reasoner contains a rule set for $T$ then $S$.
- For each rule in $S$:
  - Find rules in $T$ that match conditions and replace.
  - Find new applicable rules in $T$ and add to $S$.
- Transform $S$ to satisfy the new rules.

Constraints:
- For each link in $S$ do:
  - If a link is in $S$ then:
    - Mark for possible deletion.
  - For each link $S$ in $S$:
    - If $S$ links to $S$ links then:
      - $N = S.links - L.links$
      - Repeat until no marker links or $N = 0$
      - Delete marker links, $N = 0$
    - If $S$ links then:
      - Generate combinations of $S$ links in $S$ for deletion.
- Test the deletion of each combination.
- If no contradiction then confirm deletion.

Else:
- For each link $S$ in $S$:
  - $N = S.links - L.links$
  - Using $T$ (or $S$) create a new component.
  - Build a component in $T$ or $S$.
  - Test the addition of each combination.
- If no contradiction then confirm addition.

Figure 5. Transformation process.

Transformation follows the steps indicated in Figure 5. Smarts first replaces the mismatching $UO$ components of the source case with $UO$ components required by the target case specifications. To do so, the case-based reasoner requests information describing such components from the heuristic reasoner. If the heuristic reasoner has rules to describe the components, the case-based reasoner invokes the first-principle reasoner. If the first-principle reasoner has the information, Smarts translates it into rules and forwards them to the case-based reasoner, which stores them for later use, thus reducing the number of future accesses to the first-principle reasoner. However, if the first-principle reasoner does not have the information, the case-based reasoner requests it from the designer.

Second, if required, Smarts reorganizes the structure obtained from the source case so that it satisfies the existing rules in the target case's application domain, or any new rules that are added to it. Reorganization takes place:
- when the application domain of the target case is different from that of the source case, and the heuristic reasoner contains a rule set for the target application domain;
- when the target and source cases belong to the same application domain, and the corresponding rule set in the heuristic reasoner has been updated after the existing cases were created.
Creating mechanisms based on kinematic structure

A mechanism is any structure that permits relative motion between its links. A kinematic structure is the study of a mechanism's relative motion. So, kinematic structures are useful for conceiving new mechanism designs and have been used to synthesize many complex mechanisms. This approach requires only one constraint on a mechanism's function. Separating structure from function substantially reduces the complexity of the design process by permitting consideration of many functional and dimensional requirements, thus letting the designer explore more design concepts.

A designer using this approach begins with a set of structural specifications, such as the number of links (components), number of joints, and degree of freedom. He then constructs a graph based on this set or selects a graph from a list of previously constructed graphs that satisfy the set. Next, he labels the graph to include all possible combinations of joint types to be used. Finally, he applies a set of evaluation rules to test the feasibility of each resulting graph.

We implemented a knowledge-based system based on this process to help designers generate and select design configurations. Although the system provides useful decisions regarding a configuration's feasibility, it also has limitations. First, it can determine feasible solutions only by exhaustively generating and evaluating configurations, which is time-consuming and lengthy. Second, the graph representation was compact and efficient but did not convey sufficient information to describe the basic mechanical components. A component identification module therefore had to translate each graph's meaning and identify the mechanism's components. Finally, the system had limited reasoning capabilities that were derived primarily from its rule-based architecture.

References

Figure 1: Exhaustive generation and evaluation of design configurations.

5. Select another graph, or label existing graph using different joint combination.

Reorganization under both conditions is similar, except the first happens during transformation and the second is a maintenance option. Under both conditions, the case-based reasoner replaces rules for each component of the source case with rules from the heuristic reasoner that have matching conditions. The case-based reasoner then adds new rules from the heuristic reasoner that describe the specific component. After replacing and adding rules, the case-based reasoner modifies the structure obtained from the source case to satisfy the new rules. This ensures that the case's rules are current and eliminates any inconsistency between the rules in the design case and those in the heuristic reasoner.

Third, Smart's checks for any difference in the constraints. A constraint is a requirement that a mechanism's input or output must have in a certain manner, such as the means to vary the piston stroke in a variable-stroke engine application. A constraint is usually satisfied by one or more components in the mechanism. If the target case has no constraints or different constraints than the source case, then the components in the source case that support the constraints are marked for potential deletion in the next step.

Finally, Smart's eliminates any difference in the number of links and joints between the source and target by adding or deleting components. Because manipulating a mechanism's structure affects its function, this addition and deletion must be executed carefully and must be constrained by design rules to ensure that the resulting kinematic structure is indeed a mechanism, and one that satisfies the functional requirements.

If there is no difference in the number of links or joints, then the intermediate structure from the source case is a feasible solution to the target case, and the transformation terminates. If the source case has more links than the target case, then the case-based reasoner reduces them to match the target. The case-based reasoner eliminates those links that were previously marked, and then searches for links that can be deleted based on each link's rules. The procedure repeats until the number of remaining links and joints matches the target. Marked links that are not deleted from the resulting structure might provide an unnecessary constraint, in which case they remain marked, and the designer must decide whether to replace them.

To understand unnecessary constraints, consider this sequence of events: For a problem that requires the design of a fixed-stroke engine, the system retrieves a variable-stroke engine as the source case and transforms it, but leaves intact the components that provide the constraint (stroke variation) while the resulting solution might be correct (since the
1. **Input/Output Requirements:**
   - Input: Rotation
   - Output: Translation
   - Input Comp: Crank
   - Output Comp: Piston

2. **Application:** Variable-stroke engine

3. **Constraints:** Control output

4. **Structural Specifications:**
   - Notion: Planar
   - Degree of freedom: 1
   - Links: 8
   - Joints: 10

5. **Connectivity:**
   - Connects (Piston, Ground, Piston Rod)
   - Connects (Piston Rod, Slit)
   - Connects (Slit, Rocker, Slider)
   - Connects (Rocker, Coupler, Ground)
   - Connects (Coupler, Slit, Ground)
   - Connects (Ground, Crank, Ground)

6. **Component Description:**
   - Binary (Piston)
   - Connected to (Ground) by Piston
   - Purpose (Input, Function (Translation))
   - Rules:
     - If (Comp, name = Piston)
     - Then (Degree Binary)
   - Default Rule:
   - Explanation: Avoid high piston side thrust.

7. **Feasibility of Mechanism:**
   - Structure (Ligamental)
   - Feasibility Degree: 0.9
   - Validated Rules: None

---

Figure 6: Training example.

A variable-stroke engine is a general case of a fixed-stroke engine, so it may be more complex to design and more expensive to construct. However, because the case-based reasoner matches the constraint components, the designer will be aware of this problem in the resulting design.

If there are fewer links in the source case than in the target case, the case-based reasoner compensates by adding links to the source case. This is a more difficult than deleting links, because the system must determine the type and source of components to add, and where to add them. It does this by maintaining a component store, a list that describes the distinct design components previously used by the system. The system automatically creates a component store for each application domain. While adding links, the case-based reasoner uses the component store of the target domain, the source domain, or both. If the component store cannot solve the problem, the case-based reasoner requests assistance from the designer. The selection of components from the component store depends on how much assistance the designer provides. If no assistance is available, the system selects a combination of available components (usually a small number), and uses the rules of the source case and of the selected components to add links.

**Learning:** The case-based reasoner learns by acquiring new specific cases rather than new general concepts about the cases. However, because transformation may produce more than one solution, the case-based reasoner reduces the number of specific cases through evaluation and classification. Learning follows three steps: evaluation, classification, and indexing and storage.

**Evaluation:** New cases must prove feasible before the case-based reasoner stores them for later use. However, because each component's design rules constrain the transformation so it is valid, and because the transformation modifies the target case's specifications, any solution resulting from the transformation will be feasible. But because a case's rules are a subset of the heuristic reasoner's rules, the designer can further refine selection of the solutions by executing the heuristic's structure and sensitivities to evaluate the resulting designs. This will produce one or more cases ranked according to their degree of feasibility.

**Classification:** This two-step process reduces the number of new cases for the case-based reasoner to store. First, the case-based reasoner groups the solutions into equivalence classes based on predefined categories such as the type of components in a structure, the type of parts that connect the components, and subsequently, the rules used in each case. Then, from each solution set, the one with the highest degree of feasibility is selected for storage. This substantially reduces the number of possible solutions because it eliminates design alternatives that are structurally equivalent to or readily deducible from one another.

**Indexing and Storage:** The case-based reasoner indexes feasible solutions so they are readily accessible. It connects the solutions to the system index at the point the search encounters the first mismatch. In the earlier example that drew on the index in Figure 3, the target case's index would start under the Translation/Rotation node. This indexing method eliminates the need for extensive generalization.

**Training Phase:** If the case-based reasoner can't find an analogous source case, the designer must construct a training example. Using the design representation described earlier, a typical training example includes the design specifications, a connectivity graph describing the mechanism's structure, and a description of each design component (including rules that govern each component's structure and sensitivities). The training phase reduces the designer's effort and ensures the example's correctness.

Figure 6 shows portions of a training example for an eight-link variable-stroke engine. Once the learner enters the mechanism's
Other implementations of analogical reasoning

Simple applies analogical reasoning to structural design. It includes a database of existing design solutions indexed according to predefined features of buildings. Simple does not transform the solution of an existing case, but gives the user a set of structural elements that are used in the design of existing building solutions.

Cyclus incorporates analogical reasoning in landscape design. It uses demand-pushing, which asks questions on how to solve problems that are encountered during design, and then finds existing cases that might have a similar problem that have been solved. Although these two methods differ from that of Smarts, the object hierarchy used for matched matching resembles the system index in Smarts (see "Search and Retrieve" in the accompanying article). Unlike Smarts, the number of index levels in Cyclus is not predetermined.

Agro demonstrates a more comprehensive application of analogies to engineering design. It learns new design cases by following a training procedure or by using existing cases to solve new problems. Like Smarts and Cyclus, Agro depends on various abstraction levels to store and retrieve relevant cases; however, it does not use a global index.

Instead, it constructs different variations of a design plan at different abstraction levels by deleting leaf nodes from the original plan. It then reconstitutes these rules to find the final solution in a partial order. It begins comparing the partial solution of a new plan with existing data. If a match is found, it finds a match. This method does not require a predetermined number of abstraction levels, and allows the comparison of other types of knowledge besides surface features. However, automatically deleting leaf nodes could eliminate useful abstract plans that require some of these nodes. Furthermore, searching the database for applicable analogies can take time, even though the macro rules are stored in a partial order.

References


Synthesizing a stamper

Let's examine a practical example involving the production of a conceptual design for an assembly line stamper, a mechanism that stamps the manufacturer's name or other information onto boxes moving along an assembly line. Our target case's specifications are:

1. Requirement: Rotation/Translation
2. Input component: Gear
3. Output component: Piston
4. Application: Stamping
5. Constraints: None
6. Motion: Piston
7. Degree of freedom: 1
8. Links: 6

In the system index in Figure 3, we'll assume that Case 5 does not exist and eliminate the path (Gear/Piston, Stamping, Planar, Twof, 6 links/7 joints, Case 5). We'll also assume that the heuristic reasoner has no rule set describing the stamper.

Given the stamper's design requirements, Smarts searches its index, beginning at the top level and assigning a similarity metric to each node there. Based on the heuristics described earlier, the Rotation/Translation node receives a maximum value (exact match), followed by Translation/Translation, which has a matching output, and finally by Translation/Rotation, which receives a lower value. Smarts then selects Rotation/Translation for expansion, which results in one node (Crack/Piston). Since Crack/Piston is the only node at this level, Smarts selects it for expansion. The expansion results in a V-stroke engine and a Fixed-stroke engine, which receive the same metric value because neither matches the stamper and because there is no heuristic rule to prefer one over the other. The system then expands the nodes in the two paths while assigning similarity metrics, totaling each path's metrics, and comparing the totals. At the index's last level, the node with 8 links/10 joints receives a higher value based on the heuristic that prefers an equal or longer number of links and joints. Smarts therefore retrieves Cases 2 and 3. The selection of one of these now depends on the user's preference, which may be based on the joint and component types or on the structure. The designer may also choose to transform both cases, producing different solutions from which to choose. We'll assume here that the selected source case is the training example for the eight-link variable-stroke engine (from Figure 6).

Smarts now compares the differences between the source and target cases:

1. The input component of the source case is Crack; that of the target case is a Gear.
2. The application domain of the source case is a variable-stroke engine; that of the target case is a stamper.
3. The source case has a constraint on input (Control Output); the target case has none.
4. The source case consists of eight links (components) and six joints; the target case requires six links and seven joints.

Based on these differences, the case-based reasoner concludes that it must perform the transformation, it must:

1. Replace the input component of the source case (Crack) with a Gear.
2. Delete two links and three joints from the source case.
1. Test the Piston.
   Purpose: Output.
   Rules: Must be binary: must be connected to Piston Rod.
   Action: Do not delete.
   Reason: The Piston satisfies the target case's output requirement.

2. Test the Piston Rod.
   Purpose: Support Piston.
   Rules: Must be binary: must connect to Piston; should not connect to the Crank.
   Action: Do not delete.
   Reason: The Piston Rod cannot be separated from the Piston.

3. Test the Slot.
   Purpose: Control Output.
   Rules: Must be connected to a slider.
   Action: Delete.
   Reason: The target case has no constraint.

4. Test the Slider.
   Purpose: Controllable Output.
   Rules: Must be connected to either Ground or a slider.
   Action: Delete.
   Reason: Slider is also used as part of the constraint.

5. Test the Rocker.
   Purpose: Support.
   Rules: Must be binary and connected to Ground.
   Action: Delete.
   Reason: The Rocker in this case does not directly affect the target case specifications. No rules prevent deletion.
   Note: Deleting the Rocker results in deleting one connection to Ground.

6. Test the Coupler.
   Purpose: Support the Crank.
   Rules: Must be binary.
   Action: Delete.
   Reason: The Coupler in this case does not directly affect the target case specifications. No rules prevent deletion.

7. Test the Crank.
   Purpose: Input.
   Rules: Must be binary.
   Action: Remove with the Gear.
   Reason: Based on the target case's input component.

**Figure 7:** Line of reasoning for deleting components.

---

(3) Begin deleting the control links first, because the target case has no I/O constraints.

Before the case-based reasoner begins the transformation, however, it must acquire information about the Gear required by the target specifications. It switches to the heuristic reasoner and searches for a stamper rule set to find any rules on Gears. Because the rule set does not exist, the search continues under the rule set for the variable-stroke engine (the source case domain). If it reaches any applicable rules, the case-based reasoner would appropriate them and use them to complete the transformation. However, we'll assume here that the case-based reasoner finds no rules and therefore, switches to the first-principle reasoner to define the unknown component, Gear. The required information is then transformed into rules that are passed to the case-based reasoner. Based on the database tables (Tables 1-3), the case-based reasoner concludes that:

1. A Gear consists of three links: Gear 1, Gear 2, and a Gear Arm.
2. A Gear Joint with two degrees of freedom connects the two gears.
3. The Gear Arm also connects the gears through Revolute joints.
4. One gear must connect to Ground.

Based on the Gear description, the case-based reasoner determines that it must delete three additional links and three joints from the source case to accommodate the Gear. Therefore, it must delete axial of five links and six joints from the source case to transform it. However, before the case-based reasoner deletes links, it confirms to each component's rules while keeping track of the target specifications. In this example, the case-based reasoner, using the Case of freedom equation, determines that using the Gear will result in a mechanism with two degrees of freedom, due to the Gear Joint. This contradicts the target case's degree of freedom and requires changing that specification from one to two.

Referring to the connectivity graph in Figure 2, the case-based reasoner follows the order of reasoning shown in Figure 7. The case-based reasoner does not delete the Piston, because it matches the output component of the target case. According to the rules of both Piston and Piston Rod, the Piston must connect to a Piston Rod, therefore the case-based reasoner will not delete the Piston Rod. On the other hand, the case-based reasoner deletes the Slot and Slider because they serve as Control Output, a constraint not found in the target case. The case-based reasoner also deletes the Rocker and Coupler because they contain no rules that prevent deletion. Finally, the Gear must replace the Crank to satisfy the input requirement of the target case. To accomplish this, the case-based reasoner relies on information from the structure and constrains tables of the first-principle reasoner, and the rules in the source case. If the Gear's connection satisfies the design specifications of the target case and does not violate any design rules, the configuration is feasible. Figure 8 represents a possible design configuration.
Although our work is preliminary, it shows immense potential and demonstrates the advantages of analogical reasoning. When we used our traditional expert system implementation to create the eight-link variable-stroke engine, it produced 270 feasible configurations, only three of which were defective. When we used Smart, it generated only two feasible configurations in addition to the starting machine example.

Smart has been implemented and tested in the creation of several mechanisms in different application domains, including variable-stroke engines, mechanical robotic hands, and yo-yo rippers.

However, Smart does have limitations. It cannot understand the explanation part of the design rules. Overcoming this will give Smart stronger reasoning and will permit easier transformation of the rules from one design to another. We also need to address Smart's lack of functional requirements and rules — a limitation inherent in the design methodology. Although many of the structural rules directly affect a mechanism's function, adding functional rules will provide more efficient and functional transformation.

Acknowledgments

We are grateful to the National Science Foundation for supporting this effort through Grant No. MSM-8805149. Thanks to Cheie Stevens for editing the manuscript. We also appreciate the reviewers' kind and helpful comments.

References


Dhanuv Isha is an assistant professor of computer science at the Pennsylvania College of Technology. His research interests include analogical reasoning, case-based reasoning, and the application of machine learning to the development of knowledge-based systems for engineering design. He received his PhD in mechanical engineering from Old Dominion University, his MS in electrical engineering from Tsinghua University, and his BS in electrical engineering from Tsinghua University. He can be reached at the Computer Science Department, Applied Science University, PO Box 925296, Annand, Jordan.

Stewart Shen is a professor of computer science at Old Dominion University, where his research interests include expert systems, database systems, and multimedia information systems. He received his PhD from Northeastern University, his MA from the University of Washington, and his BS from Tsinghua University. He is a member of the IEEE Computer Society and a member of ACM and AAAI. He can be reached at the College of Science, Old Dominion University, Norfolk, VA 23529-0152.

Meng Song Chew is an associate professor of mechanical engineering at Lehigh University. His research interests are the conceptual design of mechanisms for suspension systems for space structures and flexible robots on Earth, expert systems for design and variable-stroke engine design, superconducting magnets, and instrumentation for nuclear physics experiments. He holds four US patents and has published more than 25 articles, including 10 that received the ASME Best Paper Award. He received his PhD, MS, and BS from Columbia University. He is a member of ASME, a senior member of IEEE, Tau Beta Pi, and Pi Tau Sigma, and is a Professional Engineer. He can be reached at the Department of Mathematical Engineering and Mechanics, Pohang Lab 1 F, Lehigh University, Bethlehem, PA 18015.