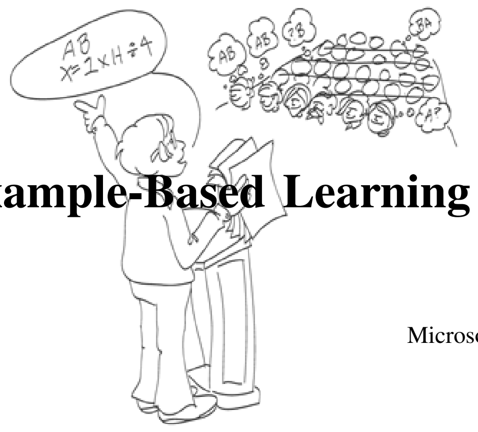
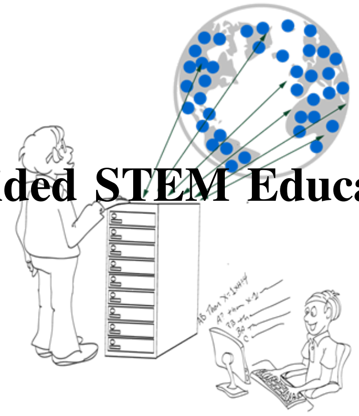




Example-Based Learning in Computer-Aided STEM Education

Sumit Gulwani

Microsoft Research, Redmond, WA, USA

sumitg@microsoft.com

Abstract

Human learning is often structured around examples. Interestingly, example-based reasoning has also been heavily used in computer-aided programming. In this article, we describe how techniques inspired from example-based program analysis and synthesis can be used for various tasks in Education including problem generation, solution generation, and feedback generation. We illustrate this using recent research results that have been applied to a variety of STEM subject domains including logic, automata theory, programming, arithmetic, algebra, and geometry. We classify these subject domains into procedural and conceptual content and highlight some general technical principles as per this classification. These results advance the state-of-the-art in intelligent tutoring, and can play a significant role in enabling personalized and interactive education in both standard classrooms and MOOCs.

1. Introduction

Human learning and communication is often structured around examples—be it a student trying to understand or master a certain concept using examples, or be it a teacher trying to understand a student's misconceptions or provide feedback using example behaviors. Interestingly, example-based reasoning has also been used in computer-aided programming community for (a) analyzing programs, including finding bugs using test input generation techniques [4, 34] or proving correctness using inductive reasoning or random examples [15], and (b) synthesizing programs using input-output examples or demonstrations [10, 16, 18, 22]. In this article, we show that such example-based reasoning techniques developed in the programming languages community can also help automate several repetitive and structured tasks in Education including problem generation, solution generation, and feedback generation.

We illustrate these connections by highlighting some recent work (from across various computer science areas) that has been applied to a wide variety of STEM subject domains including logic [1], automata theory [3], programming [27], arithmetic [5, 6], algebra [26], and geometry [17]. More significantly, we identify some general principles and methodologies that are applicable across multiple subject domains. For this purpose, we introduce a useful generalization of the above-mentioned subject domains into procedural and conceptual problems.

Procedural vs. Conceptual Problems *Procedural problems* are those whose solution requires following a specific procedure that the student is expected to memorize and apply. Examples of such procedural problems include: (a) Mathematical procedures [5] taught in middle-school or high-school curriculum such as addition, long division, GCD/LCM computation, Gaussian elimination, and basis transformations. (b) Algorithmic procedures taught in undergraduate computer science curriculum, wherein students are expected to demonstrate understanding of certain classic algorithms (on specific inputs), such as breadth-first search, insertion sort, Dijkstra's shortest-path algorithm, regular expression to automaton conversion, or even computing tensor/inner product of qubits.

	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 1. Examples are used in three different ways in computer-aided educational technologies: as an *input* (for intent expression), as an *output* (to generate the intended artifact), and *inside* the underlying algorithm (for inductive reasoning).

Conceptual problems include all non-procedural problems for which there is no decision procedure (that the student is expected to know and apply), but require creative thinking in the form of pattern matching or making educated guesses. Besides possibly other kinds of problems, these include:

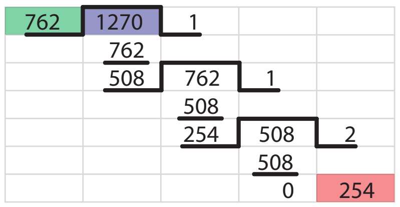
- *Proof problems*: E.g., natural deduction proofs [1], proofs of algebraic theorems [26], proofs of non-regularity of languages.
- *Construction problems*: E.g., construction of computational artifacts such as geometric constructions [17], automata [3], algorithmic procedures [27], and bitvector circuits.

Example-based Learning Examples have multi-faceted use in the educational technologies that we present in this article. Below, we classify this usage according to interaction with the underlying technology (Fig. 1).

Input: For several educational tasks, examples act as a natural means to express intent. In case of solution generation for procedural problems (§2.1), teachers can demonstrate example traces with the goal of synthesizing procedures for those problems. In case of problem generation for conceptual problems (§3.2), teachers can provide an example problem with the goal of generating similar problems. In case of feedback generation for procedural problems (§4.1), teachers can provide examples of buggy traces with the goal of learning any algorithmic misconceptions that the student might have. In case of feedback generation for conceptual problems (§4.2), teachers can provide examples of common local error corrections with the goal of finding some appropriate combination of those corrections that can correct a given incorrect attempt. For these cases, we describe techniques inspired by work in the area of programming by example (PBE) [10, 16, 18, 22].

Output: For some educational tasks, examples act as the intended output artifact. In case of problem generation for procedural problems (§3.1), teachers desire to produce example inputs that exercise various paths in the given procedure to generate a progression of problems. In case of feedback generation for conceptual problems (§4.2), teachers desire to produce counterexamples that expose incorrect behavior in the student's solution. For these cases, we describe techniques inspired from program analysis, and in particular, test input generation techniques [4, 34], often used for bug finding.

(a)



(b)

```
GCF(int array array $T$, int $I_1$, int $I_2$)
Assume $T[0,0], T[1,0]$ contain $I_1, I_2$ respectively.
for ($j := 0$; $T[2j, j] \neq 0$; $j := j+1$):
    $T[2j, j+2] :=$ Floor($T[2j, j+1] \div T[2j, j]$);
    $T[2j+1, j+1] := T[2j, j+2] \times T[2j, j]$;
    $T[2j+2, j+1] := T[2j, j+1] - T[2j+1, j+1]$;
    $T[2j+2, j+2] := T[2j, j]$;
return $T[2j, j+1]$;
```

Figure 2. Solution Generation for procedural problems [6]. (a) Demonstration of GCF procedure over inputs 762 and 1270 to produce output 254. (b) Procedure GCF synthesized automatically from the demonstration in (a).

Inside: Examples can also be used inside the underlying algorithms for performing inductive reasoning. This happens in case of both solution generation and problem generation for conceptual problems (§2.2, §3.2). This style of reasoning is inspired by how humans often approach problem generation and solving, and the underlying techniques are inspired by work in the area of establishing program correctness using random examples [15], and program synthesis using examples [16].

Next, we elaborate on example-based learning technologies by giving specific instances and highlighting general principles. This article is organized by the three key tasks in intelligent tutoring [33], namely solution generation (§2), problem generation (§3), and feedback generation (§4). We motivate each of these inter-related tasks, and give multiple instances of example-based learning technologies for each task. We also describe some evaluation associated with each of these instances. While several of these instances are in infancy, some of them have been deployed and evaluated relatively more thoroughly (§3.1, §4.2).

2. Solution Generation

Solution generation is the task of automatically generating solutions given a problem description in some subject domain. This is important for several reasons. First, it can be used to generate sample solutions for automatically generated problems. Second, given a student's incomplete solution, it can be used to complete the solution, which can be much more illustrative for a student compared to providing a completely different sample solution. Third, given a student's incomplete solution, it can also be used to generate hints on the next step or an intermediate goal.

2.1 Procedural Problems

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

Such procedures can be synthesized using PBE technology [10, 16, 22], which has traditionally been applied to end-user applications. Recently, PBE was used for synthesizing various kinds of spreadsheet tasks including string transformations and table layout transformations [18]. We observe that mathematical procedures can be viewed as spreadsheet procedures that involve both (a) computation of new values from existing values in spreadsheet cells (as in string transformations, which produce a new output string from substrings of input strings), and (b) positioning that value in an appropriate spreadsheet cell (as in table transformations, which reposition contents of input spreadsheet table). We have combined ideas from learning of string and table transformations to learn mathematical procedures from example traces, wherein a trace is a sequence of (value, cell) pairs [6]. We use dynamic programming to compute all sub-programs that are consistent with various sub-traces (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. Fig. 2 illustrates synthesis of a GCD procedure from an example trace, wherein the teacher-specified operators include $-$, $\times$, $\div$, and Floor.

2.2 Conceptual Problems

Solution generation for conceptual problems often requires performing search over the underlying solution space. Following are two complementary principles, each of which we have found useful across multiple subject domains. They also reflect how humans themselves might search for such solutions.

S1: **Perform reasoning over examples as opposed to abstract symbolic reasoning.** The idea here is to reason about the behavior of a solution on some/all examples (i.e., concrete inputs) instead of performing symbolic reasoning over an abstract input. This reduces search time by large constant factors because it is much faster to execute part of a construction/proof on concrete inputs than to reason symbolically about it.

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

We next illustrate these principles using multiple subject domains.

Geometry Constructions A geometry construction is a method for constructing a desired geometric object from other objects by applying a sequence of ruler and compass constructions (Fig. 3(e)). Such constructions are an important part of high-school geometry. The automated geometric theorem proving community (one of the successful stories of automated reasoning) has developed tools (e.g., Geometry Explorer [32] or Geometry Expert [14]) that allow students to create geometry constructions and use interactive provers to prove properties of those constructions. Below, we describe how to synthesize these constructions 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], wherein the goal is to synthesize a straight-line program (Fig. 3(d)) that realizes the relational specification between inputs and outputs (Fig. 3(b)).

The semantics of geometry operations is too complicated for use of symbolic methods for synthesis, or even, verification. We observe that ruler/compass operators are analytic functions, which implies that the validity of a geometry construction can be probabilistically inferred from testing on random examples. This 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 this random selection, $f(Y) \neq g(Y)$.*

Rule Name	Premises	Conc
Modus Ponens (MP)	p→q, p	q
Hypo. Syllogism (HS)	p→q,q→r	p→r
Disj. Syllogism (DS)	p∨q, ¬p	q
Simplification (Simp)	p∧q	q

(a)

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)

(b)

Rule Name	Proposition	Equivalent Proposition
Distribution	p ∨(q ∧r)	(p ∨q) ∧(p ∨r)
Double Negation	p	¬¬p
Implication	p →q	¬p ∨q
Equivalence	p ≡q	(p →q)∧(q →p)
	p ≡q	(p ∧q)∨(¬p ∧¬q)

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)

Figure 4. Solution Generation for natural deduction [1]. (a) Sample inference rules. (b) Sample replacement rules. (c) An abstract proof of the original problem in Fig. 7(b). The second column lists the 32 bit integer representation of the truth-table (over 5 variables). (d) A natural deduction proof of the original problem in Fig. 7(b) with inference rule applications shown in bold. (e) Natural deduction proof of a *similar problem* from Fig. 7(b) having same inference rule steps.

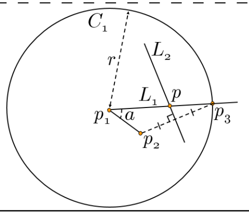
(a) English ⇓ Description	Construct a triangle given its base L (with end-points p_1,p_2), a base angle a,& sum r of the other two sides.
(b) PreCondition	$r > \texttt{Length}(p_1, p_2)$
⇓ PostCondition	$\texttt{Angle}(p, p_1, p_2) = a \quad \wedge$ $\texttt{Length}(p, p_1) + \texttt{Length}(p, p_2) = r$
(c) Random ⇓ Example	$L = \texttt{Line}(p_1 = \langle 81.62, 99.62\rangle, p_2 = \langle 99.62, 83.62\rangle)$ $r = 88.07$ $a = 0.81$ radians $p = \langle 131.72, 103.59\rangle$
(d) Geometry Program	$\texttt{ConstructTriangle}(p_1,p_2,L,r,a)$: L_1 := ConstructLineGivenAngleLinePoint(L,a,p_1); C_1 := ConstructCircleGivenPointLength(p_1,r); (p_3, p_4) := LineCircleIntersection(L_1,C_1); L_2 := PerpendicularBisector2Points(p_2,p_3); p := LineLineIntersection(L_1,L_2); return p;
(e) Geometry Construction	 C_1 r L_2 L_1 p p_1 a p_3 p_2

Figure 3. Solution Generation for geometry constructions [17].

Property 1 follows from the fact that non-zero analytic functions have isolated zeroes, i.e., for every zero point of an analytic function, there exists a neighborhood in which the function is non-zero. Thus, the number of non-zero points of the non-zero analytic function $f(X) - g(X)$ dominates the number of its zero points. [1]

Hence, the problem of synthesizing geometry constructions that satisfy a symbolic relational specification between inputs/outputs can be reduced to that of synthesizing constructions that are consistent with randomly chosen input-output *examples* (Principle S1). This forms the basis of our synthesis algorithm for geometry constructions [17], which involves the following two key steps (also illustrated in Fig. 3) reflecting the two general principles discussed above: (i) Generate random input-output examples (Fig. 3(c)) from the logical description (Fig. 3(b)) of the given problem using off-the-shelf numerical solvers. The logical description in turn is generated from the natural language description (Fig. 3(a)) using natural language translation technology. (ii) Perform brute-force search over a library of ruler/compass operators to find a construction (Fig. 3(d)) that transforms the randomly selected input(s) into corresponding output(s). The search is performed over an extended library of ruler/compass operators that includes higher-level primitives such as perpendicular/angular bisectors (Principle S2). This not only shortens a solution size (allowing for efficient search), but is also more readable for the student. On our benchmark of 25 problems [17], use of extended library shortened solutions to 2-13 steps from 3-45 steps and increased the success rate from 75% to 100%.

Natural Deduction Proofs Natural deduction (taught in introductory logic courses) 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 step derives a proposition, which is either a premise or is derived from preceding propositions by application of some inference rule (Fig. 4(a)) or replacement rule (Fig. 4(b)) and the last of which is the conclusion of the argument. Fig. 4(d) shows such a proof. Ditmarsc [29] presents a survey of proof assistants for teaching natural deduction (like Pandora [9]), some of which can also solve problems. We describe a different and a scalable way to solve such problems that also paves the way for generating fresh problems (§3.2).

While SAT/SMT solving and theorem proving communities [8] have focused on solving large-sized proof problems in a reasonable amount of time, our recent approach [1] to generating natural deduction proofs in real-time leverages the observation that classroom sized instances are small. Our approach involves the following two aspects reflecting the two general principles discussed above. (i) We abstract a proposition using its truth-table, which can be represented using a bitvector representation [20]–this avoids expensive symbolic reasoning and reduces application of inference rules to simple bitvector operations (Principle S1). (ii) We break the proof search into multiple smaller (and hence more efficient) proof searches (Principle S2). First, we search for an *abstract proof* that involves only inference rule applications over truth table representation (note that replacement rules are identity operations over truth table representation). Then, we refine this abstract proof over truth-table representation to a complete proof over symbolic propositions by searching for sequences of replacement rules in between every two neighboring inference rules. Fig. 4(c) gives an example of an abstract proof, while Fig. 4(d) shows its refinement to a complete proof. Note that the size of an abstract proof and the number of replacement rules in between any two inference rules is much smaller than the size of the overall proof. Our methodology solved 84% of 279 problems collected from various textbooks (generating proofs of ≤ 27 steps), while a baseline algorithm (that used symbolic representation for propositions and performed breadth-first search for the complete proof) solved 57% of these problems [1].

[1] However, unlike polynomial identity testing theorem [25], which allows performing modular arithmetic over numbers selected from a finite integer set for efficient evaluation, this result does not provide any constructive guidance on the size of the selection set and requires precise arithmetic. We approximate this process by using finite precision floating point arithmetic and using a threshold for comparing equality—this, in our practical experiments, did not yield any unsoundness or incompleteness.

(a)

```
Add(int array $A$, int array $B$)
  $\ell$ := Max(Len($A$), Len($B$));
  for $i$=0 to $\ell$-1                          ▷ Loop over digits (L)
    if ($i \geq$ Len($A$)) $t := B[i]$;           ▷ Different # of digits (D)
    else if ($i \geq$ Len($B$)) $t := A[i]$;      ▷ Different # of digits (D)
    else $t$:=$A[i]$+$B[i]$;
    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 ($C[\ell]$ == 1) $R[\ell] := 1$;             ▷ Extra digit in output (E)
```

(b)

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

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

3. Problem Generation

Generating fresh problems that have specific solution characteristics (such as a certain difficulty level and that involves using a certain set of concepts) is a tedious task for the teacher. Automating this has several benefits. Generating problems that are 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 provide instructors with a fresh source of problems to be used in their assignments or lecture notes. Second, it can help *prevent cheating* [23] in classrooms or MOOCs (with unsynchronized instruction) since each student can be provided with a different problem of the same difficulty level. Third, when a student fails to solve a problem, and ends up looking at the sample solution, then the student may be presented with another similar practice problem. Generating problems that have a given difficulty level and that exercise a given set of concepts can help *create personalized workflows* for students. If a student solves a problem correctly, then the student may be presented with a problem that is more difficult than the last problem, or exercises a richer set of concepts. If a student fails to solve a problem, then the student may be presented with simpler problems to identify any core concepts the student has not yet mastered and to reinforce those concepts.

On the other hand, fresh problems create new pedagogical challenges since teachers may no longer recognize those problems and students may not be able to discuss those problems with each other after assignment submission. These challenges may be mitigated by solution generation (§2) and feedback generation (§4) capabilities.

3.1 Procedural Problems

A procedural problem can be characterized by the trace that it generates through the corresponding procedure. Various features of this trace can be used to identify the difficulty level of a procedural problem and the various concepts that 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. A trace that executes more iterations of a loop might be more difficult than the one that executes fewer iterations.

The use of a trace-based framework allows for using test input generation tools [4] for generating problems that have certain trace features. We have used this methodology to automatically synthesize practice problems for elementary and middle school mathematics [5]. Fig. 5 illustrates this in the context of an addition procedure. Note that various addition concepts can be modeled as trace properties, and in particular, regular expressions over procedure locations. Furthermore, use of a trace-based framework allows for using notions of procedure coverage [34] to evaluate the comprehensiveness of a certain collection of expert-designed problems and to fill in 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, or whether or not the exceptional case of a conditional branch was executed, and the set of n-grams present in the trace. We used this partial order to synthesize *progressions* of problems, and even analyze and compare existing progressions across various textbooks.

Recently, we used our trace-based framework [5] to synthesize a progression of thousands of levels for Refraction, a popular math puzzle game. We conducted an A/B test with 2,377 players (on the portal www.newgrounds.com) that showed our synthesized progression motivated players to play for similar lengths of time as in case of the original expert-designed progression. The median player in the synthesized progression group played for 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 within end-user applications. Many modern user applications have advanced features, and learning these procedures constitutes a major effort on the part of the user. Therefore designers have focused their energy on trying to reduce this effort. For example, Dong et.al. created a series of mini games to teach users advanced image manipulation tasks in Adobe Photoshop [11]. Our methodology may assist in creating such tutorials/games by automatically generating progressions of tasks from procedural specifications of advanced tasks.

3.2 Conceptual Problems

Problem generation for several kinds of conceptual problems can be likened to discovering new theorems, which ought to be a search intensive activity that can be aided with domain-specific strategies. However, there are two general principles that we have found useful across multiple subject domains.

P1: **Example-based template generalization.** This involves generalizing a given example problem into a template and searching for all possible instantiations of that template for valid problems. Given that the search space might be huge, this methodology is usually applicable when it is possible to quickly check the validity of a given candidate problem. This methodology does not necessarily require access to a solution generation technology, though presence of a solution generation technology can guarantee the difficulty level of the generated problems.

P2: **Problem Generation as reverse of Solution Generation.** This applies to only proof problems. The idea here is to perform a reverse search in the solution search space starting from the goal and leading up to the premises. This methodology has the advantage of ensuring that the generated problems have specific solution characteristics.

We next illustrate these principles using multiple subject domains.

Algebraic Proof Problems Problems that require proving algebraic identities (Fig. 6) are common in high-school math curriculum. Generating such problems is a very tedious task for the teacher since the teacher can't simply arbitrarily change constants (unlike in procedural problems) or variables to generate a correct problem.

Example Problem $\Downarrow$	$\frac{\sin A}{1+\cos A} + \frac{1+\cos A}{\sin A} = 2\csc A$	$\begin{vmatrix} (x+y)^2 & zx & zy \\ zx & (y+z)^2 & xy \\ yz & xy & (z+x)^2 \end{vmatrix} = 2xyz(x+y+z)^3$
Generalized Problem Template $\Downarrow$	$\frac{T_1 A}{1 \pm T_2 A} + \frac{1 \pm T_3 A}{T_4 A} = 2\ T_5 A$ where $T_i \in \{\cos, \sin, \tan, \cot, \sec, \csc\}$	$\begin{vmatrix} F_0(x,y,z) & F_1(x,y,z) & F_2(x,y,z) \\ F_3(x,y,z) & F_4(x,y,z) & F_5(x,y,z) \\ F_6(x,y,z) & F_7(x,y,z) & F_8(x,y,z) \end{vmatrix} = c\ F_9(x,y,z)$ where $F_i (0 \le i \le 8)$ and F_9 are homogeneous polynomials of degrees 2 and 6 respectively, $\forall (i,j) \in \{(4,0),(8,4),(5,1),\ldots\} : F_i = F_j[x \to y; y \to z; z \to x]$, and $c \in \{\pm 1, \pm 2, \ldots, \pm 10\}$.
New Similar Problems	$\frac{\cos A}{1-\sin A} + \frac{1-\sin A}{\cos A} = 2\tan A$ $\frac{\cos A}{1+\sin A} + \frac{1+\sin A}{\cos A} = 2\sec A$ $\frac{\cot A}{1+\csc A} + \frac{1+\csc A}{\cot A} = 2\sec A$ $\frac{\tan A}{1+\sec A} + \frac{1+\sec A}{\tan A} = 2\csc A$ $\frac{\sin A}{1-\cos A} + \frac{1-\cos A}{\sin A} = 2\cot A$	$\begin{vmatrix} y^2 & x^2 & (y+x)^2 \\ (z+y)^2 & z^2 & y^2 \\ z^2 & (x+z)^2 & x^2 \end{vmatrix} = 2(xy+yz+zx)^3$ $\begin{vmatrix} -xy & yz+y^2 & yz+y^2 \\ zx+z^2 & -yz & zx+z^2 \\ xy+x^2 & xy+x^2 & -zx \end{vmatrix} = xyz(x+y+z)^3$ $\begin{vmatrix} yz+y^2 & xy & xy \\ yz & zx+z^2 & yz \\ zx & zx & xy+x^2 \end{vmatrix} = 4x^2y^2z^2$

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.

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

$\neg x_4$
$x_4 \equiv x_5$
$x_4 \equiv x_2$
$x_4 \to x_2$
$x_4 \to \neg x_1$

(a)

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

(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	Concl.
$(x_1 \to x_3) \to x_2$	$x_2 \to x_3$	$\neg x_3$	$x_1 \wedge \neg x_3$
$x_3 \to x_1$	$(x_3 \equiv x_1) \to x_2$	$\neg x_2$	$x_1 \wedge \neg x_3$
$(x_1 \equiv x_3) \vee (x_1 \equiv x_2)$	$(x_1 \equiv x_2) \to x_3$	$\neg x_3$	$x_1 \equiv x_3$
$x_1 \equiv \neg x_3$	$x_2 \vee x_1$	$x_3 \to \neg x_2$	$x_1 \wedge \neg x_3$
$x_3 \to x_1$	$x_1 \to (x_2 \wedge x_3)$	$x_3 \to \neg x_2$	$\neg x_3$

(c)

Figure 7. Various Problem Generation interfaces for natural deduction problems [1]. (a) Proposition replacement. (b) Similar problem generation. (c) Parameterized problem generation.

Our algebra problem generation methodology (illustrated in Fig. 6) uses Principle P1 to generate fresh problems that are similar to a given *example* problem [26]. First, a given example problem is generalized into a template that contains 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 either providing more example problems or by manually editing the initially generated template. Then, we automatically enumerate all possible instantiations of the template and check the validity of an instantiation by testing on random inputs. The probabilistic soundness of such a check follows from Property 1. This methodology works for identities over analytic functions, which can involve a variety of common algebraic operators including trigonometry, integration, differentiation, logarithm, exponentiation, etc. Note that this methodology would not be feasible if symbolic reasoning was used (instead of random testing) to check the validity of a candidate instantiation since it would be too slow to test out all instantiations in real time (Principle S1 in §2.2).

Natural Deduction Problems Fig. 7 illustrates three different interfaces to generating new natural deduction problems [1]. The proposition replacement interface (Fig. 7(a)) finds replacements for a given premise or the conclusion in a given *example* problem. It generates those propositions as replacements that ensure that the new problem is well-defined, i.e., one whose conclusion is implied by the premises, but not by any strict subset of those premises. This interface is based on Principle P1 wherein we check all possible small-sized 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 in §2.2). 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 that have a solution that uses exactly the same sequence of inference rules as is used by a solution of an *example* problem. Fig. 7(b) shows some automatically generated problems given an example problem. Fig. 4(e) shows a solution for the first new problem in Fig. 7(b). Observe that this solution uses exactly the same sequence of inference rules (shown in bold) as the solution for the original example problem, shown in Fig. 4(d). The parameterized problem generation interface finds problems that have specific features such as a given number of premises and variables, maximum size of propositions, and whose smallest proof involves a given number of steps and makes use of a given set of rules. Fig. 7(c) shows some automatically generated problems given some parameters. Both these interfaces find desired problems by performing a reverse search in the solution space (Principle P2) that is explored by the solution generation technology for natural deduction (§2.2). The similar problem generation interface further uses the solution template obtained from a solution of the example problem for search guidance (Principle P1).

4. Feedback Generation

Feedback generation may include several aspects: identifying whether or not the student's solution is incorrect, *why* is it incorrect, and *where* or *how* can it be fixed. A teacher might even want

```python
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)

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

(b)

```python
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)

```python
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)

```python
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)

Figure 8. Automated Grading of introductory programming problems [27]. (a) depicts a reference implementation (in Python) for the problem of computing a derivative of a polynomial. (b) describes some rewrite rules that capture common errors. (c), (d), and (e) denote three different student submissions along with the respective feedback that has been automatically generated.

to generate a *hint* in order to enable to student to identify and/or fix mistakes on their own. In examination settings, the teacher would even like to award a *numerical grade*.

Automating feedback generation is important for several reasons. 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 back to students. From the student's perspective, this is highly undesirable since by the time they receive their graded assignments, their motivation to learn from their mistakes might be lost because of the need to page in the desired context. Furthermore, maintaining grade consistency across students and graders is a difficult task. The same grader may award different scores to two very similar solutions, while different graders may award different scores to the same solution.

4.1 Procedural Problems

Generating feedback for procedural problems is relatively easy (compared to conceptual problems) since they almost 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 that students make in procedural problems is to employ a wrong algorithm. Van Lehn has identified over 100 bugs that students demonstrate in subtraction alone [30]. Ashlock has identified a set of buggy computational patterns for a variety of algorithms based on real student data [7]. For instance, following are some bugs that Ashlock describes for the addition procedure (Fig. 5(a)).

- Add each column and write the sum below each column, even if it is greater than nine (page 34 in [7]).
- Add each column, from left to right. If the sum is greater than nine, write the tens digit beneath the column and the ones digit above the column to the right (page 35 in [7]).

We observe that all such bugs have a clear procedural meaning and can be captured as a procedure. These *buggy procedures* can be automatically synthesized from *examples* of incorrect student traces by using the same PBE technology referred to in §2.1. In fact, each of the 40 bugs that Ashlock describes in [7] is illustrated using a set of 5-8 example traces, and we were able to synthesize 28 (out of 40) buggy procedures from their example traces [6].

Identifying such *buggy procedures* has multiple benefits. It can inform teachers about the misconceptions that a student has. It can also be used to automatically generate a progression of problems (as in §3.1) that are specifically tailored to highlighting differences between the correct procedure and the buggy procedure.

Aleven et.al. [2] also 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, it allows teachers to add annotations for providing feedback to students who get stuck or start following a known incorrect path.

4.2 Conceptual Problems

Feedback for proof problems can be generated by checking correctness of each individual step (assuming that the student is using a correct proof methodology) and using a solution generation technology to generate proof completions from the onset of any incorrect step [13]. In this section, we focus on feedback generation for construction problems. Following are two general principles each of which we have found useful across multiple subject domains.

F1: **Edit distance.** The idea here is to find the smallest set of edits to the student's solution that will transform it into a correct solution. Such a feedback informs the student about *where* the error is in their solution and *how* can it 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 solution—this captures a common mistake of misunderstanding the problem description. Such a feedback can inform the student about *why* their solution is incorrect. The number and type of edits can be used as a criterion for awarding numerical grades.

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

Next, we illustrate these principles using different subject domains.

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

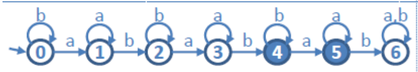
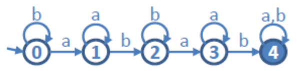
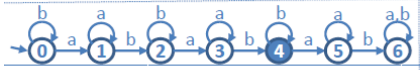
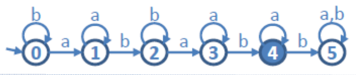
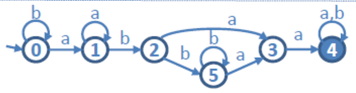
DFA Attempt	Feedback and Grade
	Accepts the correct language **Grade: 10/10**
	Accepts the strings that contain `ab' at least twice instead of exactly twice **Grade: 5/10**
	Misses the final state 5 **Grade: 9/10**
	Behaves correctly on most of the strings **Grade: 6/10**
	Accepts the strings that contain `ab' at least twice instead of exactly twice **Grade: 5/10**

Figure 9. Automated Grading of automata problems [3]. The figure shows several attempts to the problem of constructing an automata that accepts strings containing the substring "ab" exactly twice. Also shown is automatically generated feedback & grade.

who find it difficult to map counterexamples to errors in their code. We describe below an edit distance based technique [27] that provides guidance on *how* to fix an incorrect solution (Principle F1).

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

Fig. 8(c-e) show three student programs, together with the respective feedback generated by our program grading tool [27]. The underlying technique involves exploring the space of all candidate programs, based on applying the teacher provided rewrite rules to the student program, to synthesize a candidate program that is equivalent to the reference solution and that requires minimum number of corrections. For this purpose, we leverage SKETCH [28], a state-of-the-art program synthesizer that employs a SAT-based algorithm to complete program sketches (programs with holes) so that they meet a given specification. We evaluated our tool on thousands of real student attempts (to programming problems) obtained from the Introduction to Programming course at MIT (6.00) and MITx (6.00x) [27]. Our tool successfully generated feedback (of up to 4 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 pre-existing knowledge base to provide feedback. While our technique does not make any assumption on the algorithms or plans that students can use, a key limitation is that it cannot provide feedback on student attempts that have big conceptual errors, which cannot be fixed by application of local rewrite rules. Furthermore, our technique 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 CS Education. JFLAP [24] is a widely used system for teaching automata and formal languages. It allows for constructing, testing, and conversion between computational models, but does not support grading. We discuss below 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$\}$. Fig. 9 shows five attempts submitted by different students and the respective feedback generated by our automata grading tool [3]. The underlying technique involves identifying different kinds of feedback including edit distance over both solution/problem (Principle F1) and *counterexamples* (Principle F2). Each feedback is associated with a numerical grade. The feedback that corresponds to the best numerical grade is then reported. 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 transform the transform the automata into a correct one. 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, namely $L' = \{s \mid s$ contains the substring "ab" at least twice$\}$, possibly reflecting the common mistake of misreading the problem description. Hence 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 remaining fourth attempt, which does not entertain a small edit distance, is based on counterexamples. The grade here is a function of the density of counterexamples with more weightage given to smaller-sized counterexamples since the student ought to have checked the correctness of their construction on smaller strings.

To automatically generate the above-mentioned feedback, we formalize problem descriptions using a logic that we call MOSEL, which is an extension of the classical monadic-second order logic (MSO) with some syntactic sugar that allows defining regular languages in a concise and natural way. In MOSEL, the languages L and L' can be described by the formulas $|\texttt{indOf}(ab)| = 2$ and $|\texttt{indOf}(ab)| \geq 2$ resp., where the `indOf` constructor returns the set of all indices where the argument string occurs. The automata grader tool implements synthesis algorithms that translate MOSEL descriptions into automata and vice versa. The MOSEL-to-automata synthesizer rewrites MOSEL descriptions into MSO and 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 automaton distance or tree distance (in case of problem descriptions), while counterexamples are computed using automata difference.

We evaluated the automata grader tool on 800+ student attempts to several problems from an automata course CS373 at UIUC [3]. For each problem we had two instructors and the tool grade each attempt. For one of these representative problems, we observed that (i) for 20% attempts, the instructors were incorrect (gave full marks to an incorrect attempt) or inconsistent (same instructor gave different marks to syntactically equivalent attempts). (ii) for 25% attempts, there was at least 3/10 point discrepancy between the tool and one of the instructors; and in more than 60% of those cases, the instructor concluded after re-reviewing that the grade of the tool was more fair. We also observed that there was more agreement between the tool and any of the instructors than between the two instructors. The instructors thus concluded that the tool should be preferred over humans for consistency & scalability.

The automata grading tool has been deployed online [3]; it provides live feedback and various kinds of hints. In a recent user study we observed that these hints were viewed as helpful, increased student perseverance, and improved problem completion time.

5. Conclusion

Providing personalized and interactive education (as in 1:1 tutoring) remains an unsolved problem for standard classrooms. The arrival of MOOCs, while providing a unique opportunity to share quality instruction with massive number of students, exacerbate this problem with an even higher student to teacher ratio. Recent advances in various computer science areas can be brought together to rethink intelligent tutoring [33], and the phenomenal rise of online education makes this investment very timely.

This article summarizes some recently published work from different computer science areas (Programming languages [17, 27], Artificial intelligence [1, 3, 26], and Human computer interaction [5]). It also exposes a common thread in this inter-disciplinary line of work, namely use of examples either 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). We hope that this illustration shall enable researchers to apply these principles to develop similar techniques for other subject domains. We also hope that this article shall inform educators about new advances that can assist with various educational activities and shall allow them to think more creatively about both curriculum and pedagogical reforms. For example, these advances can enable development of gaming layers that can take computational thinking down to K-12.

This article is focused on a rather technical perspective to computer-aided Education. While these technologies can help impact education in a positive manner, we ought to devise ways to quantify its benefits on student learning—this may be critical to attract funding. Furthermore, this article only discusses logical reasoning based techniques. These techniques can be augmented with complementary techniques that leverage large amounts of student populations and student data, whose availability has been facilitated by recent interest in online education platforms like Khan Academy and MOOCs. For example, we can leverage large amounts of student data to collect different correct solutions to a (proof) problem and use them to generate feedback [13] or to discover effective learning pathways to guide problem selection. We can leverage large student populations to crowdsource tasks that are difficult to automate [31] as is done in peer grading [21]. A synergistic combination of logical reasoning, machine learning, and crowdsourcing methods may lead to self-improving and advanced intelligent tutoring systems that shall revolutionize education.

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

References

[1] U. Ahmed, S. Gulwani, and A. Karkare. Automatically generating problems and solutions for natural deduction. In *IJCAI*, 2013.

[2] V. Aleven, B. M. McLaren, J. Sewall, and K. R. Koedinger. A new paradigm for intelligent tutoring systems: Example-tracing tutors. *Artificial Intelligence in Education*, 19(2), 2009.

[3] R. Alur, L. D'Antoni, S. Gulwani, D. Kini, and M. Viswanathan. Automated grading of DFA constructions. In *IJCAI*, 2013. Tool accessible at `http://www.automatatutor.com/`.

[4] S. Anand, E. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn. An orchestrated survey on automated software test case generation. *Journal of Systems and Software*, 86(8), 2013.

[5] E. Andersen, S. Gulwani, and Z. Popovic. A trace-based framework for analyzing & synthesizing educational progressions. In *CHI*, 2013.

[6] E. Andersen, S. Gulwani, and Z. Popovic. Programming by demonstration applied to procedural math problems. Technical report, 2013.

[7] R. Ashlock. *Error Patterns in Computation: A Semi-Programmed Approach*. Merrill Publishing Company, 1986.

[8] N. Bjørner. Taking satisfiability to next level with Z3. In *IJCAR*, 2012.

[9] K. Broda, J. Ma, G. Sinnadurai, and A. J. Summers. Pandora: A reasoning toolbox using natural deduction style. *IGPL*, 15(4), 2007.

[10] A. Cypher, editor. *Watch What I Do: Programming by Demonstration*. MIT Press, 1993.

[11] T. Dong, M. Dontcheva, D. Joseph, K. Karahalios, M. Newman, and M. Ackerman. Discovery-based games for learning software. In *CHI*, 2012.

[12] C. Douce, D. Livingstone, and J. Orwell. Automatic test-based assessment of programming: A review. *Edu. Resources in Comp.*, 2005.

[13] E. Fast, C. Lee, A. Aiken, M. S. Bernstein, D. Koller, and E. Smith. Crowd-scale interactive formal reasoning & analytics. In *UIST*, 2013.

[14] X.-S. Gao and Q. Lin. MMP/Geometer-a software package for automated geometric reasoning. In *Aut. Deduction in Geometry*. 2004.

[15] S. Gulwani. *Program Analysis Using Random Interpretation*. PhD thesis, UC-Berkeley, 2005. ACM SIGPLAN Dissertation Award.

[16] S. Gulwani. Synthesis from examples: Interaction models and algorithms. In *SYNASC*, 2012. Invited talk paper.

[17] S. Gulwani, V. A. Korthikanti, and A. Tiwari. Synthesizing geometry constructions. In *PLDI*, pages 50–61, 2011.

[18] S. Gulwani, W. Harris, and R. Singh. Spreadsheet data manipulation using examples. *Communications of the ACM*, Aug 2012. http://research.microsoft.com/users/sumitg/flashfill.html.

[19] W. Johnson. *Intention-based Diagnosis of Novice Programming Errors*. Morgan Kaufmann, 1986.

[20] D. E. Knuth. *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1*. Addison-Wesley Professional, 2011.

[21] C. Kulkarni, K. Pang, H. Le, D. Chia, K. Papadopoulos, J. Cheng, D. Koller, and S. Klemmer. Peer and self assessment in massive online design classes. *ACM TOCHI*, 2013.

[22] H. Lieberman. *Your Wish Is My Command: Programming by Example*. Morgan Kaufmann, 2001.

[23] M. Mozgovoy, T. Kakkonen, and G. Cosma. Automatic student plagiarism detection : future perspectives. *Edu. Comp. Res.*, 43(4), 2010.

[24] S. Rodger and T. Finley. JFLAP-An Interactive Formal Languages and Automata Package. 2006.

[25] J. T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM*, 27(4):701–717, 1980.

[26] R. Singh, S. Gulwani, and S. Rajamani. Automatically generating algebra problems. In *AAAI*, 2012.

[27] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated feedback generation for introductory programming assignments. In *PLDI*, 2013.

[28] A. Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, UC Berkeley, 2008.

[29] H. Van Ditmarsch. User interfaces in natural deduction programs. *User Interfaces*, 98:87, 1998.

[30] K. VanLehn. *Mind Bugs: The Origins of Procedural Misconceptions*. MIT Press, 1991.

[31] D. S. Weld, E. Adar, L. Chilton, R. Hoffmann, E. Horvitz, M. Koch, J. Landay, C. H. Lin, and M. Mausam. Personalized online education—a crowdsourcing challenge. In *Workshops at AAAI*, 2012.

[32] S. Wilson and J. D. Fleuriot. Combining dynamic geometry, automated geometry theorem proving & diagram. proofs. In *UITP*, 2005.

[33] B. Woolf. *Building Intelligent Interactive Tutors*. Morgan Kaufman, 2009. ISBN 978-0-12-373594-2.

[34] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, Dec. 1997.