DOI:10.1145/2634273

**Example-based reasoning techniques developed for programming languages also help automate repetitive tasks in education.**

BY SUMIT GULWANI

# Example-Based Learning in Computer-Aided STEM Education

HUMAN LEARNING AND COMMUNICATION is often structured around examples, possibly a student trying to understand or master a certain concept through examples or a teacher trying to understand a student's misconceptions or provide feedback through example behaviors. Example-based reasoning is also used in computer-aided programming to analyze programs, including to find bugs through test-input-generation techniques[4,34] and prove correctness through inductive reasoning or random examples[15] and synthesize programs through input/output examples or demonstrations.[10,16,18,22] This article explores how such example-based reasoning techniques developed in the programming-languages community can also help automate certain repetitive and structured tasks in education, including problem generation, solution generation, and feedback generation.

These connections are illustrated through recent work (in computer science) applied to a variety of STEM subject domains, including logic,[1] automata theory,[3] programming,[27] arithmetic,[5,6] algebra,[26] and geometry.[17] More significant, the article identifies some general principles and methodologies that are applicable across multiple subject domains.

**Procedural vs. conceptual problems.** Procedural problems involve solutions that require following a specific procedure students are expected to memorize and apply; examples include mathematical procedures[5] taught in middle school or high school (such as addition, long division, greatest common divisor computation, Gaussian elimination, and basis transformations) and algorithmic procedures taught in undergraduate computer science, where students are expected to demonstrate their understanding of certain classic algorithms on specific inputs (such as breadth-first search, insertion sort, Dijkstra's shortest-path

## » key insights

- **Computing technologies can automate repetitive tasks in education, including problem generation, solution generation, and feedback generation, for numerous subject domains, including programming, logic, automata theory, arithmetic, algebra, and geometry.**
- **This can make standard and online classrooms more efficient and enable new pedagogies involving personalized workflows, saved teacher time, and improved student learning.**
- **Computer-aided education requires cross-disciplinary computing technologies; highlighted here are contributions from programming languages, human-computer interaction, and artificial intelligence; natural language understanding and machine learning also play a significant role.**

IMAGE BY SERGEY NIVENS

algorithm, regular expression to automaton conversion, and even computing tensor/inner product of qubits).

Conceptual problems include all nonprocedural ones for which there is no decision procedure (the student is expected to know and apply) but require creative thinking in the form of pattern matching or educated guesses. Problems include:

*Proof problems.* Natural deduction proofs,[1] proofs of algebraic theorems,[26] and proofs of non-regularity of languages; and

*Construction problems.* Construction of computational artifacts (such as geometric constructions,[17] automata,[3] algorithmic procedures,[27] and bitvector circuits).

**Example-based learning.** Examples have multifaceted use in educational technologies. This article classifies their use according to interaction with the underlying technology (see Figure 1).

*Input.* For several educational tasks, examples constitute a natural means to express intent. In the case of solution generation for procedural problems, teachers can demonstrate example traces with the goal of synthesizing procedures for the problems. In the case of problem generation for conceptual problems, teachers can provide an example problem with the goal of generating similar problems. In the case of feedback generation for procedural problems, teachers can provide examples of buggy traces with the goal of learning the algorithmic misconceptions a student might have. In the case of feedback generation for conceptual problems, teachers can provide examples of common local error corrections, aiming to find some appropriate combination of the corrections that correct a given incorrect attempt. For such cases, this article describes techniques inspired by research in programming by example (PBE).[10,16,18,22]

*Output.* For some educational tasks, examples constitute the intended output artifact. In the case of problem generation for procedural problems, teachers want to produce example inputs that exercise various paths in the given procedure to generate a progression of problems. In the case of feedback generation for conceptual problems, teachers want to produce counterexamples that expose incorrect behavior in the student's solution. For such cases, the article describes techniques inspired by program analysis, in particular by test-input-generation techniques[4,34] often used to find bugs.

*Inside.* Examples can also be used

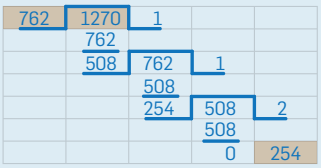
inside the underlying algorithms to perform inductive reasoning, which happens in both solution generation and problem generation for conceptual problems. It is inspired by how humans often approach problem generation and solving, with the underlying techniques inspired by research in establishing program correctness using random examples[15] and program synthesis using examples.[16]

The article next explores example-based learning technologies through specific instances, highlighting general principles. It is organized around the three key tasks in intelligent tutoring[33]—solution generation, problem generation, and feedback generation—through multiple instances of example-based learning technologies for each task. Also described are several evaluations associated with each of these instances. While several of the instances are preliminary, some have been deployed and evaluated more thoroughly.

**Figure 1. Three ways examples are used in computer-aided educational technologies as input (for intent expression); as output (to generate the intended artifact); and inside the underlying algorithm (for inductive reasoning).**

| | Procedural | Conceptual |
|---|---|---|
| Solution Generation | Input[6] | Inside[1,17] |
| Problem Generation | Output[5] | Input,[1,26] Inside[1,26] |
| Feedback Generation | Input[6] | Output,[3] Input[27] |

**Figure 2. Solution generation for procedural problems:[6] (a) demonstrate greatest common factor (GCF) procedure over inputs 762 and 1270 to produce output 254; and (b) synthesize procedure GCF automatically from the demonstration in (a).**




(a)

```
GCF (int array array T, int I1, int I2)
1 Assume T, [0,0], T[1,0] contain I1, I2 respectively.
2 for (j := 0; T[2j, j] ≠ 0; j := j + 1):
3       T[2j, j + 2] := Floor(T[2j, j + 1] ÷ T[2 j, j]);
4       T[2j + 1, j + 1] := T[2j, j + 2] × T[2 j, j]);
5       T[2j + 1, j + 1] := T[2j, j + 1] − [2j + 1, j + 1];
6       T[2j + 2, j + 2] := T[2j, j];
7 return T[2j, j + 1];
```

(b)

**Figure 3. Solution generation for geometry constructions.[17]**

(a) English Description ⇓
Construct a triangle given its base $L$ (with end-points $p_1$, $p_2$), a base angle $a$, and sum $r$ of the other two sides.

(b) PreCondition
$r > \texttt{Length}(p_1, p_2)$

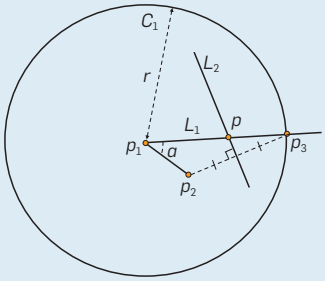
PostCondition ⇓
$\texttt{Angle}(p, p_1, p_2) = a \wedge$
$\texttt{Length}(p, p_1) + \texttt{Length}(p, p_2) = r$

(c) Random Example ⇓
$L = \texttt{Line}(p_1 = \langle 81{:}62;\ 99{:}62\rangle,\ p_2 = \langle 99{:}62;\ 83{:}62\rangle)$
$r = 88{:}07 \quad a = 0{:}81 \text{ radians} \quad p = \langle 131{:}72;\ 103{:}59\rangle$

(d) Geometry Program

```
ConstructTriangle(p1, p2, L, r, a):
L1 := ConstructLineGivenAngleLinePoint(L, a, p1);
C1 := ConstructCircleGivenPointLength(p1, r);
(p3, p5) := LineCircleIntersection(L1, C1);
L2 := PerpendicularBisector2Points(p2, p3);
p := LineLineIntersection(L1, L2);
return p;
```

(e) Geometry Construction




## Solution Generation

Solution generation involves automatically generating solutions, given a problem description in some subject domain, and is important for several reasons: It can be used to generate sample solutions for automatically generated problems; given a student's incomplete solution, it can be used to complete a solution that can be much more illustrative for the student compared to providing a completely different sample solution; and, given a student's incomplete solution, it can also be used to generate hints on the next step or toward an intermediate goal.

**Procedural problems.** Solution generation for procedural problems can be achieved by writing down and executing the corresponding procedure for a given problem. While these procedures can be written manually, technologies for automatic procedure synthesis (from examples) can enable nonprogrammers to create customized procedures on the fly. The number of such procedures and their stylistic variations in how they are taught can be significant and may not be known in advance to outsource manual creation of the procedures.

The procedures can be synthesized through PBE technology[10,16,22] traditionally applied to end-user applications. More recently, PBE has also been used to synthesize programs for spreadsheet tasks, including string transformations and table layout transformations.[18] Mathematical procedures can be viewed as spreadsheet procedures involving computation of new values from existing values in spreadsheet cells, as in string transformations that produce a new output string from substrings of input strings, and positioning that value in an appropriate spreadsheet cell, as in table transformations that reposition the content of an input spreadsheet table. Ideas from learning string and

table transformations can be combined to learn mathematical procedures from example traces, where a trace is a sequence of (value, cell) pairs.[6] Dynamic programming can be used to compute all subprograms that are consistent with various subtraces (in order of increasing length). The underlying algorithm starts out by computing, for each trace element $(v, c)$, the set of all program statements (over a teacher-specified set of operators) that can produce $v$ from previous values in the trace; see Figure 2 for synthesis of a greatest common divisor procedure from an example trace, where the teacher-specified operators include –, ×, ÷, and `Floor`.

**Conceptual problems.** Solution generation for conceptual problems often requires performing search over the underlying solution space. Following are two complementary principles, each useful across multiple subject domains while also reflecting how humans might search for such solutions.

*S1: Perform reasoning over examples as opposed to abstract symbolic reasoning.* The idea is to reason about the behavior of a solution on some or even all examples, or concrete inputs, instead of performing symbolic reasoning over an abstract input. Such reasoning reduces search time by large constant factors because executing part of a construction or proof on concrete inputs is much quicker than reasoning symbolically about the construction or proof.

*S2: Reduce solution space to solutions with small length.* The idea is to extend the solution space with commonly used macro constructs in which each such construct is a composition of several basic constructs/steps. This extension reduces the size of solutions, making search more feasible in practice.

The following illustrates these principles in multiple subject domains:

*Geometry constructions.* Geometry construction is a method for constructing a desired geometric object from other objects by applying a sequence of ruler and compass constructions (see Figure 3). Such constructions are an important part of high school geometry. The automated geometric-theorem-proving community (one of the success stories in automated reasoning) has developed

**The use of trace-based modeling allows for test-input-generation tools for generating problems with certain trace features.**

tools (such as Geometry Explorer[32] and Geometry Expert[14]) that allow students to create geometry constructions and use interactive provers to prove properties of the constructions. How are these constructions synthesized in the first place?

Geometry constructions can be regarded as straight-line programs that manipulate geometry objects—points, lines, and circles—using ruler/compass operators. Hence, their synthesis can be phrased as a program-synthesis problem[17] in which the goal is to synthesize a straight-line program, as in Figure 3d, that realizes the relational specification between inputs and outputs, as in Figure 3b.

The semantics of geometry operations is too complicated for symbolic methods for synthesis or even for verification. Ruler/compass operators are analytic functions, implying the validity of a geometry construction can be probabilistically inferred from testing on random examples, an implication that follows from the following extension of the classical result on polynomial identity testing[25] to analytic functions:

*Property 1 (probabilistic testing of analytic functions).* Let $f(X)$ and $g(X)$ be non-identical real-valued analytic functions over $R^n$. Let $Y \in R^n$ be selected uniformly at random, then with high probability over the random selection $f(Y) \neq g(Y)$. Property 1 follows from the fact that non-zero analytic functions have isolated zeroes; that is, for every zero point of an analytic function, there exists a neighborhood in which the function is non-zero. The number of non-zero points of the non-zero analytic function $f(X) - g(X)$ thus dominates the number of its zero points.[a]

The problem of synthesizing geometry constructions that satisfy a symbolic relational specification between inputs and outputs can thus be reduced to synthesizing constructions that are consis-

a Unlike the polynomial identity testing theorem,[25] which allows performing modular arithmetic over numbers selected randomly from a finite integer set for efficient evaluation, this result provides no constructive guidance on the size of the selection set and requires precise arithmetic. This process is approximated by using finite-precision floating-point arithmetic and a threshold for comparing equality; in practical experiments, it has yielded no unsoundness or incompleteness.

**Figure 4. Solution generation for natural deduction:[1] (a) sample inference rules; (b) sample replacement rules; (c) abstract proof of the problem in Figure 7b, with second column listing the 32-bit integer representation of the truth-table over five variables; (d) natural deduction proof of the problem in Figure 7b, with inference rule applications in bold; and (e) natural deduction proof of a problem similar to the one in Figure 7b with the same inference rule steps.**

| Rule Name | Premises | Conc |
|---|---|---|
| Modus Ponens (MP) | $p \rightarrow q$, $p$ | $q$ |
| Hypo. Syllogism (HS) | $p \rightarrow q$, $q \rightarrow r$ | $p \rightarrow r$ |
| Disj. Syllogism (DS) | $p \vee q$, $\neg p$ | $q$ |
| Simplification (Simp) | $p \wedge q$ | $q$ |

(a)

| Rule Name | Proposition | Equivalent Proposition |
|---|---|---|
| Distribution | $p \vee (q \wedge r)$ | $(p \vee q) \wedge (p \vee r)$ |
| Double Negation | $p$ | $\neg\neg p$ |
| Implication | $p \rightarrow q$ | $\neg p \vee q$ |
| Equivalence | $p \equiv q$ | $(p \rightarrow q) \wedge (q \rightarrow p)$ |
| | $p \equiv q$ | $(p \wedge q) \vee (\neg p \wedge \neg q)$ |

(b)

| Step | Truth-table | Reason |
|---|---|---|
| P1 | 1048575 | Premise |
| P2 | 4294914867 | Premise |
| P3 | 3722304989 | Premise |
| 1 | 16777215 | **P1, Simp** |
| 2 | 4294923605 | **P2, P3, HS** |
| 3 | 1442797055 | **1, 2, HS** |

(c)

| Step | Proposition | Reason |
|---|---|---|
| P1 | $x_1 \vee (x_2 \wedge x_3)$ | Premise |
| P2 | $x_1 \rightarrow x_4$ | Premise |
| P3 | $x_4 \rightarrow x_5$ | Premise |
| 1 | $(x_1 \vee x_2) \wedge (x_1 \vee x_3)$ | P1, Distr. |
| 2 | $x_1 \vee x_2$ | **1, Simp.** |
| 3 | $x_1 \rightarrow x_5$ | **P2, P3, HS.** |
| 4 | $x_2 \vee x_1$ | 2, Comm. |
| 5 | $\neg\neg x_2 \vee x_1$ | 4, Double Neg |
| 6 | $\neg x_2 \rightarrow x_1$ | 5, Implication |
| 7 | $\neg x_2 \rightarrow x_5$ | **6, 3, HS.** |
| 8 | $\neg\neg x_2 \vee x_5$ | 7, Implication |
| Conc | $x_2 \vee x_5$ | 8, Double Neg |

(d)

| Step | Proposition | Reason |
|---|---|---|
| P1 | $x_1 \equiv x_2$ | Premise |
| P2 | $x_3 \rightarrow \neg x_2$ | Premise |
| P3 | $(x_4 \rightarrow x_5) \rightarrow x_3$ | Premise |
| 1 | $(x_1 \rightarrow x_2) \wedge (x_2 \rightarrow x_1)$ | P1, Equivalence |
| 2 | $x_1 \rightarrow x_2$ | **1, Simp.** |
| 3 | $(x_4 \rightarrow x_5) \rightarrow \neg x_2$ | **P3, P2, HS.** |
| 4 | $\neg\neg x_2 \rightarrow \neg(x_4 \rightarrow x_5)$ | 3, Transposition |
| 5 | $x_2 \rightarrow \neg(x_4 \rightarrow x_5)$ | 4, Double Neg |
| 6 | $x_1 \rightarrow \neg(x_4 \rightarrow x_5)$ | **2, 5, HS.** |
| 7 | $x_1 \rightarrow \neg(\neg x_4 \vee x_5)$ | 6, Implication |
| 8 | $x_1 \rightarrow (\neg\neg x_4 \wedge \neg x_5)$ | 7, De Morgan's |
| Conc | $x_1 \rightarrow (x_4 \wedge \neg x_5)$ | 8, Double Neg. |

(e)

tent with randomly chosen input-output examples (Principle S1).

This reduction is the basis of Gulwani et al.'s[17] synthesis algorithm for geometry constructions involving two key steps (see also Figure 3) reflecting the two general principles discussed earlier:

- Generate random input-output examples, as in Figure 3c, from the logical description, as in Figure 3b, of the given problem using off-the-shelf numerical solvers; the logical description is in turn generated from the natural language description, as in Figure 3a, using natural language translation technology; and
- Perform brute-force search over a library of ruler-and-compass operators to find a construction, as in Figure 3d, that transforms the randomly selected input(s) into corresponding output(s).

The search is performed over an extended library of ruler and compass operators that includes higher-level primitives, such as perpendicular and angular bisectors (Principle S2). The use of an extended library not only shortens the size of a solution (allowing for efficient search) but also makes a solution more readable for students. On Gulwani et al.'s[17] benchmark of 25 problems, the extended library helped reduce the maximum solution size from 45 steps to 13 steps and increased the success rate from 75% to 100%.

*Natural deduction proofs.* Natural deduction (taught in introductory logic courses in college) is a method for establishing the validity of arguments in propositional logic, where the conclusion of an argument is derived from the premises through a series of discrete steps. Each one derives a proposition that is either a premise or derived from preceding propositions through application of some inference rule (see Figure 4a) or replacement rule (see Figure 4b), the last of which concludes the argument; see Figure 4d for a proof. Ditmarsc[29] surveyed proof assistants for teaching natural deduction (such as Pandora[9]), some of which also solve problems. This article describes a different, scalable, way to solve such problems while also paving the way for generating fresh problems, as described in the next section.

While the SAT (Boolean satisfiability), SMT (satisfiability modulo theories), and theorem-proving communities[8] continue to focus on solving large-size proof problems in a reasonable amount of time, one recent approach, by Ahmed et al.,[1] to generating natural deduction proofs in real time leverages the observation that classroom-size instances are small. The Ahmed et al. approach reflects use of the two general principles discussed earlier: abstract a proposition using its truth table, which can be represented using a bitvector representation,[20] thus avoiding expensive symbolic reasoning and reducing application of inference rules to simple bitvector operations (Principle S1); and break the proof search into multiple smaller (and hence more efficient) proof searches (Principle S2).

First, an abstract proof is discovered that involves only inference-rule applications over truth-table representation; note replacement rules are identity operations over truth-table representation. This abstract proof over truth-table representation is then refined to a complete proof over symbolic propositions by searching for sequences of replacement rules between consecutive inference rules; see Figure 4c for an example of an abstract proof and Figure 4d for its refinement to a complete proof. Note the size of an abstract proof and the number of replacement rules inserted between any two consecutive inference rules is much smaller than the size of the overall proof. The Ahmed et al. approach solved 84% of 279 problems from various textbooks (generating proofs of $\leq$27 steps), while a baseline algorithm (using symbolic representation for propositions and performing breadth-

first search for the complete proof) solved 57% of the same problems.[1]

## Problem Generation

Generating fresh problems with specific solution characteristics (such as a certain difficulty level and set of concepts) is tedious for the teacher. Automating the generation of fresh problems has several benefits: Generating problems similar to a given problem can help avoid copyright issues. It may not be legal to publish problems from textbooks on course websites. A problem-generation tool can give instructors a fresh source of problems for their assignments or lecture notes. It can also help prevent cheating[23] in classrooms or MOOCs (with unsynchronized instruction) since each student can be given a different problem with the same difficulty level. And when a student fails to solve a problem and ends up looking at the sample solution, the student may be assigned a similar practice problem by an automated system, not necessarily by human teacher. Generating problems with a given difficulty level and exercising a given set of concepts can help create personalized workflows for students. Students who solve a problem correctly may be given a problem more difficult than the last problem or that involves a richer set of concepts.

On the other hand, fresh problems create new pedagogical challenges since teachers may no longer recognize the problems and students may be unable to discuss them with one another after assignment submission. These challenges can be mitigated through solution-generation and feedback-generation capabilities.

**Procedural problems.** A procedural problem can be characterized by the trace it generates through the corresponding procedure. Various features of the trace can then be used to identify the difficulty level of a procedural problem and the concepts it exercises; for instance, a trace that executes both sides of a branch (in multiple iterations through a loop) might exercise more concepts than the one that simply executes only one side of that branch, and a trace that executes more iterations of a loop might be more difficult than the one that executes fewer iterations.

Trace-based modeling allows for test-input-generation tools[4] for generating problems with certain trace features. Andersen et al.[5] used this insight to automatically synthesize practice problems for elementary and middle school mathematics;[5] Figure 5 outlines such automatic synthesis in the context of an addition procedure. Note various addition concepts can be modeled as trace properties and, in particular, regular expressions over procedure locations. Moreover, trace-based modeling allows for use of notions of procedure coverage[34] to evaluate the comprehensiveness of a certain collection of expert-designed problems and fill any holes. It also allows for defining a partial order over problems by defining a partial order over corresponding traces based on trace features (such as number of times a loop was executed and whether the exceptional case of a conditional branch was executed) and the set of n-grams present in the trace. Andersen et al.[5] used this partial order to synthesize progressions of problems and even to analyze and compare existing progressions across multiple textbooks.

As part of follow-on work, Andersen et al. used their trace-based framework to synthesize a progression of thousands of levels for *Refraction*, a popular math puzzle game. An A/B test with 2,377 players (on the portal http://www.newgrounds.com) showed automatically synthesized progression can motivate players to play for similar lengths of time, as in the case of the original expert-designed progression. The median player in the synthesized progression group played 92% as long as the median player in the expert-designed progression group.

Effective progressions are important not just for school-based learning but also for usability and learnability in end-user applications. Many modern user applications have advanced features, and learning them constitutes a major effort by the user. Designers have thus focused on trying to reduce the effort; for example, Dong et al.[11] created a series of mini-games to teach users advanced image-manipulation tasks in Adobe Photoshop. The Andersen et al.[5] methodology may assist in creating such tutorials and games by automatically generating progressions of tasks from procedural specifications of advanced tasks.

**Conceptual problems.** Problem generation for certain conceptual problems can be likened to discovering new theorems, a search-intensive activity that can be aided by domain-specific strategies. However, two general principles are useful across multiple subject domains:

**Figure 5. Problem generation for procedural problems:[5] (a) addition procedure to add two numbers, instrumented with control locations on the right side; and (b) concepts expressed in terms of trace features and corresponding example inputs that satisfy those features (such example inputs can be generated through test-input-generation techniques).**

```
Add(int array A, int array B)
  ℓ := Max(Len(A), Len(B));
  for i=0 to ℓ-1 .
    if (i ≥ Len(A)) t := B[i];          ▷ Loop over digits (L)
    else if (i ≥ Len(B)) t := A[i];     ▷ Different # of digits (D)
    else t:=A[i]+B[i];                  ▷ Different # of digits (D)
    if (C[i] == 1) t:=t+1;              ▷ Carry from prev. step (C)
    if (t > 9) {R[i]:=t-10; C[i + 1]:=1;}
    else R[i] := t;
  if (C[ℓ] == 1) R[ℓ] := 1;             ▷ Extra digit in output (E)
```

(a)

| Concept | Trace characteristic | Example input |
|---|---|---|
| Single-digit addition | $L$ | 3 + 2 |
| Multiple-digit addition without carry | $LL^+$ | 1234 + 8765 |
| Single carry | $L^*(LC)L^*$ | 1234 + 8757 |
| Two single carries | $L^*(LC)L^+(LC)L^*$ | 1234 + 8857 |
| Double carry | $L^*(LCLC)L^*$ | 1234 + 8667 |
| Triple carry | $L^*(LCLCLC)L^*$ | 1234 + 8767 |
| Extra digit in input and new digit in output | $L^*CLDCE$ | 9234 + 900 |

(b)

*P1: Example-based template generalization.* This involves generalizing a given example problem into a template and searching for all possible instantiations of the template for valid problems. Given the search space might be vast, it is usually applicable when the validity of a given candidate problem can be checked quickly. It does not necessarily require access to a solution-generation technology, though such technology can be used to ascertain the difficulty level of the generated problems; and

*P2: Problem generation as reverse of solution generation.* This applies only to proof problems. The idea is to perform a reverse search in the solution-search space starting with the goal and leading up to the premises. It has the advantage of ensuring the generated problems have specific solution characteristics.

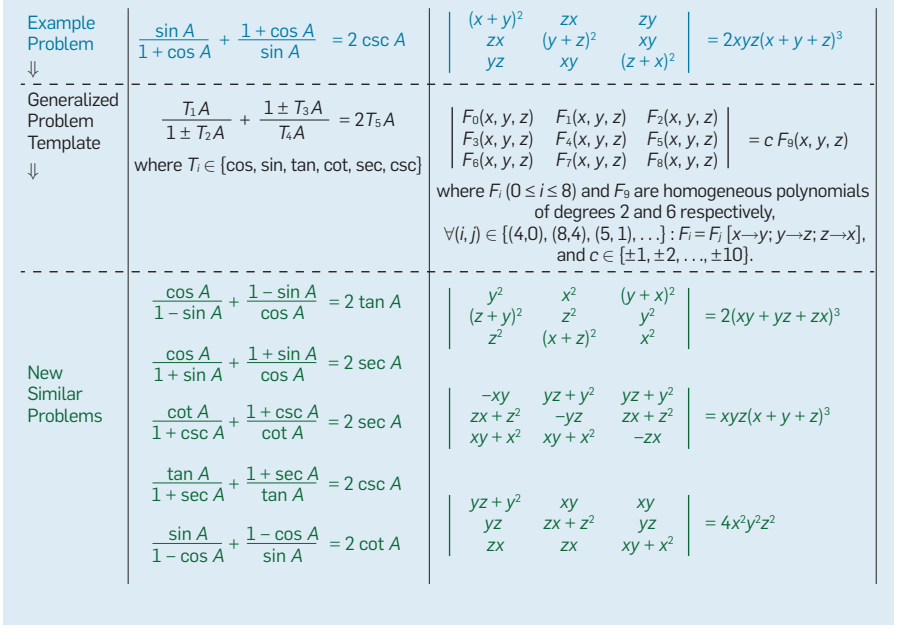
The following sections illustrate how these principles are used in multiple subject domains.

*Algebraic proof problems.* Problems that require proving algebraic identities (see Figure 6) are common in high school math curricula. Generating such problems is tedious for the teacher since the teacher cannot arbitrarily change constants (unlike in procedural problems) or variables to generate a correct problem.

The Singh et al.[26] Algebra-problem-generation methodology, as in Figure 6, uses Principle P1 to generate fresh problems similar to a given example problem. First, a given example problem is generalized into a template with a hole for each operator in the original problem to be replaced by another operator of the same type signature. The teacher can guide the template-generalization process by providing more example problems or manually editing the initially generated template. All possible instantiations of the template are automatically enumerated, and the validity of an instantiation is checked by testing on random inputs. The probabilistic soundness of such a check follows from Property 1. The methodology works for identities over analytic functions involving common algebraic operators, including trigonometry, integration, differentiation, logarithm, and exponentiation. Note the methodology would not be feasible if symbolic reasoning were used (instead of random testing) to check the validity of a candidate instantiation since symbolic reasoning is much slower (Principle S1) and the density of valid instantiations is often quite low.

*Natural deduction problems.* Figure 7 covers three interfaces for generating new natural deduction problems:[1]

**Figure 6. Problem generation for algebraic-proof problems involving identities over analytic functions (such as trigonometry and determinants);[26] a given problem is generalized into a template, and valid instantiations are found by testing on random values for free variables.**




The proposition replacement interface (see Figure 7a) finds replacements for a given premise or the conclusion in a given example problem. It generates those propositions as replacements that ensure the new problem is well defined, or one whose conclusion is implied by the premises but not by any strict subset of the premises. This interface, based on Principle P1, involves checking all possible small-size propositions as replacements. The validity of each candidate problem is checked by performing bitvector operations over bitvector-based truth table representation of the propositions[20] (Principle S1). A candidate problem is valid if the bitwise-and of the premise bitvectors is bitwise smaller than the conclusion bitvector.

The similar problem-generation interface finds problems with a solution that uses exactly the same sequence of inference rules used by a solution of an example problem. Figure 7b lists automatically generated problems, given an example problem. Figure 4e describes a solution for the first new problem in Figure 7b. Observe this solution uses exactly the same sequence of inference rules (in bold) as the solution for the original example problem in Figure 4d. The parameterized problem-generation interface finds problems with specific features (such as a given number of premises and variables, maximum size of propositions, and smallest proof involving a given number of steps and given set of rules). Figure 7c lists automatically generated problems, given some parameters. Both these interfaces find desired problems by performing a reverse search in the solution space (Principle P2) explored by the solution-generation technology for natural deduction described earlier. The similar-problem-generation interface further uses the solution template obtained from a solution of the example problem for search guidance (Principle P1).

## Feedback Generation

Feedback generation may involve identifying whether or not a student's solution is incorrect, why it is incorrect, and where or how it can be fixed. A teacher might even want to generate a hint to enable students to identify and/or fix mistakes on their own. In examination

**Figure 7. Problem-generation interfaces for natural deduction problems;[1] (a) proposition replacement; (b) similar-problem generation; and (c) parameterized-problem generation.**

Some replacements for Premise 3 in Example Problem in (b):

| |
|---|
| $\neg x_4$ |
| $x_4 \equiv x_5$ |
| $x_4 \equiv x_2$ |
| $x_4 \rightarrow x_2$ |
| $x_4 \rightarrow \neg x_1$ |

(a)

| Premise 1 | Premise 2 | Premise 3 | Conclusion |
|---|---|---|---|
| Example Problem | | | |
| $x_1 \vee (x_2 \wedge x_3)$ | $x_1 \rightarrow x_4$ | $x_4 \rightarrow x_5$ | $x_2 \vee x_5$ |
| New Similar Problems | | | |
| $x_1 \equiv x_2$ | $x_3 \rightarrow \neg x_2$ | $(x_4 \rightarrow x_5) \rightarrow x_3$ | $x_1 \rightarrow (x_4 \wedge \neg x_5)$ |
| $x_1 \wedge (x_2 \rightarrow x_3)$ | $(x_1 \vee x_4) \rightarrow \neg x_5$ | $x_2 \vee x_5$ | $(x_1 \vee x_4) \rightarrow \neg x_3$ |
| $(x_1 \vee x_2) \rightarrow x_3$ | $x_3 \rightarrow (x_1 \wedge x_4)$ | $(x_1 \wedge x_4) \rightarrow x_5$ | $x_1 \rightarrow x_5$ |
| $(x_1 \rightarrow x_2) \rightarrow x_3$ | $x_3 \rightarrow \neg x_4$ | $x_1 \vee (x_5 \vee x_4)$ | $x_5 \vee (x_2 \rightarrow x_1)$ |
| $x_1 \rightarrow (x_2 \wedge x_3)$ | $x_4 \rightarrow \neg x_2$ | $(x_3 \equiv x_5) \rightarrow x_4$ | $x_1 \rightarrow (x_3 \equiv \neg x_5)$ |

(b)

Parameters: # of premises = 3, Size of propositions ≤ 4, # of variables = 3, # of inference steps = 2, Inference rules = { DS, HS }

| Premise 1 | Premise 2 | Premise 3 | Conclusion |
|---|---|---|---|
| $(x_1 \rightarrow x_3) \rightarrow x_2$ | $x_2 \rightarrow x_3$ | $\neg x_3$ | $x_1 \wedge \neg x_3$ |
| $x_3 \rightarrow x_1$ | $(x_3 \equiv x_1) \rightarrow x_2$ | $\neg x_2$ | $x_1 \wedge \neg x_3$ |
| $(x_1 \equiv x_3) \vee (x_1 \equiv x_2)$ | $(x_1 \equiv x_2) \rightarrow x_3$ | $\neg x_3$ | $x_1 \equiv x_3$ |
| $x_1 \equiv \neg x_3$ | $x_2 \vee x_1$ | $x_3 \rightarrow \neg x_2$ | $x_1 \wedge \neg x_3$ |
| $x_3 \rightarrow x_1$ | $x_1 \rightarrow (x_2 \wedge x_3)$ | $x_3 \rightarrow \neg x_2$ | $\neg x_3$ |

(c)

settings, the teacher would even like to award a numerical grade.

Automating feedback generation is important for several reasons: First, it is quite difficult and time-consuming for a human teacher to identify what mistake a student has made. As a result, teachers often take several days to return graded assignments to their students. In contrast, if students get immediate feedback (due to automation), it can help them realize and learn from their mistakes faster and better. Furthermore, maintaining grade consistency across students and graders is difficult. The same grader may award different scores to two very similar solutions, while different graders may award different scores to the same solution.

**Procedural problems.** Generating feedback for procedural problems is relatively easy (compared to conceptual problems) since they all have a unique solution; the student's attempt can simply be syntactically compared with the unique solution. While student errors may include careless mistakes or incorrect fact recall, one common class of mistakes students make in procedural problems is employing a wrong algorithm. Van Lehn[30] identified more than 100 bugs students introduce in subtraction alone. Ashlock[7] identified a set of buggy computational patterns for a variety of algorithms based on real student data. Here are two bugs Ashlock described for the addition procedure (see Figure 5a):

- Add each column and write the sum below each column, even if it is greater than nine; and
- Add each column from left to right; if the sum is greater than nine, write the 10s digit beneath the column and the ones digit above the column to the right.

All such bugs have a clear procedural meaning and can be captured as a procedure. The buggy procedures can be automatically synthesized from examples of incorrect student traces using the same PBE technology discussed earlier in the context of solution generation for procedural problems. In fact, each of the 40 bugs described by Ashlock[7] is illustrated with a set of five to eight example traces, and Andersen et al.[6] were able to synthesize 28 (out of 40) buggy procedures from their example traces.

Identifying buggy procedures has multiple benefits; for instance, it can inform teachers about a student's misconceptions. It can also be used to automatically generate a progression of problems specifically tailored to highlighting differences between the correct procedure and the buggy procedure.

Aleven et.al.[2] used PBE technology to generalize demonstrations of correct and incorrect behaviors provided upfront by the teacher. While their generalization is restricted to loop-free procedures, teachers are able to add annotations as feedback to students who get stuck or follow a known incorrect path.

**Conceptual problems.** Feedback for proof problems can be generated by checking correctness of each individual step (assuming students are using a correct proof methodology) and using a solution-generation technology to generate proof completions from the onset of any incorrect step.[13] Here, this article focuses on feedback generation for construction problems, including two general principles useful across multiple subject domains:

*F1: Edit distance.* The idea is to find the smallest set of edits to the student's solution that will transform it into a correct solution. Such feedback informs students about where the error is in their solution and how it can be fixed. An interesting twist is to find the smallest set of edits to the problem description that will transform it into one that corresponds to the student's incorrect solution, thus capturing the common mistake of misunderstanding the problem description. Such feedback can inform students as to why their solution is incorrect. The number and type of edits can be used as a criterion for awarding numerical grades; and

*F2: Counterexamples.* The idea is to find input examples on which a student's solution does not behave correctly. Such feedback informs the student about why the solution is incorrect. The density of such inputs can be used as a criterion for awarding grades.

The following illustrates how these principles are used in different subject domains:

*Introductory programming assignments.* The standard approach to grading programming assignments is to examine its behavior on a set of test inputs that can be written manually or generated automatically.[4] Douce et al.[12] surveyed various systems developed for automated grading of programming assignments. Failing test inputs, or counterexamples, can provide guidance as to why a given solution is incorrect (Principle F2). However, this guidance alone is not ideal, especially for

beginners who find it difficult to map counterexamples to errors in their code. An edit-distance-based technique[27] offers guidance on fixing an incorrect solution (Principle F1).

Consider the problem of computing the derivative of a polynomial whose coefficients are represented as a list of integers, teaching conditionals and iteration over lists (see Figure 8a for a reference solution). For this problem, students struggled with low-level Python semantics involving list indexing and iteration bounds. Students also struggled with conceptual aspects of the problem (such as missing the corner case of handling lists consisting of single element). A teacher could leverage this knowledge of common example errors to define an edit distance model consisting of a set of weighted rewrite rules that capture potential corrections (along with their cost) for mistakes students might make in their solutions. Figure 8b includes sample rewrite rules: The first such rule transforms the index in a list access; the second transforms the right-hand side of a constant initialization; and the third transforms the arguments for the range function.

Figure 8c–e show three student programs, together with respective feedback generated by Singh et al.'s program-grading tool.[27] The underlying technique involves exploring the space of all candidate programs, applying teacher-provided rewrite rules to the student's incorrect program, to synthesize a candidate program equivalent to the reference solution while requiring a minimum number of corrections. For this purpose, the underlying technique leverages SKETCH,[28] a state-of-the-art program synthesizer that employs a SAT-based algorithm to complete program sketches (programs with holes) so they meet a given specification. Singh et al. evaluated their tool on thousands of real student attempts (at programming problems) obtained from the 2012 Introduction to Programming course at MIT (6.00) and MITx (6.00x).[27] The tool generated feedback (up to four corrections) on over 64% of all submitted solutions that were incorrect in about 10 seconds on average.

Intention-based matching approaches[19] match plans in student programs with those in a preexisting knowledgebase to provide feedback.

**The underlying technique involves exploring the space of all candidate programs, applying teacher-provided rewrite rules to the student's incorrect program, to synthesize a candidate program equivalent to the reference solution while requiring a minimum number of corrections.**

While the Singh et al. tool makes no assumption as to the algorithms or plans students can use, a key limitation is it cannot provide feedback on student attempts with big conceptual errors that cannot be fixed through local rewrite rules. Moreover, the Singh et al. tool is limited to providing feedback on functional equivalence, as opposed to performance or design patterns.

*Automata constructions.* Deterministic finite automaton (DFA) is a simple but powerful computational model with diverse applications and hence is a standard part of computer science education. JFLAP[24] is a widely used system for teaching automata and formal languages that allows for constructing, testing, and conversion between computational models but does not support grading. The following paragraphs explore a technique for automated grading of automata constructions.[3]

Consider the problem of constructing a DFA over alphabet $\{a, b\}$ for the regular language $L = \{s \mid s$ contains the substring "ab" *exactly* twice$\}$. Figure 9 includes five attempts submitted by different students and the respective feedback generated by the Alur et al.[3] automata grading tool. The underlying technique involves identifying different kinds of feedback, including edit distance over both solution and problem (Principle F1) and counterexamples (Principle F2), with each feedback associated with a numerical grade. The feedback that corresponds to the best numerical grade is then reported to the student. The reported feedback for the third attempt is based on edit distance to a correct solution, and the grade is a function of the number and kind of edits needed to convert the student's incorrect automaton into a correct automaton. In contrast, the rest of the incorrect attempts have a large edit distance and hence are based on other kinds of feedback. The second attempt and the last attempt correspond to a slightly different language description; that is, $L' = \{s \mid s$ contains the substring "ab" *at least* twice$\}$, possibly reflecting the common student mistake of misreading the problem description. The reported feedback here is based on edit distance over problem descriptions, and the associated grade is a function of the number and kind of edits required. The reported feedback for the fourth at-

**Figure 8. Automated grading of introductory programming problems:[27] (a) reference implementation (in Python) for the problem of computing a derivative of a polynomial; (b) rewrite rules that capture common errors; and (c), (d), and (e) denoting three different student submissions, along with respective feedback generated automatically.**

```
def computeDeriv(poly):
result = []
for i in range(len(poly)):
  result += [i * poly[i]]
if len(poly) == 1:
  return result
  # return [0]
else:
  return result[1:]
  # remove the leading 0
```

(a)

$$x[a] \rightarrow x[[a+1, a-1, ?a]]$$
$$x = n \rightarrow x = \{n+1, n-1, 0\}$$
$$\text{range}(a_0, a_1) \rightarrow \text{range}([0, 1, a_0-1, a_0+1], \{a_1+1, a_1-1\})$$

(b)

```
def computeDeriv(poly):
deriv, zero = [], 0
if (len(poly) == 1):
  return deriv
for e in range(0,len(poly)):
  if (poly[e] == 0):
    zero += 1
  else
    deriv.append(poly[e]*e)
return deriv
```

The program requires **3** changes:
- In the return statement `return deriv` in line 4, replace `deriv` by `[0]`.
- In the comparison expression `(poly[e] == 0)` in line 6, change `(poly[e] == 0)` to `False`.
- In the expression range `(0, len(poly))` in line 5, increment `0` by `1`.

(c)

```
def computeDeriv(poly):
idx = 1
deriv = list([])
plen = len(poly)
while idx <= plen:
  coeff = poly.pop(1)
  deriv +- [coeff*idx]
  idx = idx + 1
  if len(poly) < 2:
    return deriv
```

The program requires **1** change:
- In the function `computeDeriv`, add the base case to return `[0]` for `len(poly) = 1`.

(d)

```
def computeDeriv(poly):
length=int(len(poly)-1)
i = length
deriv = range(1,length)
if len(poly) == 1:
  deriv = [0.0]
else:
  while i >= 0:
    new = poly[i] * i
    i -= 1
    deriv[i] = new
return deriv
```

The program requires **2** changes:
- In the expression `range(1, length)` in line 4, increment `length` by `1`.
- In the comparison expression `(i >= 0)` in line 8, change operator `>=` to `!=`.

(e)

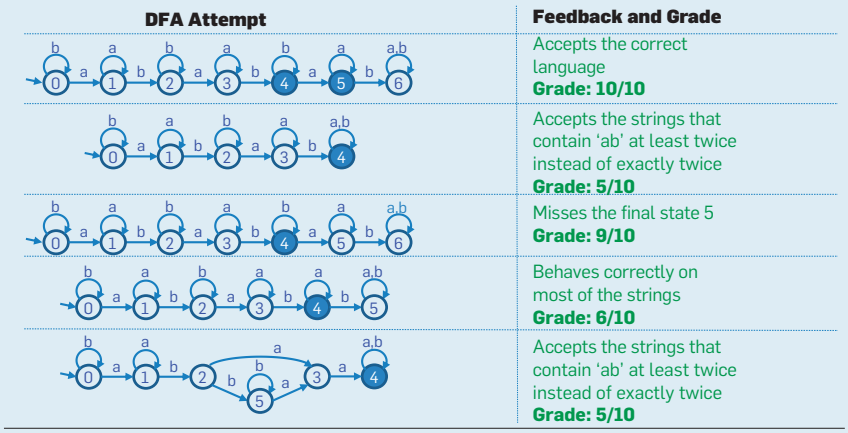
tempt, which does not involve a small edit distance, is based on counterexamples. The grade here is a function of the density of counterexamples, with more weight given to smaller-size counterexamples since students ought to have checked the correctness of their construction on smaller strings.

To automatically generate feedback, Alur et al.[3] formalized problem descriptions using a logic called MOSEL, an extension of the classical monadic-second order logic (MSO) with some syntactic sugar that allows defining regular languages in a concise, natural way. In MOSEL, the languages $L$ and $L'$ can be described by the formulas $|\texttt{indOf}(ab)| = 2$ and $|\texttt{indOf}(ab)| \geq 2$ respectively, where the `indOf` constructor returns the set of all indices where the argument string occurs. Their automata-grader tool implements synthesis algorithms that translate MOSEL descriptions into automata and vice versa. The MOSEL-to-automaton synthesizer rewrites MOSEL descriptions into MSO, then leverages standard techniques to transform an MSO formula into the corresponding automaton. The automaton-to-MOSEL synthesizer uses brute-force search to enumerate MOSEL formulas in order of increasing size to find one that matches a given automaton. Edit distance is then computed based on notions of automata distance or tree distance (in case of problem descriptions), while counterexamples are computed using automata difference.

Alur et al.[3] evaluated their automata-grader tool on 800+ student attempts to solve several problems from an automata course—CS373 at the University of Illinois at Urbana Champaign in Spring 2013. Each submission was graded by two instructors and the tool. For one of these representative problems, instructors were incorrect (having given full marks to an incorrect attempt) or inconsistent (same instructor having given different marks to syntactically equivalent attempts) for 20% of attempts. For another 25% of attempts, there was at least a three (out of 10) points discrepancy between the tool and one of the instructors; in more than 60% of these cases, the instructor concluded (after re-reviewing) that the tool's grade was more fair. The two instructors thus concluded that the tool is preferable to humans for consistency and scalability.

The automata grading tool[3] has been deployed online, providing live feedback and a variety of hints. In Fall 2013, Alur et al.,[3] together with Bjoern Hartmann of the University of California, Berkeley, conducted a user study around the utility of the tool at the University of Pennsylvania and the University of Illinois at Urbana-Champaign, observing such hints were helpful, in-

**Figure 9. Automated grading of automata problems:[3] several student attempts to construct an automaton that accepts strings containing the substring "ab" exactly twice, along with automatically generated feedback and grade.**



creased student perseverance, and improved problem-completion time.

## Conclusion

Providing personalized and interactive education (as in one-on-one tutoring) remains an unsolved problem in standard classrooms. The arrival of MOOCs, despite being an opportunity for sharing quality instruction with a large number of students, exacerbates the problem with an even higher student-to-teacher ratio. Recent advances in computer science can be brought together to rethink intelligent tutoring,[33] with the phenomenal rise of online education making this investment very timely.

This article has summarized recently published work from different areas of computer science, including programming languages,[17,27] artificial intelligence,[1,3,26] and human-computer interaction.[5] It also reveals a common thread in this interdisciplinary line of work, namely the use of examples as an input to the underlying algorithms (for intent understanding), as an output of these algorithms (for generating the intended artifact), or even inside these algorithms (for inductive reasoning). This may enable other researchers to apply these principles to develop similar techniques for other subject domains. This article should inform educators about new advances to assist various educational activities, allowing them to think more creatively about curriculum and pedagogical reforms; for instance, these advances can enable development of gaming layers that take computational thinking into K–12 classrooms.

This article has applied a rather technical perspective to computer-aided education. While the technologies can affect education in a positive manner, computer-aided education researchers must still devise ways to quantify its benefits on student learning, which may be critical to attract funding. Furthermore, this article has discussed only logical-reasoning-based techniques, but these techniques can be augmented with complementary techniques that leverage large student populations and data whose availability is facilitated by recent interest in online education platforms like Khan Academy and MOOCs; for instance, large amounts of student data can be used to collect different correct solutions to a (proof) problem, which in turn can be used to generate feedback[13] or discover effective learning pathways to guide problem selection. Large student populations can be leveraged to crowdsource tasks that are difficult to automate,[31] as in peer grading.[21] A synergistic combination of logical reasoning, machine learning, and crowdsourcing methods may lead to self-improving advanced intelligent tutoring systems that can revolutionize all education.

## Acknowledgments

I thank Moshe Y. Vardi for encouraging me to write this article. I thank Ben Zorn and the anonymous reviewers for providing valuable feedback on earlier versions of the draft.

**References**


1. Ahmed, U., Gulwani, S., and Karkare, A. Automatically generating problems and solutions for natural deduction. In *Proceedings of the International Joint Conference on Artificial Intelligence* (Beijing, Aug. 3–9, 2013).
2. Aleven, V., McLaren, B.M., Sewall, J., and Koedinger, K.R. A new paradigm for intelligent tutoring systems: Example-tracing tutors. *Artificial Intelligence in Education 19*, 2 (2009), 105–154.
3. Alur, R., D'Antoni, L., Gulwani, S., Kini, D., and Viswanathan, M. Automated grading of DFA constructions. In *Proceedings of the International Joint Conference on Artificial Intelligence* (Beijing, Aug. 3–9, 2013); tool at http://www.automatatutor.com/
4. Anand, S., Burke, E., Chen, T.Y., Clark, J., Cohen, M.B., Grieskamp, W., Harman, M., Harrold, M.J., and McMinn, P. An orchestrated survey on automated software test case generation. *Journal of Systems and Software 86, 8* (2013), 1978–2001.
5. Andersen, E., Gulwani, S., and Popovic, Z. A trace-based framework for analyzing and synthesizing educational progressions. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems* (Paris, Apr. 27–May 2). ACM Press, New York, 2013, 773–782.
6. Andersen, E., Gulwani, S., and Popovic, Z. *Programming by Demonstration Framework Applied to Procedural Math Problems Technical Report MSR-TR-2014-61.* Microsoft Research, Redmond, WA, 2014.
7. Ashlock, R. *Error Patterns in Computation: A Semi-Programmed Approach.* Merrill Publishing Company, Princeton, NC, 1986.
8. Bjørner, N. Taking satisfiability to the next level with Z3. In *Proceedings of the Sixth International Joint Conference on Automated Reasoning* (Manchester, U.K., June 26–29). Springer, 2012, 1–8.
9. Broda, K., Ma, J., Sinnadurai, G., and Summers, A.J. Pandora: A reasoning toolbox using natural deduction style. *Logic Journal of the Interest Group in Pure and Applied Logics 15*, 4 (2007), 293–304.
10. Cypher, A., Ed. *Watch What I Do: Programming by Demonstration.* MIT Press, Cambridge, MA, 1993.
11. Dong, T., Dontcheva, M., Joseph, D., Karahalios, K., Newman, M., and Ackerman, M. Discovery-based games for learning software. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems* (Austin, TX, May 5–10). ACM Press, New York, 2012, 2083–2086.
12. Douce, C., Livingstone, D., and Orwell, J. Automatic test-based assessment of programming: A review. *Journal of Educational Resources in Computing 5*, 3 (2005), 511–531.
13. Fast, E., Lee, C., Aiken, A., Bernstein, M.S., Koller, D., and Smith, E. Crowd-scale interactive formal reasoning and analytics. In *Proceedings of the ACM Symposium on User Interface Software and Technology* (St. Andrews, Scotland, Oct. 8–11). ACM Press, New York, 2013, 363–372.
14. Gao, X.-S. and Lin, Q. MMP/Geometer-a software package for automated geometric reasoning. In *Proceedings of the Fourth International Workshop on Automated Deduction in Geometry* (Hagenberg Castle, Austria, Sept. 4–6). Springer, 2004, 44–66.
15. Gulwani, S. *Program Analysis Using Random Interpretation.* Ph.D. thesis. University of California, Berkeley, 2005; http://research.microsoft.com/en-us/um/people/sumitg/pubs/dissertation.pdf
16. Gulwani, S. Synthesis from examples: Interaction models and algorithms (invited talk paper). In *Proceedings of the 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing* (Timisoara, Romania, Sept. 26–29). IEEE Computer Society, 2012, 8–14.
17. Gulwani, S., Korthikanti, V.A., and Tiwarim, A. Synthesizing geometry constructions. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation* (San Jose, CA, June 4–8), ACM Press, New York, 2011, 50–61.
18. Gulwani, S., Harris, W., and Singh, R. Spreadsheet data manipulation using examples. *Commun. ACM 55*, 8 (Aug. 2012), 97–105.
19. Johnson, W. *Intention-based Diagnosis of Novice Programming Errors.* Morgan Kaufmann, Burlington, MA, 1986.
20. Knuth, D.E. *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1.* Addison-Wesley Professional, Boston, 2011.
21. Kulkarni, C., Pang, K., Le, H., Chia, D., Papadopoulos, K., Cheng, J., Koller, D., and Klemmer, S. Peer and self assessment in massive online design classes. *ACM Transactions on Computer-Human Interaction 20*, 6 (2013).
22. Lieberman, H. *Your Wish Is My Command: Programming by Example.* Morgan Kaufmann, Burlington, MA, 2001.
23. Mozgovoy, M., Kakkonen, T., and Cosma, G. Automatic student plagiarism detection: Future perspectives. *Journal of Educational Computing Research 43*, 4 (2010), 511–531.
24. Rodger, S. and Finley, T. *JFLAP: An Interactive Formal Languages and Automata Package.* Jones and Bartlett Publishers, Inc., Sudbury, MA, 2006.
25. Schwartz, J.T. Fast probabilistic algorithms for verification of polynomial identities. *Journal of the ACM 27*, 4 (1980), 701–717.
26. Singh, R., Gulwani, S., and Rajamani, S. Automatically generating algebra problems. In *Proceedings of the 26th conference on Artificial Intelligence* (Toronto, July 22–26). AAAI Press, 2012.
27. Singh, R., Gulwani, S., and Solar-Lezama, A. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th annual ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, June 16–22). ACM Press, New York, 2013, 15–26.
28. Solar-Lezama, A. *Program Synthesis By Sketching.* Ph.D. thesis. University of California, Berkeley, 2008; http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-177.pdf
29. Van Ditmarsch, H. User interfaces in natural deduction programs. In *Proceedings of the User Interfaces for Theorem Provers Workshop* (Eindhoven, The Netherlands, July 1998), 87–95.
30. VanLehn, K. *Mind Bugs: The Origins of Procedural Misconceptions.* MIT Press, Cambridge, MA, 1991.
31. Weld, D.S., Adar, E., Chilton, L., Hoffmann, R., Horvitz, E., Koch, M., Landay, J., Lin, C.H., and Mausam, M. Personalized online education: A crowdsourcing challenge. In *Proceedings of the Fourth Human Computation Workshop at the 26th Conference on Artificial Intelligence* (Toronto, July 22–26, 2012).
32. Wilson, S. and Fleuriot, J.D. Combining dynamic geometry, automated geometry theorem proving and diagrammatic proofs. In *Proceedings of the User Interfaces for Theorem Provers Workshop* (Edinburgh, Apr.). Springer, 2005.
33. Woolf, B. *Building Intelligent Interactive Tutors.* Morgan Kaufman, Burlington, MA, 2009.
34. Zhu, H., Hall, P.A.V., and May, J.H.R. Software unit test coverage and adequacy. *ACM Computing Surveys 29*, 4 (Dec. 1997), 366–427.


**Sumit Gulwani** (sumitg@microsoft.com) is a principal researcher at Microsoft Research, Redmond, WA, adjunct faculty in the Department of Computer Science and Engineering at the Indian Institute of Technology, Kanpur, India, and affiliate faculty in the Department of Computer Science & Engineering at the University of Washington, Seattle.