A Trace-based Framework for Analyzing and Synthesizing Educational Progressions

Erik Andersen[1], Sumit Gulwani[2], and Zoran Popović[1]

[1]Center for Game Science
Computer Science & Engineering
University of Washington
{eland,zoran}@cs.washington.edu

[2]Microsoft Research
Redmond, WA
sumitg@microsoft.com

A key challenge in teaching a procedural skill is finding an effective progression of example problems that the learner can solve in order to internalize the procedure. In many learning domains, generation of such problems is typically done by hand and there are few tools to help automate this process. We reduce this effort by borrowing ideas from test input generation in software engineering. We show how we can use execution traces as a framework for abstracting the characteristics of a given procedure and defining a partial ordering that reflects the relative difficulty of two traces. We also show how we can use this framework to analyze the completeness of expert-designed progressions and fill in holes. Furthermore, we demonstrate how our framework can automatically synthesize new problems by generating large sets of problems for elementary and middle school mathematics and synthesizing hundreds of levels for a popular algebra-learning game. We present the results of a user study with this game confirming that our partial ordering can predict user evaluation of procedural difficulty better than baseline methods.

Author Keywords
education; problem generation; execution traces; games

ACM Classification Keywords
H.5.0 Information interfaces and presentation: General

INTRODUCTION
One of the most important domains of human learning is procedural task learning, which spans a wide range of human activities. Humans learn to execute procedures that range from a simple list of actions, such as a cooking recipe, to more complex procedures involving loops and conditionals, such as prime factorization, long division, and solving systems of equations. The standard human practice of learning such procedures is by solving a sequence of training problems. This sequence of problems allows a learner to develop an internal model of the procedural algorithm over time, so that ultimately it can be applied correctly to all possible inputs for that procedure.

Procedural learning has been studied in HCI as part of software learnability [13, 23, 26]. Many study designs evaluate application usability by measuring whether a user can successfully execute a target procedure in that application. HCI researchers frequently wish to evaluate the degree to which a user interface design facilitates the learning of such procedures.

A fundamental problem of teaching procedural tasks in both HCI and education is determining the optimal sequence of training problems. Textbooks for elementary and middle school mathematics typically start with problems that only require a few steps to solve and grow to more complex multi-step problems that vary based on the input. These progressions often vary widely and many of them are likely suboptimal. The quality of a training sequence depends on many factors, such as the structure of the target procedure, cognitive processes that lead to the creation of procedural models in the mind, level of engagement towards the task, learner background, and learning preferences.

There are a number of guiding principles for learning progressions. Reigeluth and Stein's Elaboration Theory [24] argues that the simplest version of a task should be taught first, followed by progressively more complex tasks that elaborate on the original task. Csikszentmihalyi's theory of flow [7] suggests that we can keep the learner in a state of maximal engagement by continually increasing difficulty to match the learner's increasing skill. By considering Vygotsky's zone of proximal development [33], we can avoid overloading the learner by introducing so many concepts at the same time that the learner cannot create a consistent internal representation. Nevertheless, many important details of optimal progression design are not covered by general principles. It has been estimated that 200-300 hours of expert development are necessary to produce one hour of content for intelligent tutors [2], of which problem ordering is a key part.

In this paper, we create a framework for reasoning about the space of possible progressions as defined by the procedural task itself. Our goal is to create a representation of the space of progressions using only a specification of the algorithmic procedure, defined directly as a computer program. We propose categorizing a procedural task based on features of the program trace obtained by executing the procedure on that task. We show how this trace-based measure can be used to measure the quality of a progression and compare the relative difficulty of two problems.

The use of a trace-based framework for characterizing procedural tasks allows us to borrow well-established techniques from the software engineering community related to software testing. In particular, it allows us to use test input generation tools [31] for generating problems that have certain trace features. It also allows us to use notions of procedure coverage [34] to evaluate the comprehensiveness of a certain progression. Furthermore, we are able to borrow techniques from the sequence comparison literature to compare two different problems, in particular n-gram models [35].

We demonstrate our method by analyzing the space of progressions for two different domains. One domain is early math procedures, such as addition, adding fractions, and comparing integers. The other domain is a well-known interactive math puzzle game. We show how we can generate large sets of practice problems for these procedures. We also show how our framework can be used to analyze and compare expert-designed progressions in terms of thoroughness and aggressiveness. Determining which kinds of progressions are more effective is beyond the scope of this paper. Our goal is to automatically discover the coverage achieved by a progression, and generate additional problems to supplement the areas that it covers sparsely.

To assess our partial ordering measures of problem difficulty, we conducted a pilot user study in which we generated several levels for an algebra-learning game and asked participants to compare these levels to those in the expert-designed progression. We found that our model was better able to predict participant responses than other baseline metrics.

We believe that our framework contributes to HCI by using procedure analysis to reason about the space of possible problems and how they relate to learning. We believe that this will reduce the effort required to create such problem sets. We intend to comprehensively study variations in progressions using our framework, and we hope that other researchers can use the same formalism to discover general principles of problem progressions for procedural tasks.

RELATED WORK

Usability and Learnability in HCI
Effective learning progressions are important not just for school-based learning; they are a key cornerstone of usability and learnability within 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 minigames to teach users advanced image manipulation tasks in Adobe Photoshop [9]. Grabler et al. created a system that allows a user to program a tutorial for image-manipulation tasks through demonstration in GIMP and Adobe Photoshop [12]. We extend these efforts by proposing a basic theory of increasing procedural complexity that may someday reduce the effort of creating such tutorials and games. We envision a methodology in which designers specify a procedure and receive an automatically generated set of tasks that are organized into a progression.

There has also been work in HCI on sociocultural bases for learning. For example, Rieman [25] examined how humans learn to perform procedural tasks through exploratory learning. Linehan et al. [19] provide a series of guidelines for designing effective educational games. Andersen et al. showed that video game tutorials are only effective when the game is too complex for players to learn through experimentation [5] and that changing a progression by introducing secondary game objectives can negatively impact engagement and retention [3, 4]. We attempt to supplement such perspectives with pragmatic approaches to designing learning systems.

Education and Intelligent Tutors
McArthur et al. [21] automatically generate algebra problems as part of an intelligent tutor for algebra. They first collect and annotate problems by what skills they use, such as isolating positive terms, and multiplying both sides of an equation by -1. They then indicate relationships between these skills, such as that one is a prerequisite for another, or that one is a generalization of another. Since each of these skills is essentially a production rule, they specify a grammar for how these rules can transform a problem state into another problem state, and use random values to generate problems. Sleeman [28] used a similar approach of specifying a grammar for algebra problems and used this to generate new problems. We expand on this work by using the idea of procedure traces to automate the identification of relationships between problems and generalize it to a larger class of problem domains. We also show how these ideas allow us not only to generate problems and progressions but also to evaluate existing progressions.

Li et al. studied problem orderings by examining them with a machine learning agent called SimStudent. They found that interleaved problem orderings led to faster learning than blocked orderings [18]. We extend this work by creating a framework in which interleaved or blocked problem progressions can be created for any procedural task. VanLehn [32] has extensively studied student acquisition of the subtraction procedure and analyzed the bugs that children display while learning this procedure. We focus on a larger domain of procedural tasks and show how these procedures can be used to generate practice problems.

Gulwani [14, 15] has applied techniques from formal methods, and in particular program synthesis, to various aspects of intelligent tutoring systems including solution generation [16], problem generation [27], automated grading, and content entry. In this work, we use test input generation techniques from formal methods for problem generation.

Problem Generation
There have been two approaches to generating problems for procedural mathematical content. In one approach, flexibility is provided for instantiating parameters of a problem with random constants [17]. In contrast, for our domain of middle school procedural problems, the choice of the constants makes all the difference in the difficulty level of a problem.

In another approach, certain features of the problem domain are provided as hard-coded options and users are able to

choose among these options and generate problems. For instance, in the domain of quadratic equations, some interesting features could be whether the equation is "simple factorable", "difficult factorable, where the leading coefficient is not 1", or "requires use of general quadratic formula". Another interesting feature can be whether or not it has imaginary solutions. Several math worksheet generator websites are based on this approach. The Microsoft Math-Worksheet Generator goes a step ahead and automatically infers such features from a problem instance [22]. Each domain has its own set of features that needs to be defined separately. Our system can infer such features from a problem instance to generate similar problems. However, more significantly, a procedural description of the problem domain leads to automatic definition of such features. Furthermore, our system considers more complicated features, such as n-grams, that relate to a sequence of decisions required to solve a problem as opposed to a single decision.

Recently, there has been some progress in generating problems in non-procedural mathematical domains. Singh et al. [27] proposed a semi-automatic template-based approach to problem generation for the domain of high-school algebra proof problems. The teacher semi-automatically generalizes a given seed problem into an abstract template. However, not all instantiations of those templates are valid proof problems. The underlying system performs a brute-force enumeration over all possible instantiations and uses novel results from randomized algebraic identity testing [16] to filter those instantiations that yield valid problems. They can generate impressive and non-trivial problems; however, there is no guarantee of the difficulty level associated with each problem.

Cerny et. al. [6] also proposed a template-based approach to problem generation for the domain of automata theory problems. A given seed problem is abstracted into a template, and most instantiations, if not all, are actually correct problems. The system then uses a sophisticated solution generation technology based on novel results in automata theory to compute the solutions for various instantiations and then partitions the problems into different equivalence classes based on some user-defined similarity metric. However, no attempt is made to compare the problems across different equivalence classes or different seed problems.

Procedural Content Generation in Games
Smith et al. used answer-set programming to generate levels for the educational puzzle game Refraction (Center for Game Science 2010) that adhered to prespecified constraints written in first-order logic [29]. Similar approaches have also been used to generate levels for platform games [30]. In the majority of existing approaches, designers must explicitly specify constraints that the generated content must reflect, for example, "the tree needs to be near the rock and the river needs to be near the tree". There has been much less work in guiding the generation based on a description of what the player needs to do. Dormans generated levels for puzzle-platform games [10] through grammars that could build a "mission" for the desired tasks the player needed to perform. We expand on these methodologies by generating not only levels but also

level progressions directly from a procedure that solves those levels. Although previous work has typically considered level solvers and level generators to be separate entities, our framework unifies them.

A TRACE-BASED FRAMEWORK
We present a trace-based framework for characterizing practice problems and estimating their relative difficulty level. For example, consider the standard addition algorithm for adding any number of positive integers:

Algorithm 1 Addition: Given as input a sequence of sequences of digits $n_i = [n_{i_q}, ..., n_{i_0}]$, add them:

```
 1: procedure ADD(n₁, ..., nₘ)
 2:     a ← 0
 3:     maxLen ← max(len(n₁), ..., len(nₘ))
 4:     for i ← 0, maxLen − 1 do        ▷ Loop over digits (D)
 5:         k ← 0
 6:         for all nⱼ do                ▷ Loop over inputs (N)
 7:             if len(nⱼ) > i then
 8:                 k ← k + nⱼ[i]         ▷ If digit exists (A)
 9:             end if
10:         end for
11:         if c[i] ≠ null then
12:             k ← k + c[i]              ▷ If carry exists (C)
13:         end if
14:         if len(k) = 2 then
15:             c[i + 1] ← k[1]           ▷ If we need to carry (O)
16:         end if
17:         a[i] ← k[0]
18:     end for
19:     if c[maxLen] = 1 then
20:         a[maxLen] ← c[maxLen]         ▷ If final carry (F)
21:     end if
22:     return a
23: end procedure
```

We believe that one of the best categorizations of the difficulty of a particular addition problem is the pathway or the trace that the procedure takes while solving that problem. A quick analysis of addition problems in any textbook will likely show many such traces. We use sequences of letters to indicate traces; these letters are output when the program executes commented lines in the above procedure.

Valid inputs to this problem have two or more input numbers and these numbers can have any number of digits. The simplest possible trace under these constraints is the trace corresponding to one-digit number plus a one-digit number where the sum is less than 10: "DNANA". In this trace, the carry branch "C" does not execute because there is no existing carry, the "O" overflow branch does not execute because there is no overflow, and there is no final carry so the final carry branch "F" also does not execute. The digit existence branch must execute because otherwise the addend would be invalid.

The following table shows several problems and their traces:

Problem	Trace
$1 + 1$	DNANA
$11 + 1$	DNANADNAN
$1 + 11$	DNANADNNA
$11 + 11$	DNANADNANA
$9 + 1$	DNANAOF
$19 + 1$	DNANAODNANC
$1 + 2 + 3$	DNANANA
$1 + 2 + 3 + 7$	DNANANANAOF
$333 + 444$	DNANADNANADNANA

Characterizing problems by their execution traces has several implications. It allows us to measure the program coverage of a problem, which can be used to evaluate the comprehensiveness of a progression. It allows us to compare problems, a notion more formally captured in the next section, based on how the procedure executes. It does *not* take into account differences in perceived difficulty that may arise from differences in the input values. For example, students may be much more comfortable with adding $1 + 1$ than $5 + 4$, and our model will not take this into account. However, such differences are generally domain-specific and cannot be identified without gathering data. Our model can easily grow to accommodate such distinctions. If a teacher or designer wants to distinguish two classes of inputs, he or she can add a conditional branch to the program that will separate these two classes.

PARTIAL ORDERINGS OVER TRACES

In order to assemble practice problems into progressions that begin simply and grow more difficult, we define a methodology for comparing two different problems. To do this, we define partial orderings over problems based on features of their corresponding traces. We can use this partial order relationship to compare two problems and determine if they are similar in difficulty, if one problem is more difficult than the other, or if no determination can be made using our model. Once we have such a partial ordering, we can identify simpler or more complex variations of a particular problem, and we can use it to sequence multiple problems together into a progression.

There are many possible ways to define a partial ordering over execution traces. Here, we define two partial orderings that we have found useful in analyzing and synthesizing progressions.

Path-based Partial Ordering

We use the following trace features to define our first partial order over traces:

- the number of times a certain loop in the procedure was executed

- whether or not the non-skip branch of a certain exceptional conditional was executed; we define a conditional to be *exceptional* if one of its branches contains no instructions and is equivalent to "skip".

We now use the following recursive definition to define a partial ordering:

Definition 1a: *A trace T_1 that contains loops is as least as complex as trace T_2 if for every loop L in the procedure, the trace T_1 has at least as many iterations of loop L as in trace T_2, and for each subtrace T_2' in T_2 that corresponds to an iteration of loop L, there exists a subtrace T_1' in T_1 that is at least as complex as T_2'.*

We also define the following base case:

Definition 1b: *A non-loopy trace T_1 is at least as complex as trace T_2 if for every exceptional condition in the procedure, if the non-skip branch of the exceptional condition was executed in T_2, then it was also executed in T_1.*

We now give some examples of trace comparisons based on the above-defined partial order.

Example 1: *Loops that execute longer are more complex.* If we compare the traces "DNANA" and "DNANANA", corresponding to the problems $1 + 2$ and $1 + 2 + 3$, respectively, the only difference is that the inner "N" loop executed one more time. Therefore, "DNANANA" subsumes "DNANA".

Example 2: *A conditional branch that executes some statements is more complex than a conditional branch that executes no statements.* If we compare "DNANA" and "DNANAOF", corresponding to the problems $1 + 2$ and $9+1$, respectively, "DNANAOF" contains all of the "work" of "DNANA", plus the additional step that there is an overflow and the 1 must be carried over to the next digit. Therefore, "DNANAOF" also subsumes "DNANA".

Example 3: Since loops can contain conditionals and other loops, each iteration of a loop that execute multiple times may vary in complexity. Our partial order also allows us to compare such traces. For example, consider the two traces "DNANADNANA" and "DNANADNAN", corresponding to the problems $11 + 11$ and $11 + 1$, respectively. In both cases, the outer "D" loop executes twice. The first iteration of this loop is the same but the second iteration is different. The second iteration for "DNANADNANA" executes a final "A" statement in an exceptional condition that "DNANADNAN" skips. Therefore "DNANADNANA" subsumes "DNANAD-NAN".

The notion of partial order also allows us to take a given trace and build more complex variants. For example, consider again the trace "DNANA" from the above addition example. There are three obvious ways in which this trace can expand. The first is by flipping conditional branches that executed no statements so that they do execute statements. One such trace is "DNANAOF", corresponding to problems in which there is a carry such as $1 + 9$. Another possible expansion is to increase the number of iterations of the "N" loop by 1, yielding the trace "DNANANA" that corresponds to problems with three one-digit numbers and no carries like $1+2+3$. Another way to expand the problem is to increase the "D" loop by one iteration, perhaps yielding the trace "DNANADNANA" corresponding to two-digit addition problems with no carries like $12 + 34$.
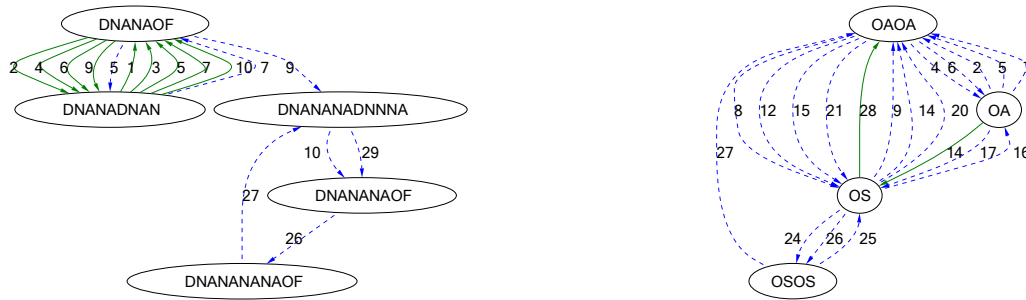
Figure 1. We can use our trace-based framework to compare progressions. The Singapore Math Sprints books organize worksheets into pairs of "A" worksheets and "B" worksheets, stating that the B side is "intended for more advanced students". The above figures compare the "A" and "B" progressions for two different problem domains. In both figures, the green solid arrows correspond to the "A" progression, and the blue dashed arrows correspond to the "B" progression. Self-edges are removed for clarity. The left side shows a worksheet pair for addition, corresponding to Algorithm 1. The right figure shows a different worksheet pair for fraction computation, corresponding to Algorithm 2. In both cases, we see that the "advanced" progression spends less time in the easier regions and quickly moves into longer pathways that require additional steps. In the case of addition, the "B" problems have more addends, and in the case of fraction computation, the "B" problems have more input fractions.

N-gram-based Partial Ordering

We now define a second partial ordering by utilizing n-grams [35]. This method defines a family of partial orderings depending on the value of positive integer n. n-gram models have been used extensively in natural language processing and search engines, where the meaning of a word can be uniquely defined by just looking at a small context around it in the parent sentence. In the case of procedural execution traces, the intuition is that students can only remember a certain amount of context, and it is most useful to test students on their ability to execute small sequences of algorithmic decisions within a larger execution trace.

The n-gram abstraction of a trace also provides a nice continuum between the standard notions of *statement coverage*, which corresponds to a uni-gram model, and *path coverage*, which corresponds to ∞-gram model in the software engineering literature [34].

Definition 2: *Let n be any positive integer. We say that a trace T_1 is at least as complex as trace T_2 if every n-gram of trace T_2 is also present in trace T_1.*

ANALYSIS OF PROGRESSIONS

In order to test the ability of our framework to analyze progressions, we gathered problems from three different workbooks for elementary and middle school mathematics:

- Math Sprints series from Singapore Math Inc.[1] These books include many worksheets that are intended to be completed in rapid one-minute sessions. They tend to involve lots of repetition and not too much difficulty.

- JUMP Math curriculum[2]. This Canadian curriculum was able to raise at-grade-level performance from 12% to 60% in a study in Lambeth, England [1].

- Skill Sharpeners: Math[3] by Evan-Moor Educational Publishers.

Evaluating Progressions

We can use our framework to explore interesting patterns in a progression. We first define an algorithm for performing multiple addition and subtraction operations on fractions:

Algorithm 2 Fraction Computation: Given as input a set of fractions $f_1, ..., f_n$ and a set of $+$ and $-$ operations $o_1, ..., o_{n-1}$, execute these operations:

1: **procedure** FCOMPUTE($f_1, ..., f_n, o_1, ..., o_{n-1}$)
2:      **if** $\exists f_i, f_j : f_i.d \neq f_j.d$ **then**
3:          $lcm \leftarrow max(d \in f_i.d)$ ▷ Denom. diff. (D)
4:          **while** $\exists f_i : lcm \bmod f_i.d \neq 0$ **do**
5:              $lcm \leftarrow lcm + max(d \in f_i.d)$ ▷ (M)
6:          **end while**
7:          **for all** f_i **do**
8:              $s \leftarrow lcm \div f_i.d$
9:              $f_i.n \leftarrow f_i.n * s$
10:             $f_i.d \leftarrow f_i.d * s$
11:         **end for**
12:     **end if**
13:     $n \leftarrow 0$
14:     **for all** o_i **do**   ▷ For each operation (O)
15:         **if** $o_i = +$ **then**
16:             $n \leftarrow n + f_{i+1}.n$ ▷ Add (A)
17:         **else if** $o_i = -$ **then**
18:             $n \leftarrow n - f_{i+1}.n$ ▷ Subtract (S)
19:         **end if**
20:     **end for**
21: **end procedure**

Figure 1 shows progressions from the Singapore Math Sprints workbooks for Algorithms 1 and 2. These worksheets are organized into pairs of "A" and "B" worksheets. The instructions state that the "A" progression is intended for weaker students and the "B" progression is intended for stronger students. For most of these pairs, the two worksheets cover the same set of concepts. However, we can see that the "A" progressions spend more time moving back and forth between simple traces and the "B" progressions move into more complicated traces that subsume the early traces.

Figure 2 shows a larger-scale analysis of progressions from all of these books. For each progression that we chose to analyze in each book, we first categorized all of the problems by trace. We can see that for most of these problem types, the number of traces is much less than the number of problems. However, there are differences between progressions; for example, the Singapore Sprints progression for addition has many more unique traces than the Skill Sharpeners progression.

Making progressions more systematic

We can use our framework to identify pathways a progression does not cover and suggest problems to fill that gap, if desired. We first define an algorithm for comparing two integers:

Algorithm 3 Integer Comparison: Given as input two sequences of digits $a = [a_0, ..., a_m]$ and $b = [b_0, ..., b_n]$, determine if $a > b$, $a < b$, or $a = b$:

```
 1: procedure COMPARE(a, b)
 2:     if len(a) > len(b) then
 3:         return more              ▷ More digits (H)
 4:     else if len(a) < len(b) then
 5:         return less              ▷ Fewer digits (L)
 6:     end if
 7:     for i ← 0, len(a) − 1 do     ▷ For each digit (D)
 8:         if aᵢ > bᵢ then
 9:             return more          ▷ Digit is larger (G)
10:         else if aᵢ < bᵢ then
11:             return less          ▷ Digit is smaller (S)
12:         end if
13:     end for
14:     return equal                 ▷ Equal (E)
15: end procedure
```

Figure 3 compares the Skill Sharpeners and JUMP Math progressions for Algorithm 3. Since there are too many nodes in the JUMP Math progression to visualize here, only the first third of the progression is shown. We can see that there is a considerable difference in how these progressions proceed. The JUMP Math progression, indicated in green, goes more quickly into more complex traces. The Skill Sharpeners progression spends more time going back and forth between simple problems, and never reaches some of the longer traces. We can also see that the first third of the JUMP Math progression omits a class of problems indicated by the traces "H" and "L". The JUMP progression eventually includes a problem in the "L" class but not until late in the progression. These traces correspond to problems in which a number is greater than another number because the number of digits is larger. Whether or not this omission is desirable is up to the educator; however, this example shows how a procedural analysis can find holes in progressions.

We can use the n-gram model presented in the partial ordering section to look for missing traces even more deeply. Figure 4 shows an analysis of the n-grams for four progressions for different problems. We compute all of the trace n-grams for each level for each progression, and identify missing n-grams by comparing them to a complete progression. Note
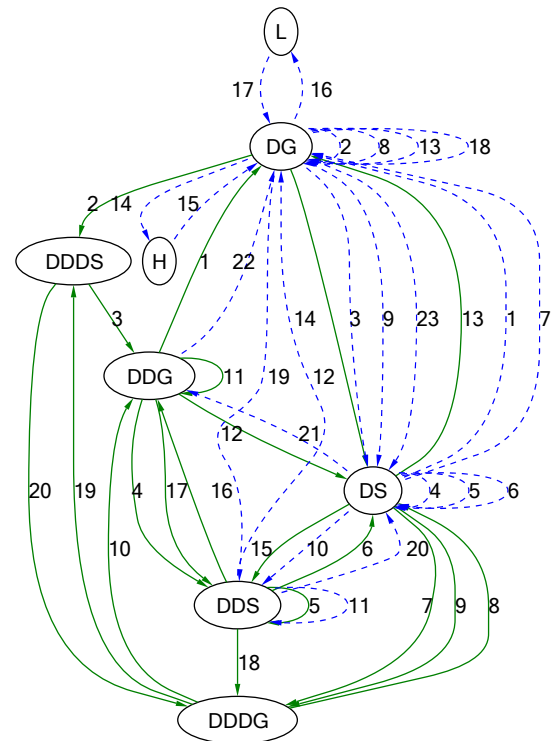

Figure 3. Filling in holes in progressions. This figure compares two progressions for integer comparison (Algorithm 3). "H" corresponds to the trace that gets executed when the first number has more digits than the other and is therefore greater. "L" corresponds to the trace that gets executed when the first number has fewer digits. "D" signifies that the number of digits is the same and that the execution path moves down the digits from left to right. "G" signifies that the digit under examination was smaller and "S" means that the digit under examination was bigger. Therefore, "DG" means that the first number is greater than the second because the first digit was greater. The blue dashed arrows indicate the Skill Sharpeners: Math progression and the green solid arrows indicate the JUMP Math progression. The JUMP Math progression quickly reaches more difficult problems than the Skill Sharpeners progression. The Skill Sharpeners progression never reaches traces longer than "DDG" and "DDS". It also omits problems in which the first number is greater or less than the second number because the number of digits is different. We can "repair" such holes by borrowing problems from other progressions or generating them automatically.

that not all combinations of all letters are possible. For example, the integer comparison trace "H" cannot combine with any other letters because the program returns immediately after that statement.

SYNTHESIS OF PROGRESSIONS

We employ four different strategies for generating and gathering problems: (i) conducting brute force enumeration over inputs, (ii) using test input generation tools to explore the space of execution paths, (iii) using test input generation tools to generate problems that lead to the execution of a desired path by writing a straight-line program with assertions corresponding to that path, and (iv) collecting problems from textbooks.

Large-scale exploration of execution paths

Manual software testing, in general, and test input generation, in particular, are labor-intensive processes. The need

Problem	Book	# problems	# unique traces	% generated
Addition	Skill Sharpeners: Math	584	17	88%
	Singapore Math Sprints	576	48	64%
Fraction Computation	Skill Sharpeners: Math	66	13	100%
	Singapore Math Sprints	248	15	100%
	JUMP Math	131	11	95%
Fraction Reduction	Skill Sharpeners: Math	66	13	100%
	JUMP Math	56	11	100%
Integer Comparison	Skill Sharpeners: Math	24	6	100%
	JUMP Math	88	12	100%

Figure 2. Summary of textbook progression analysis. We analyzed progressions from three textbooks for four different problem domains. We partitioned problems into groups based on their execution trace. The final column shows how many of these traces we were able to generate using Pex, a test input generation tool.

Problem	Book	n-gm.	% Cov.	Missing Traces
Frac. Comp.	JUMP	1	100%	
		2	100%	
		3	93%	SOA
		4	65%	MOAO, OSOA, MOSO, DOSO, SOSO, AOSO, SOAO
	S.S.	1	100%	
		2	75%	OA, OS
		3	50%	DMO, OAO, AOA, OSO, SOA, AOS, SOS
		4	30%	DMOS, DMOA, MOAO, OAOA, OSOA, MOSO, DOAO, AOAO, OSOS, DOSO, SOSO, AOSO SOAO
Int. Comp.	JUMP	1	80%	H
		2	100%	
		3	100%	
		4	100%	
		5	100%	
		6	100%	
	S.S.	1	100%	
		2	100%	
		3	67%	DDD
		4	0%	DDDG, DDDS, DDDD
		5	0%	DDDDG, DDDDS, DDDDD
		6	0%	DDDDDG, DDDDDS, DDDDDD

Figure 4. We can use the n-gram model to evaluate progressions. This table shows an analysis of progressions from JUMP Math and Skill Sharpeners (abbreviated as S.S.) for both fraction computation (Algorithm 2) and integer comparison (Algorithm 3). The table lists what percentage of the feasible n-grams, for various values of n, were covered by all problems together in the progression. The table also lists the missing n-grams. Integer comparison traces involving "E" are omitted because the goal of the exercise was to determine which number is larger.

to reduce the cost of software testing and maintenance has led to development of automated tools for generating test inputs that achieve a high level of coverage. Pex [31] is one such tool that automatically produces a small test suite with high code coverage for a .NET program. It performs path-bounded model-checking by repeatedly executing the program and solving constraint systems to obtain inputs that will steer the program along different execution paths, following the idea of dynamic symbolic execution [11]. Pex uses the theorem prover and constraint solver Z3 [8] to reason about the feasibility of execution paths, and to obtain ground models for constraint systems.

We applied Pex to generate problems for several elementary school mathematics procedures, shown in Figure 5. For each of these problems, we let Pex run for a certain length of time that we varied depending on how many paths we wanted to generate for a particular problem. We then analyzed the execution pathways of each of the problems that Pex generated and partitioned them by trace.

Generating problems to exercise a specific pathway

Pex's goal is to achieve high code coverage. While this allows Pex to generate a large variety of pathways quickly and efficiently, it also creates the possibility that it may not find certain pathways because its goal is statement coverage, not path coverage. However, we can still use Pex to generate problems for a specific trace by converting the procedure to a straight-line procedure. Essentially, we take the desired path and instrument the procedure with assertions that will constrain the execution of the procedure along that particular path.

Using this method on Algorithm 2, we were able to generate an input problem that creates the trace "DAAA" in 40 seconds and "DMMMMMMS" in 35 seconds.

EVALUATION OF OUR SYNTHESIS TECHNIQUES

Textbook problems

Figure 5 shows how we were able to generate large numbers of unique traces with Pex for several math topics. This table indicates how long we let Pex run and how many unique traces it produced. Figure 2 shows what percentage of the set of traces found in the textbooks we were able to generate. It is not 100% for all problems, but we can use the straight-line program method to generate the missing buckets. These results show that using test input generation tools is sufficient for generating problems in this domain.

An Algebra-based Learning Game

In order to validate our framework on a different procedural task domain, we conducted a user study by using Pex to generate several levels for a video game.

DragonBox (WeWantToKnow 2012), shown in Figure 6, is a video game that became the most purchased game in Norway on the Apple App Store [20]. It features game mechanics that involve solving algebra equations. The game does not appear to be an algebra lesson, but the player cannot succeed without learning algebraic simplification and variable isolation. These topics are learned through the level progression. DragonBox represents algebraic expression trees visually as a set

Problem	Time	# unique traces	Constraints
Addition	64 min	1433	# digits ≤ 4, # addends ≤ 6
Fraction Computation	16 min	72	# fractions ≤ 4, numerator ≤ 20, denominator ≤ 20, operations $\in$ addition, subtraction
Fraction Reduction	2 min	29	numerator ≤ 50, denominator ≤ 50, numerator $\leq$ denominator
Integer Comparison	2 min	26	# digits ≤ 8
Subtraction	2 min	157	# digits ≤ 6, difference ≥ 0
Prime Factorization	2 min	70	number ≤ 100000

Figure 5. We used Pex to generate many practice problems for these mathematical procedures. In each exploration, we let Pex run for the indicated length of time and it produced the indicated number of unique traces for that problem. Pex can clearly generate many interesting inputs by directly exploring traces through the procedure. Figure 2 shows the percentage of each textbook progression that this exploration was able to cover.

Figure 6. A level of DragonBox. The goal is to simplify an algebraic equation by isolating the variable, indicated by a box. Each half of the screen represents a side of the equation. The green spiral represents a zero and the two fish cards represent 5 and -5. The bug card represents 10. The player can solve this level by executing the following three rules: $+0 \to \emptyset$, $a + (-a) \to 0$, and $+0 \to \emptyset$. ©WeWantToKnow

of cards that represent numbers and variables. On each level, the screen is divided into two halves that represent the two sides of an equation. By dragging cards around, the player can add, multiply, or divide both sides of the equation by a number or a variable, and perform algebraic simplifications. The goal for each level is to isolate the DragonBox card, which represents an unknown variable.

Level Synthesis

DragonBox levels are essentially expression trees that represent algebraic equations. Leaf nodes can be a variable "x" or an integer. Integers can have multiple forms. They can either be an animal card with a unique picture, a number, or a constant like "a" or "c". Since these forms are essentially equivalent, we only focused on generating integer values and rendered all of these values with an animal card. Internal nodes can be either addition, multiplication, or division. The DragonBox interface imposes an ordering on these operations: addition cannot appear below multiplication or division, and division cannot appear below multiplication.

The procedure we wrote to solve DragonBox levels is described in Algorithm 4 in the Appendix. This algorithm is sufficient to solve all of the levels of the expert progression except for three at the end, which require a more complex algorithm that we leave for future work.

Because there are a huge number of possible expression trees for both the left and right-hand sides of the equation, we used

Pex to rapidly explore pathways and generate a good coverage of test levels. Pex ran for two hours and generated 781 unique levels. These levels corresponded to 152 unique traces. These traces covered roughly 45% of the original progression traces, but represented a large number of unique and interesting levels.

Synthesis User Study

In order to evaluate the synthesized levels and determine whether or not they were equivalent to the expert-designed levels, we conducted a user study. Twenty participants were recruited through an internal email list at the University of Washington that is read by faculty, staff, and students.

Each participant played 30 levels of DragonBox. These levels were organized into fifteen pairs, each consisting of one automatically generated level and one expert-designed level from the original progression. After playing the two levels in the pair, each participant was asked to judge the difficulty of the first level compared to the second level. Five options were given: "much easier", "slightly easier", "about the same", "slightly harder", and "much harder". Participants played the pairs in a random order and the order of levels within each pair was also randomized.

We organized the levels such that five of the pairs contained levels that our model considered to be equivalent. For these pairs, the traces were exactly the same. In four of the level pairs, the trace of the original level subsumed the trace of the generated level according to our path-based partial ordering, and therefore our model considered the original level to be more difficult. In another three level pairs, the trace of the generated level subsumed the trace of the original level, and the original level was considered to be easier. For the final three pairs, we picked a generated level and an original level with the same trace length but different actions, indicating different conditional branching in the trace. As a result, our model specified no partial ordering for these pairs. Within each category, we tried to pick pairs so that the set would have a range of trace lengths and actions (like "add a to both sides"). For the "easier" and "harder" categories, the more difficult trace included one to three more actions than the easier trace.

Partial ordering was better than random by factor of 3
We first tried to determine whether our partial ordering could predict whether the level pairs were equal in difficulty or not. We found that the percentage of participants who considered a level pair to be equal increased from 25% when our model predicted that the pair was unequal to 62% when our model

predicted that the pair was equal. This was statistically significant, $\chi^2(1, 241) = 33.97$, $p < 0.0001$. This indicates that our model reflects some human intuition about level equality.

We then compared the root-mean-square error of our model's predictions with that of other baselines. We first grouped the "slightly harder" and "much harder" responses together, giving them a value of 1, and we did the same for "slightly easier" and "much easier" giving those responses a value of -1. We assigned a value of 0 to an equal response. We then measured the average response for each level, and computed the root-mean-square error of our model's predictions for each level. The root-mean-square error of our model overall was 0.35, whereas the error for making random predictions each time was 1.00, roughly three times as much.

Just using the trace length is insufficient
We observed that a simpler metric, the trace length, makes many of the same predictions as our path-based partial ordering. Therefore, to motivate the need for our more complex partial ordering, we compared it with this simpler model. In Algorithm 4, whenever our model predicts that level L_1 is harder than level L_2, it is always the case that the trace of L_1 has more statements than the trace of L_2. Therefore, for this particular procedure, a simpler model that just compares the length of these traces makes similar predictions, and actually makes the same predictions as our path-based partial ordering in cases where one trace subsumes another.

However, our partial ordering does not make a prediction when one trace does not clearly subsume the other, indicating where more data is needed. One such case is when two traces have the same number of actions but the actions themselves are different. In this event, a trace-length metric would predict that these levels are equal, although this is likely naïve because the actions themselves are different and those differences would likely cause a human to feel that the levels are different.

To examine this, we calculated the error for the trace-length metric on the three level pairs with the same trace length but different actions. This error was 0.51, which is higher than the 0.35 error that the path-based partial ordering achieved on the rest of the problems. This suggests that looking at trace length alone is not sufficient, and we need a model that carefully examines the sequence of statements within a trace.

CONCLUSION
We have defined a trace-based framework for using execution traces to explore the space of problem progressions for any procedural task that can be specified as a computer program. We have shown how this trace-based framework can categorize problems according to their traces, define partial orderings that can compare two problems and assemble a set of problems into a progression, and analyze the thoroughness and depth of existing progressions.

Our framework has allowed us to borrow ideas from test input generation in software engineering and apply them to problem generation. Test input generation tools can automatically explore pathways through a procedure, systematically generate large sets of problems that correspond to various unique

pathways through this procedure, and generate problems that follow a particular desired path. We applied the framework to two domains: early mathematics education, and level generation for a interactive puzzle game. Our framework synthesized hundreds of unique levels for a popular algebra puzzle game and we conducted a user study to confirm that these levels are similar to the expert-designed progression.

Although most of grade-school mathematics education is procedural in nature, some of it also involves conceptual elements that our framework does not fully address. Future work will allow us to include some of these conceptual elements.

In the future, we plan to address student misconceptions and errors. We believe that our framework can be extended to automatically identify likely errors, generate problems that can isolate such errors, and possibly even preempt them by synthesizing interventions in the form of animations or additional problems. We also intend to collect learning data from multiple problem progressions and use machine learning to learn features of these progressions that are predictive of success.

REFERENCES
1. N. Aduba. JUMP Mathematics in Lambeth: Impact on KS2 National Tests 2009. http://jumpmath1.org/research_reports, Nov. 2009.

2. V. Aleven, B. M. McLaren, J. Sewall, and K. R. Koedinger. The cognitive tutor authoring tools (CTAT): preliminary evaluation of efficiency gains. In *Proceedings of the 8th international conference on Intelligent Tutoring Systems*, ITS'06, pages 61–70, Berlin, Heidelberg, 2006. Springer-Verlag.

3. E. Andersen, Y.-E. Liu, R. Snider, R. Szeto, S. Cooper, and Z. Popović. On the harmfulness of secondary game objectives. In *FDG '11: Proceedings of the Sixth International Conference on the Foundations of Digital Games*, New York, NY, USA, 2011. ACM.

4. E. Andersen, Y.-E. Liu, R. Snider, R. Szeto, and Z. Popović. Placing a value on aesthetics in online casual games. In *CHI '11: Proceedings of the SIGCHI conference on Human factors in computing systems*, New York, NY, USA, 2011. ACM.

5. E. Andersen, E. O'Rourke, Y.-E. Liu, R. Snider, J. Lowdermilk, D. Truong, S. Cooper, and Z. Popović. The impact of tutorials on games of varying complexity. In *CHI '12: Proceedings of the SIGCHI conference on Human factors in computing systems*, New York, NY, USA, 2012. ACM.

6. P. Cerny, S. Gulwani, T. Henzinger, A. Radhakrishna, and D. Zufferey. Specification, verification and synthesis for automata problems. Technical report, 2012.

7. M. Csikszentmihalyi. *Flow: The Psychology of Optimal Experience*. Harper & Row Publishers, Inc., New York, NY, USA, 1990.

8. L. M. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, pages 337–340, 2008.

9. T. Dong, M. Dontcheva, D. Joseph, K. Karahalios, M. Newman, and M. Ackerman. Discovery-based games for learning software. In *Proceedings of the 2012 ACM annual conference on Human Factors in Computing Systems*, CHI '12, pages 2083–2086, New York, NY, USA, 2012. ACM.

10. J. Dormans. Adventures in level design: generating missions and spaces for action adventure games. In *PCGames '10: Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, pages 1–8, New York, NY, USA, 2010. ACM.

11. P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI*, pages 213–223, 2005.

12. F. Grabler, M. Agrawala, W. Li, M. Dontcheva, and T. Igarashi. Generating photo manipulation tutorials by demonstration. In *ACM SIGGRAPH 2009*, New York, NY, USA, 2009. ACM.

13. T. Grossman, G. Fitzmaurice, and R. Attar. A survey of software learnability: Metrics, methodologies and guidelines. In *CHI '09: Proceedings of the 27th international conference on Human factors in computing systems*, New York, NY, USA, 2009. ACM.

14. S. Gulwani. Synthesis from examples. *WAMBSE (Workshop on Advances in Model-Based Software Engineering) Special Issue, Infosys Labs Briefings*, 10(2), 2012. Invited talk paper.

15. S. Gulwani. Synthesis from examples: Interaction models and algorithms. *14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, 2012. Invited talk paper.

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

17. N. Jurkovic. Diagnosing and correcting student's misconceptions in an educational computer algebra system. In *ISSAC*, pages 195–200, 2001.

18. N. Li, W. W. Cohen, and K. R. Koedinger. Problem order implications for learning transfer. In *ITS*, pages 185–194, 2012.

19. C. Linehan, B. Kirman, S. Lawson, and G. Chan. Practical, appropriate, empirically-validated guidelines for designing educational games. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 1979–1988, New York, NY, USA, 2011. ACM.

20. J. Liu. Dragonbox: Algebra beats angry birds. *Wired*, June 2012.

21. D. McArthur, C. Stasz, J. Hotta, O. Peter, and C. Burdorf. Skill-oriented task sequencing in an intelligent tutor for basic algebra. *Instructional Science*, 7(4):281–307, 1988.

22. Microsoft. Math Worksheet Generator. http://www.educationlabs.com/projects/MathWorksheetGenerator/Pages/default.aspx.

23. J. Nielsen. *Usability Engineering*. Morgan Kaufmann, San Francisco, CA, USA, 1993.

24. C. M. Reigeluth and F. S. Stein. The elaboration theory of instruction. In *Instructional Design Theories and Models: An Overview of their Current States*, Hillsdale, NJ, 1983. Lawrence Erlbaum.

25. J. Rieman. A field study of exploratory learning strategies. *ACM Trans. Comput.-Hum. Interact.*, 3(3):189–218, Sept. 1996.

26. B. Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

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

28. D. H. Sleeman. A rule-based task generation system. In *Proceedings of the 7th international joint conference on Artificial intelligence - Volume 2*, IJCAI'81, pages 882–887, San Francisco, CA, USA, 1981. Morgan Kaufmann Publishers Inc.

29. A. M. Smith, E. Andersen, M. Mateas, and Z. Popović. A case study of expressively constrainable level design automation tools for a puzzle game. In *FDG '12: Proceedings of the Seventh International Conference on the Foundations of Digital Games*, New York, NY, USA, 2012. ACM.

30. G. Smith, M. Treanor, J. Whitehead, and M. Mateas. Rhythm-based level generation for 2D platformers. In *Proceedings of the 4th International Conference on Foundations of Digital Games*, FDG '09, pages 175–182, New York, NY, USA, 2009. ACM.

31. N. Tillmann and J. de Halleux. Pex-white box test generation for .NET. In *TAP*, pages 134–153, 2008.

32. K. VanLehn. *Mind Bugs: The Origins of Procedural Misconceptions*. MIT Press, Cambridge, MA, USA, 1991.

33. L. S. Vygotsky. *Mind in Society: The Development of Higher Psychological Processes*. Harvard University Press, November 1980 / 1930.

34. Wikipedia. Code coverage. http://en.wikipedia.org/wiki/Code_coverage.

35. Wikipedia. N-gram models. http://en.wikipedia.org/wiki/N-gram.

APPENDIX

Algorithm 4 Dragon Box

```
1:  procedure DRAGONBOX(left, right)
2:      simplify(left, right)
3:      if x ∈ denom(left) ∨ x ∈ denom(right) then
4:          multiply(left, right, x)
5:          simplify(left, right)
6:      end if
7:      if x ∈ num(left) then
8:          variableSide ← left
9:          otherSide ← right
10:     else
11:         variableSide ← right
12:         otherSide ← left
13:     end if
14:     while variableSide ≠ x do
15:         isolate(variableSide, otherSide)
16:         simplify(otherSide, variableSide)
17:     end while
18: end procedure
19: procedure SIMPLIFY(node)
20:     if node = + ∧ child = 0 then
21:         child ← null                    ▷ Remove +0
22:     else if node = + ∧ a ∈ child₁ ∧ −a ∈ child₂ then
23:         child₁ ← 0                      ▷ Remove a + (−a)
24:         child₂ ← null
25:     else if node = × ∧ child = 1 then
26:         child ← null                    ▷ Remove ×1
27:     else if node = ÷ ∧ a ∈ child₁ ∧ a ∈ child₂ then
28:         child₁ ← 1                      ▷ Remove a ÷ a
29:         child₂ ← null
30:     end if
31: end procedure
32: procedure ISOLATE(varSide, otherSide)
33:     varChild ← varSide.child with variable
34:     nonVarChild ← other child
35:     if varSide = + then
36:         subtract(varSide, otherSide, nonVarChild)
37:     else if varSide = × then
38:         divide(varSide, otherSide, nonVarChild)
39:     else if varSide = ÷ then
40:         multiply(varSide, otherSide, nonVarChild)
41:     end if
42: end procedure
```