Automatic Diagnosis of Students' Misconceptions in K-8 Mathematics

Molly Q Feldman[1], **Ji Yong Cho**[2,*] **Monica Ong**[1], **Sumit Gulwani**[3], **Zoran Popović**[4] **& Erik Andersen**[1]

[1]Department of Computer Science, Cornell University
[2]Department of Computer & Information Science, University of Pennsylvania
[3]Microsoft Research Redmond
[4]Paul G. Allen School of Computer Science & Engineering, University of Washington
{mqf3,myo3,ela63}@cornell.edu, jiycho@seas.upenn.edu, sumitg@microsoft.com, zoran@cs.washington.edu

ABSTRACT
K-8 mathematics students must learn many procedures, such as addition and subtraction. Students frequently learn "buggy" variations of these procedures, which we ideally could identify automatically. This is challenging because there are many possible variations that reflect deep compositions of procedural thought. Existing approaches for K-8 math use manually specified variations which do not scale to new math algorithms or previously unseen misconceptions. Our system examines students' answers and infers how they incorrectly combine basic skills into complex procedures. We evaluate this approach on data from approximately 300 students. Our system replicates 86% of the answers that contain clear systematic mistakes (13%). Investigating further, we found 77% at least partially replicate a known misconception, with 53% matching exactly. We also present data from 29 participants showing that our system can demonstrate inferred incorrect procedures to an educator as successfully as a human expert.

ACM Classification Keywords
H.5.0 Information Interfaces and Presentation: General; K.3.1 Computer Uses in Education

Author Keywords
programming by demonstration; elementary education

INTRODUCTION
K-8 mathematics students learn many fundamental procedures, such as how to add 3-digit numbers or how to reduce fractions. During this process, they frequently make mistakes and can even learn entirely incorrect procedures. Educators need to identify these errors for a variety of reasons (providing corrections, granting partial credit, etc.), but this process is hard and time-consuming. Math education experts [4,7,46] have analyzed large sets of known student errors and recommended training materials to help educators learn to identify errors better. However, this process remains a roadblock for educators [17]. We envision a future in which educators spend less time trying to reconstruct what their students are thinking and more time working directly with their students.

Automatic identification of students' procedural errors is part of a large body of work in HCI studying user intent. HCI researchers have considered user intent in intelligent tutoring systems [3,6,29–31], generating curriculum/learning material [38,44,45], text, spreadsheet, or web processing [5,9,27,37], visual manipulation [10, 13, 14], and physical interactions [18]. Many of these systems rely on expert authoring, or work done by a domain expert that models how the system should behave for a specific application. Systems then use that expert authoring to convert system input into output automatically. However, expert authoring frequently requires work for every new input, which limits scalability. In addition, inferring the user's intended meaning from their input is a recurring challenge. Our work aims to limit expert authoring and develop technology that can effectively infer user intent in K-8 math.

Several existing approaches for identifying students' math errors [4,7,46] concentrate on a specific type of systematic procedural error, which we refer to as a *misconception*, that occurs when students learn the wrong process for solving certain types of problems. These systems make use of "bug libraries", which are sets of known student misconceptions for a given problem type. Since this approach relies on existing collections of misconceptions, a new collection must be defined for every new math topic. It is also not robust to never-before-seen misconceptions. Ideally, a system to identify students' procedural errors would trace a student's solution process by exploring the set of possible procedures they may have used, rather than comparing against known error patterns. Reconstructing this process allows for more fine-grained understanding. Our approach uses basic math operations, such as single-digit addition or incrementing a number, and combines them together to build a procedure that leads to the student's solution. Specifically, we generate a program with potentially complex control flow (conditionals and nested loops) that models how a student solved a problem set incorrectly (see Fig. 1 for an overview).

[*]Work performed at Cornell University

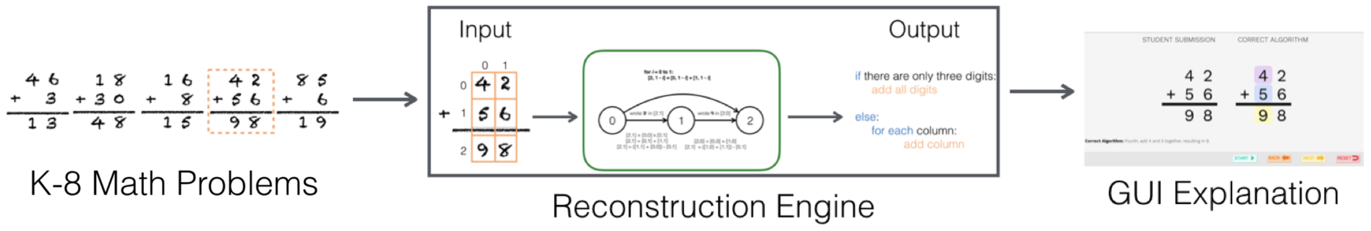


Figure 1. Our system has two major components: a thought-process reconstruction engine which uses program synthesis and a GUI for displaying reconstructed thought processes to an educator. The input is a set of problems that have been solved systematically (although possibly incorrectly) by a student. The engine attempts to synthesize a computer program from the input problems to try to explain what the student was doing. This program is then passed to the GUI, which automatically produces a step-by-step tutorial explaining the error to an educator.

We evaluate our approach on multiple collections of misconceptions. We use a set of common student mistakes curated by an expert [4] to test robustness. We are able to replicate 70% of misconceptions in algorithms ranging from subtraction to fraction reduction. We then evaluate our system's real-world applicability by analyzing it on data from 296 students in 17 classrooms at 11 schools across 7 states collected in 2014. On this data set we are able to generate programs that replicate 86% of the students' solutions to problems classified as containing a systematic error (13%). Investigating further, we found 77% at least partially replicate a known misconception, with 53% matching exactly.

We also evaluate how well our system can help educators understand what students are thinking. Many existing classroom tools for K-8 mathematics help educators understand their students' progress, but they concentrate almost exclusively on correctness [16, 33]. In order to inform educators of their students' misconceptions, we automatically generate step-by-step visual demonstrations of the programs produced by our system. We present results from a 29-participant user study showing that our system can explain incorrect student procedures to an educator as well as a math education expert.

The main contributions of this work are as follows:

- We present a system that reconstructs student misconceptions in K-8 mathematics by combining basic math operators into full procedures that replicate how a student solved a given problem set incorrectly
- We demonstrate that our system can replicate student misconceptions by evaluating it on a set of common student mistakes curated by an expert and data from 296 students
- We built a visualization of our system's output and show that it can can explain students' misconceptions to educators as well as human experts

RELATED WORK

Aiding and Modeling the Learner in Education

There has been significant work in modeling how students approach solving problems in procedural domains, such as mathematics or programming. Brown and Van Lehn's repair theory defines a generative model for reproducing the errors students make when solving procedural problems [8]. Langley and Ohlsson [25] developed the concept of production rules, which check if a particular condition is true about a problem state and then perform an operation. Intelligent tutoring systems [3, 6] train students in procedural tasks using production rules. Later work [30, 31] used a production-rule-learning framework to learn students' errors; however, this approach often learned production rules that were too general or too specific. Jarvis et al. [20] apply machine learning to generate production rules for automating intelligent tutoring system creation. The size of production rules produced by this system is limited due to the brute force nature of its algorithm. Li et al. use a machine learning agent to learn complex production rules for algebra from examples [29]. They test the validity of their technique for a single data set in a single domain (algebra). In comparison, we evaluate our approach on two data sets containing data for multiple math algorithms and hundreds of students.

In contrast to production rules, our approach generates complete imperative programs with nested loops and conditionals within loops. This is important because accurately identifying systematic errors requires a complete understanding of the student's overall process, such as determining that the student is (or is not) applying the same process to each column in a subtraction problem.

BUGGY [7] and DEBUGGY attempted to generate descriptions of K-8 math student errors using a hardcoded bug library built from a large set of incorrect student solutions. Van Lehn [46] built the Sierra system which produced student errors based on repair theory and training from the BUGGY data set. Sison and Shimura provide an overview of other core AI methods for feedback generation [43]. Since a bug library cannot address previously unseen errors, our approach instead searches through how a student may combine basic math operations, like single-digit addition, into complex procedures. The DIAGNOSER system [19, 28] helps students learn conceptual physics by asking them to justify their answers for multiple choice problems. In comparison, our system works for open-ended math problems and the misconceptions that arise from incorrectly learned math procedures.

More recently, there have been procedural methods for assessing correctness in programming [39, 42], discrete finite automata [1], and embedded systems [21]. Refazer [40] moves beyond correctness to determine how students transform programs while writing assignments, and can learn transformations without any hardcoded bug library or error model. Head et al. [15] extend Refazer by building a system that directly interacts with an educator. They capitalize on both expertise and automation to provide better feedback. Other recent work has clustered student programming assignments into similar groups and provides personalized feedback using semi-supervised methods [22]. Many of the systems noted above leverage modern advances in program synthesis technology. We leverage program synthesis to combine small conceptual units into procedures that model student intent. We believe ours is the first program synthesis system that directly demonstrates this capability for K-8 mathematics.

Programming-by-Demonstration (PbD) in HCI

There is a significant body of work in HCI focused on inferring user intent. A common technique is programming-by-demonstration (PbD), which analyzes demonstrations of user intent and constructs a program that reproduces these demonstrations. A major application area for PbD is automatic tutorial generation [36, 44, 45] and instructional scaffolding [2, 38]. Our method for visualizing the results of our PbD system was heavily inspired by O'Rourke et al.'s work on automatic visual tutorials for procedural skills [38].

PbD has also been applied to text editing and viewing. Mitchell et al. introduced the concept of version space algebras which allow efficient computation of a large number of hypothetical programs [35]. The SMARTedit system used a version space algebra to learn repetitive text-editing procedures with simple loop structures [27]. Our technique is able to synthesize programs with nested loop structures and conditionals. DocWizards [5] generates automatic walkthroughs of computer documentation. As input it records a user walkthrough of a procedural task. DocWizards then automatically generates documentation for the task that can guide a new user by suggesting the steps they need to take, potentially including non-linear control flow. Unlike our work, DocWizards walkthroughs cannot use portions of one walkthrough to inform the creation of another unrelated tutorial. In other words, DocWizards' internal representation of steps is not modular. Our system is able to model user intent that contains complex control flow using composable math operators. We apply this system to recover user intent in K-8 mathematics by representing student thought processes as programs.

Program synthesis also extends to more general HCI applications. In particular, there has been significant work building systems that interact with the user through demonstration. Sketch-n-Sketch [11] allows automatic loading of input-output examples to generate scalable vector graphics in real time. The FlashProg system [32] allows users to specify data extraction tasks in a UI which are then executed and modeled using a program synthesis backend. As noted above, our contribution is using program synthesis to model student intent in K-8 mathematics by combining individual mathematics concepts into programs representing student solution processes.

Commercially Available Software

Time to Know® produced one of the first fully digital learning platforms complete with personalized question sets for the CommonCore curriculum. Their technology is currently being used by McGraw-Hill in their Thrive environment [33]. A similar curriculum based tool, HeyMath!® [16], is popular for its data driven feedback mechanism. However, these tools do not attempt to address misconceptions or understand what the student is doing at a semantic level; they concentrate almost exclusively on correctness. Our system, in contrast, is able to explain misconceptions step-by-step to an educator. Gradescope [41] aims to help educators grade students more efficiently by providing smart aggregation of similar student responses, distributed grading of one assignment across many graders, and a single unified rubric that can apply pre-specified comments. We focus on modeling student intent to help resolve misconceptions that lead to incorrect answers on assignments and lower grades.

MISCONCEPTIONS IN MATHEMATICS

Student errors in mathematics include careless mistakes, incorrect fact recall, and systematic errors in which the wrong algorithm is used [4, 46]. We focus only on this last class of systematic errors, called *misconceptions* in the literature [12]. We leverage four well-known sources in this area: the misconceptions used in the BUGGY and Sierra systems as described by Van Lehn [46], a math education resource book from Ashlock [4], a data collection study by Cox [12], and a Department of Education Technical Report [26]. These resources typically present a misconception as a problem set that a student has solved incorrectly, alongside a text description of the misconception that was either written by an expert or obtained through an interview with the student.

The main challenge in automatic misconception identification is the sheer number of possible misconceptions for a single topic. Van Lehn [46] identified over 100 distinct misconceptions for subtraction alone. As an example, consider these systematic errors for addition ($a_1 + a_2$) problems from [4]:

A-W-1: Add each column and write the sum below, even if it is greater than nine.
A-W-2: Add each column from left to right. If the sum is greater than nine, write the tens digit below and the ones digit above the column to the right.
A-W-3 (Fig. 2, left): Only applies to problems in which a_1 has two digits and a_2 has two digits or one digit. If a_2 has one digit, add all three digits and write the sum. If a_1 and a_2 both have two digits, add each column normally.
A-W-4: Only applies to problems in which a_1 has two digits and a_2 has one digit. Add in a manner similar to multiplication. For each column, moving from right to left, add the digit of a_1 in that column to a_2. Carry if the sum is greater than nine and include in the next sum.

We define a *solved problem set (SPS)* as a set of 3 or more problems with solutions provided by a single student. A-W-3 and E-F-3 in Fig. 2 are SPSes that contain a misconception.

Although there are many different individual misconceptions for a variety of math topics, the steps a student takes while computing an incorrect solution are remarkably similar, when viewed through the proper lens. For instance, consider E-F-3 (Fig. 2, right) in which a student divides the larger of the numerator and denominator by the other value, drops any remainder, and uses the larger number as the denominator. Comparing E-F-3 to A-W-3, it is not immediately clear how one could develop a unified approach to reconstructing both students' thought processes. Our key insight is that at each

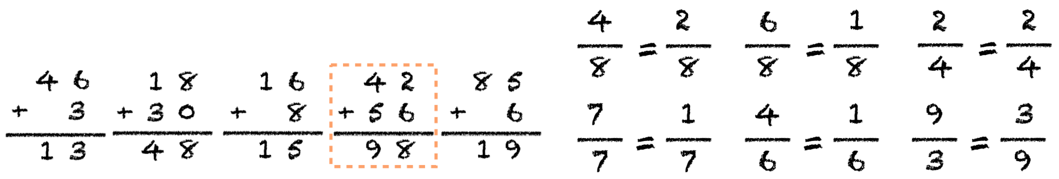


Figure 2. Misconceptions in solved problem sets A-W-3 (left) and E-F-3 (right) from [4]. Expert descriptions of these misconceptions are in Fig. 9.

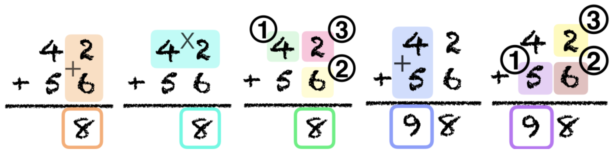


Figure 3. Our thought process reconstruction algorithm tries to generate hypotheses for why the student wrote each number in his or her solution. For each value in the input demonstration, the algorithm tries to explain that value using a set of operators provided to the system. For example, the 8 could be $2+6$, or 4×2, or ① + ② − ③, i.e. $4+6-2$. The 9 could be $4+5$, or ① + ② − ③, i.e. $5+6-2$.

step of their process, both students are using a basic math operation to combine one or more numbers in a given problem. For instance, the first step in A-W-3 could be "add 2 and 6 together". The equivalent first step in E-F-3 may be "divide the larger number by the smaller number."

Our goal is to build a system that can automatically produce programs representing student thought processes, when given a SPS. We call a program produced by our system for a given SPS a *reconstruction program*. Our system specifically focuses on building reconstruction programs by exploring procedural compositions of basic conceptual units without any additional information, such as referencing known misconceptions or a correct algorithm.

Automatically generating reconstruction programs presents multiple challenges. First, we need to be able to recognize *low-level operations*, such as basic column addition. Second, we need to be able to infer *high-level control flow* over those basic operations. For instance, we need to recognize that the columns are being added left to right in A-W-2.

In order to accomplish these goals, we need to search through a large space of hypotheses. For example, in order to explain how the student solved the boxed problem in A-W-3, we need to first explain why they wrote the 9 and the 8. We can do this by searching through a set of *base operators* that are provided to our system, such as single-digit addition, decrementing a value, taking the ones digit of a sum, determining the smaller of two values, etc. A set of operators is specified for each problem type (addition has a set, subtraction another, etc.) Note that there are at least three unique operators that can be used to calculate 8 as the ones digit for the boxed problem, as shown in Fig. 3. Determining which hypothesis is most likely or most accurate is very difficult as our input is a single SPS.

In order to learn the student's high-level process we need to search through an even larger space of possible control flow. This can include conditionals to represent choice and loops to represent a repetitive process. For example, we need to be able to infer that the student is adding the same way for every column in A-W-3 for 2-digit plus 2-digit problems.

Our technical solution to considering all hypotheses is to frame this task as a programming-by-demonstration problem and design an algorithm that can search through a large hypothesis space. In order to encode SPSes as input to our algorithm, we use a Thought Process Language (TPL) as introduced by O'Rourke et al. [38]. Our adapted TPL includes constants, integer operators (e.g. +, -, *, /), and Boolean operators (e.g. <, >, ==, !=). It also includes three types of statements: update statements that write values (i.e. perform computations by applying an operator), conditionals, and for loops. Our algorithm combines these statements together to create a program that represents the student's thought process.

The set of base operators and the TPL are specified once and then can be used for multiple misconceptions, allowing our system to detect misconceptions that are novel combinations of the same basic conceptual units into an incorrect high-level (or even low-level) process. The system can capture any systematic error that is (1) constructed from the set of provided base operators, (2) encodable in TPL, and (3) sufficiently short in length. We have tested the approach and obtained reconstruction programs with up to 14 statements. As an example, our system generates the following reconstruction program for E-F-3 (shown here in psuedocode):

```
if (denominator < numerator):
  resultNumerator = numerator / denominator
  resultDenominator = numerator
else:
  resultNumerator = denominator / numerator
  resultDenominator = denominator
```

THOUGHT PROCESS RECONSTRUCTION

Input Format

Most of the misconceptions in our primary sources [4, 12, 26, 46] can be represented as computations on cells of a table. Indeed, many different topics in K-8 math, such as addition, fraction reduction, and long division, can be represented as table computation problems. Therefore, we encode each problem in a SPS as its own table with a specific row / column for the solution. For example, see Fig. 4 for an encoding of our running example. The student's thought process then becomes equivalent to manipulating table values. The top left cell of the table is always the origin ([0,0]). Each problem type uses the table slightly differently, but each problem in a given SPS needs to be encoded in the same way for the system to work. The exact input format we use consists of a set of tuples (value, row, column, time). The time represents when the student wrote the value during their solution process.

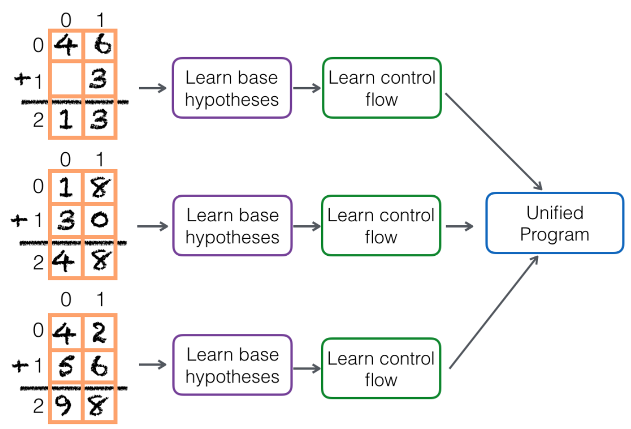


Figure 4. A high-level diagram of our approach. For each problem in a SPS, we first generate hypotheses for why the student might have written each digit (Step 1). Then, for each example, we learn higher-level control flow such as loops and conditionals (Step 2). Finally, we intersect the hypotheses generated for each example to obtain one single set of hypotheses that matches all of the examples (Step 3).

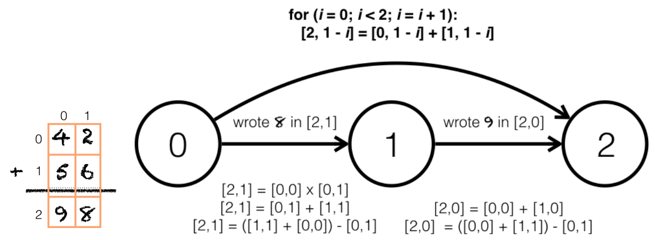


Figure 5. We use a directed acyclic graph (DAG) to store a large set of hypotheses regarding what the student's process was. Node n represents the state of the table at time n. An edge from node m to node n represents operations that change the table from its state at time m to its state at time n. Initially, this includes the concrete value that was written to the table and its location, such as that an "8" was added to the table at location [2,1] (Step 1). Later on, we add hypotheses to the DAG that this value was written because it was the result of an operation applied to other values in the table (Step 2). Even later, higher-order hypotheses are added, such as the loop that connects node 0 to node 2 (Step 3).

Algorithm Details

The high-level process used by the algorithm is shown in Fig. 4. We take a bottom-up approach, in which we first learn the most basic operators, and then iteratively learn higher-order control flow structures over these basic operators. Since the tool takes a sequence of values written to the table as input, we organize the possible ways to combine operators (which we call *hypotheses*) by where they appear in this sequence. To do this, we use a directed acyclic graph (DAG) which is a linear chain of nodes. Each node represents a step in the student's computation. Edges hold programs that convert the state of the table from time n to time $n+1$ (Fig. 5).

Step 1: We first try to explain why the student performed each low-level operation. To do this, we step through the input. For each value that the student wrote, we apply the provided operators to problem digits until we obtain a hypothesis that results in the student's answer (Fig. 3). We store all of these hypotheses in the DAG. For example,

start $\to$ *end*	Hypotheses
$0 \to 1$	$[2,1] = [0,0] \times [0,1]$ $[2,1] = [0,1] + [1,1]$ $[2,1] = ([0,0] + [1,1]) - [0,1]$
$1 \to 2$	$[2,0] = [0,0] + [1,0]$ $[2,0] = ([1,0] + [1,1]) - [0,1]$

Step 2: The next step is to try to learn control flow over these basic operations. We first try to see what repetitive operators can be pulled into a loop. For example, the following two hypotheses:

$0 \to 1$	$[2,1] = [0,1] + [1,1]$
$1 \to 2$	$[2,0] = [0,0] + [1,0]$

can be unified into the following loop, which appears at the top of Fig. 5:

$0 \to 2$	`for` $(i=0; i<2; i=i+1)$`:` $[2,1-i] = [0,1-i] + [1,1-i]$

Loops can potentially begin or end at any step. Therefore, we learn complex control flow by trying all start and end points in the DAG as follows:

```
for ($i=0$; $i<n$; $i=i+1$):
  for ($j=i+1$; $j<n$; $j=j+1$):
    find loops from node $i$ to node $j$ in DAG;
    add these loops to DAG
```

This approach is novel as it can identify repetitive thought processes that begin or end at any step, making it expressive and scalable enough to capture a huge range of possible control flow. We can run the loop learning process multiple times to learn complex control flow structures like nested loops and conditionals inside loops.

To learn loop bodies (the sequence of statements inside loops), we use *templates*, which are skeletons of code consisting of "holes" for statements or conditionals. Here are some examples of possible templates:

```
(A)            (B)            (C)
<statement>    <statement>    if <conditional>:
               <statement>      <statement>
               <statement>      <statement>
                              else:
                                <statement>
                                <statement>
```

In practice, we have found that most reconstruction programs we want to generate are small enough that this process is sufficient. To learn loops, we enumerate all templates with no more than X statements, Y (possibly nested) conditionals, and Z statements per conditional. For the results in this paper, $X = 5$, $Y = 3$, and $Z = 3$.

Step 3: Once we have learned base hypotheses and control flow for each problem separately, the next step is to unify the problems together by identifying a single set of hypotheses consistent with every problem in the input SPS. To do this, we use an intersect operation that eliminates all of the hypotheses that are not consistent with the entire SPS. For example, when we intersect the DAGs associated with the problems $42+56=98$ and $18+30=48$, two hypotheses remain: that the student added both columns in a loop, or that the process really does just involve two separate column additions. These hypotheses are functionally equivalent for these two examples, but they are distinct semantically. It is significant whether or not the student knows that they need to do something for every column.

We also use templates for the unification process. They are similar in format to those used to learn loop bodies, but may contain more statements because they represent the structure of the student's entire thought process (not just the part that can be represented as a loop). We used 164 templates at maximum, which were the set of all possible templates with a maximum of 8 statements, 1 conditional and 3 statements per conditional branch. Instead of trying all templates at once, we used an iterative, phased strategy that tried various subsets of templates until a program was generated.

When the provided hypotheses for the problems $46+3=13$, $16+8=15$, and $85+6=19$ are intersected, we can obtain:

$$\begin{array}{c|l} 0\rightarrow 1 & [2,1]=([0,0]+[0,1]+[1,1])\%10 \\ 1\rightarrow 2 & [2,0]=([0,0]+[0,1]+[1,1])/10 \end{array}$$

We are able to learn a final reconstruction program when we intersect these two hypotheses and the loop that was learned in Step 2 using template (C) above:

```
if ($[0,1]$ is empty):
    $[2,1]=([0,0]+[0,1]+[1,1])\%10$
    $[2,0]=([0,0]+[0,1]+[1,1])/10$
else:
    for ($i=0; i<2; i=i+1$):
        $[2,1-i]=[0,1-i]+[1,1-i]$
```

We fill in conditionals by keeping track of the state of the table every time it executes for each problem, and later searching through the provided boolean operators to identify a condition that produces the correct result (*True* or *False*) for each intended invocation. This can be complex, as the number of row/column combinations is frequently large.

Note that even this reconstruction program isn't really "complete", in the sense that it cannot predict how a student would solve a problem that had three columns instead of two. This is because the provided SPSes are not sufficiently rich to disambiguate whether the student is doing something for each column, or whether the student is intending to add exactly two columns. More data from the student would be required.

Step 4: If we obtain multiple reconstruction programs that are consistent with the entire problem set, we arbitrarily pick one of them.

Evaluation plan

We perform two evaluations. The first uses curated data from Ashlock [4] which demonstrates the breadth of math topics that our approach can handle. The second uses student data collected by MetaMetrics [34] to determine how well our system can handle real classroom data.

Problem Type	Name	<T(s)	A?	Problem Type	Name	<T(s)	A?
Add (+)	A-W-1	1	Y	*Frac. Red.*	E-F-3	1	Y
Add (+)	A-W-2	20	Y	Frac. +	A-F-1	1	Y
Add (+)	A-W-3	1	Y	Frac. +	A-F-2	1	Y
Add (+)	A-W-4	1	Y	Frac. +	A-F-3	1	Y
Subtract (−)	S-W-1	1	Y	Frac. +	A-F-4	1	Y
Subtract (−)	S-W-2	1	Y	Frac. −	S-F-1	1	Y
Subtract (−)	S-W-3	10	Y	Frac. −	S-F-2	130	Y
Subtract (−)	S-W-4	30	Y	Frac. −	S-F-3	1	Y
Subtract (−)	S-W-5	1	N	Frac. −	S-F-4	50	N
Multiply (∗)	M-W-1	1	Y	Frac. ∗	M-F-1	1	N
Multiply (∗)	M-W-2	1	Y	Frac. ∗	M-F-2	1	Y
Divide (÷)	D-W-1	1	N	Frac. ÷	D-F-1	1	Y
Frac. Red.	E-F-1	1	Y	Frac. ÷	D-F-2	1	Y
Frac. Red.	E-F-2	5	Y	Dec. +	A-D-1	1	Y

Figure 6. Summary of misconception benchmarks from [4]. We only present the 28 successful misconceptions here for space reasons. *Name* shows Ashlock's unique identifier for an error pattern. Problems in *blue* are featured in our user study. *<T(s)* shows an upper bound on the number of seconds taken by our algorithm to generate a program solving all of the provided demonstrations. *A?* states whether the program generated by our system adequately represented a student's thought process.

EVALUATION ON CURATED EXPERT DATA

Our first evaluation attempts to generate reconstruction programs for the 40 errors described by Ashlock [4]. We measure our algorithm's performance by how many misconceptions it can identify, along with their complexity and accuracy.

Our system can replicate 70% of the Ashlock misconceptions
We were able to replicate 28 of the 40 misconceptions in Ashlock (excluding those in the appendix), which is about 70% coverage (Fig. 6). The most complex control flow in a reconstruction program was generated for S-F-4 (2 nested loops) and S-F-1 (3 conditionals, 14 total statements).

The misconceptions we were not able to replicate fell into three categories. First, some misconceptions involved base operators that were quite unrelated to the operators used in the correct algorithm. For example, one misconception for multiplying two fractions f_1 and f_2 required a base operator defined as $f_1.num * f_2.denom + (10 * f_1.denom + f_2.num)$. Although our system can replicate misconceptions of this type if provided such a non-standard operator, this was too impractical to include as a successful benchmark. Second, some misconceptions involved too many steps for our system to handle. Third, some misconceptions involved word problems, which are outside of the scope of our system.

We identified that most reconstruction programs generated by the algorithm match the student's behavior as described by Ashlock. In particular, we say that a reconstruction program is accurate if, reading it as psuedocode, we convinced ourselves that it could directly represent a student's thought process. Programs that strictly violated this made use of unnatural operator combinations or unnecessary loop / boolean conditions. We do not claim that this is a quantitative or even unbiased metric. However, it provides a sense of how well the system can represent thought processes. With this in mind, 23 of the 28 (82%) reconstruction programs generated for the Ashlock misconceptions are accurate.

EVALUATION ON STUDENT DATA

We use a data set collected by MetaMetrics in November 2014 from 17 different classes across 11 schools in 7 states for this evaluation. The data contain responses from 296 students to up to 32 addition and subtraction problems.

MetaMetrics collected this data set to study their Quantile® Framework, which targets instruction and progress of a student's math understanding. Each student was assigned to one of six worksheets. The variation between each worksheet included the problem's orientation (horizontal or vertical, see Fig. 7) and skill level. The worksheets tested four math algorithms: addition without regrouping, addition with regrouping, subtraction without regrouping, and subtraction with regrouping.

The study was administered by teachers in their classroom. MetaMetrics provided teachers with a manual, part of which included the following text, which they asked to be read to the students: "You will take a short mathematics test. For most of the items you will need to write the answer on scratch paper and then type your answer into the computer. For the samples, you will not need scratch paper. To complete these items, first do the math problem. Type the answer in the box. Please do the first sample item." Students typed their answer into a text box positioned to the right of the problems (see Fig. 7).

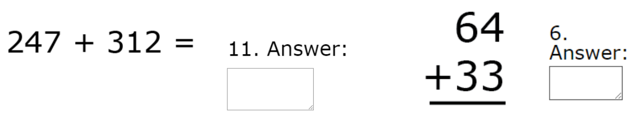


Figure 7. MetaMetrics input method: students calculated their responses and entered them into the answer text boxes. In our work, we do not differentiate between the data based on which interface was used. (Images Courtesy of MetaMetrics, Inc.)

Data Processing

In order to evaluate our algorithm on this data set, we first generated SPSes in the following way. For the set of 32 questions presented in each worksheet, there are contiguous "runs" of 3 to 4 problems asking students about the same type of problem with the same skill level. We consider each of these runs its own problem set, resulting in 6 possible problem sets for every student (4 for addition, 2 for subtraction). Each individual student's answers to each one of these problem sets is a SPS. We only used SPSes with one or more incorrect solutions, yielding 868 total SPSes.

Our system requires a table representation of each problem indicating when the student wrote each digit of their solution. Since this information was not recorded during the MetaMetrics study, we entered each solution digit in order from right to left and kept the leftmost column empty (null). This data set does not contain any written carry values. Compared to the Ashlock evaluation, we used slightly expanded operator sets and a differently optimized template phasing strategy.

We used the following process to analyze the SPSes:

Step 1: We manually inspected each of the 868 SPSes defined above to determine if a SPS had a clear misconception. Using our best judgement, we analyzed each SPS and identified whether it 1) contained a misconception defined in the literature or 2) contained a undocumented misconception that we could clearly identify. We identified 111 SPSes with a misconception, representing 13% of the 868 SPSes.

We observed that in some cases there were multiple valid possibilities for the misconception that was expressed by the SPS. In these cases, we chose one misconception to associate with the SPS, but in our analysis we considered our system successful and/or accurate if it modeled any of them.

Step 2: We inputted each SPS with a misconception into our system and obtained either a reconstruction program or system failure as output. Our system successfully produced a program for 95 of the SPSes with misconceptions (86%).

Step 3: We determined how well each reconstruction program appeared to represent a student's likely thought process. Since there is no established precedent for matching programs describing student behavior to SPSes, we empirically studied preliminary results and used expert consensus between two researchers (the first two authors) to simultaneously develop a coding scheme and apply it to the data.

Ultimately, our coding scheme classified programs into three groups: "Accurate," "Somewhat Accurate," and "Not Accurate." A program was categorized as "Accurate" if it exactly modeled the SPS's misconception. If the control flow or a computation(s) in the reconstruction program did not match exactly, it was categorized as "Somewhat Accurate." If the reconstruction program could not be interpreted as a student thought process or the system failed, the program was categorized as "Not Accurate." If a SPS exhibited multiple misconceptions, we counted the program as "Accurate" if it matched any of the misconceptions. After the two researchers individually classified the data, we obtained a Cohen's Kappa of 0.66, which represents significant agreement [24]. A final category was chosen for all programs where the two researchers disagreed by discussion and ultimate consensus.

Results

Our system replicated 86% of SPSes with a misconception

The first measure of success for evaluating our system was whether or not a reconstruction program was generated for a given SPS with a misconception. For 86% of the SPSes with a misconception, our algorithm successfully produced a reconstruction program. This means that our algorithm is able to generate representations for almost 100 student solutions to addition and subtraction problem sets.

77% of reconstruction programs generated by our system are at least Somewhat Accurate

After classifying reconstruction programs according to the method outlined above, 77% of the generated reconstruction programs were either "Somewhat Accurate" (23%) or "Accurate" (53%). We believe this result points to the robustness of our system's ability to reconstruct student misconceptions.

For an example of a reconstruction program classified as "Somewhat Accurate," consider the program generated for $20+70=50$, $44+32=12$, and $57+10=47$. This SPS exhibits the third misconception in Table 5, Subtable 2 in [12]. Our system generated the following reconstruction program:

```
for ($i=0$; T[0,$2-i$] $\neq$ null; $i=i+1$):
  if (T[$1-i$,1] > T[1,$2-i$]):
    T[2,$2-i$] = T[0,$2-i$] − T[1,$2-i$]
  else:
    T[2,1] = 5
```

This reconstruction program was classified as "Somewhat Accurate" because of the Boolean conditional and else branch. A program that more accurately matches the associated misconception would not differentiate between the first problem (the else branch) and the remaining problems by using a conditional. However, the current program represents a possible misconception for a student who has difficulty with subtracting larger numbers from smaller numbers.

As mentioned previously, the problem of accurately describing the thought process behind a SPS is very difficult because of the poverty of the data set. To be specific, there are typically multiple consistent hypotheses for a given SPS and our system must choose one. Since the main goal of this work was to build a system that could explore basic conceptual units *without any other information*, trying to incorporate additional information like comparison to the correct algorithm or heuristics, etc. is beyond the scope of this work.

53% of the reconstruction programs matched a known misconception exactly

Our system exactly replicated student misconceptions for 53% (or 59/111) of the SPSes. This advances the state-of-the-art for this domain, as there is no other system without an explicit bug library that can compose base operators into programs with potentially complex control flow. The most frequent misconception replicated accurately was "smaller-from-larger" described in [46]. For example, for $322-157=235$, $405-127=322$, $635-166=531$, and $700-586=286$, the following program was classified as "Accurate" ($|\cdot|$ represents absolute value):

```
for ($i=0$; T[0,$3-i$] $\neq$ null; $i=i+1$):
  T[2,$3-i$] = | T[0,$3-i$] − T[1,$3-i$] |
```

The system also captured misconceptions where the student reversed the result digit order from right to left to left to right. For example, consider $44+534=875$, $247+312=955$, and $444+531=579$. This SPS exhibits the first two-digit addition misconception in Appendix A of [26]. The following program was classified as "Accurate":

```
for ($i=0$; T[1,$i+1$] $\neq$ null; $i=i+1$):
  T[2, $2-i$] =
    SumSelectRange(0, $i+1$, 1, $i+1$) % 10
```

Note that the SumSelectRange(m_1,n_1,m_2,n_2) is equivalent to T[0,$i+1$] + T[1,$i+1$] for this problem, except for when an addend does not have a hundreds digit. In that case it carries down the existing hundreds digit (i.e. 5 for $44+534$). Although, from the reader's perspective, this program may not seem ideal, the system is doing exactly what it is built for: constructing a single representative program for the entire SPS using base operators.

23% of the reconstruction programs were not accurate

Programs that were classified as "Not Accurate" either generated code for each problem in the SPS individually or the system failed to generate a program altogether. The majority of programs in this category (12/26) exhibited either multiple misconceptions at once or a misconception not found in the literature. The main reason we believe the system fails on these SPSes is that the system is not being provided with enough data. For the case of multiple misconceptions occurring simultaneously, the system may not have enough data to model both misconceptions individually, let alone together.

Discussion

For the 13% of MetaMetrics SPSes with misconceptions, we generated a reconstruction program for 86%. Of these, 77% at least partially matched a known or plausible misconception. 53% exactly matched a known misconception. Our analysis of the "Somewhat Accurate" and "Not Accurate" SPSes revealed that one of the most common situations in which the reconstruction program deviates from what we would expect is when it infers unnecessary or implausible control flow. For instance, this occurs when the program uses conditionals to deal with inconsistencies between problems, such as the number of digits in the addend vs. the subtrahend. Therefore, adding heuristics that encode some measure of plausibility and place value to guide the search is warranted.

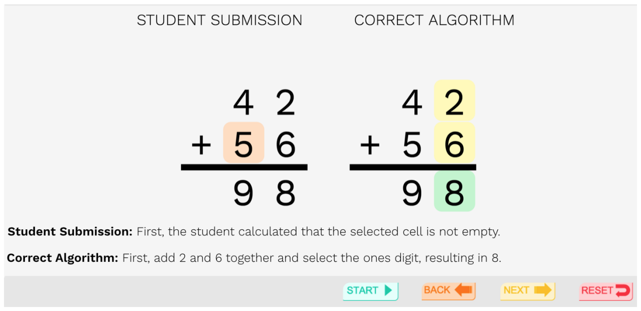
Limitations of the MetaMetrics data set

Students entered their answers on a computer, which omitted parts of the students' solution process that might normally be captured on paper, such as carry values or other intermediate calculations. Furthermore, this data set has not been assessed by a math education expert. This means that although we assigned each SPS a misconception(s), a math education expert may have chosen a different or additional misconception. We classified a SPS as "Accurate" if it modeled any of a possible set of misconceptions that fit the SPS.

Challenges of automatic thought process reconstruction

Our approach is less successful with SPSes that contain correct solutions. The SPSes with misconceptions that we evaluated either contain all incorrect solutions (fully incorrect) or a mix of correct and incorrect solutions (partially incorrect). Our system was more accurate on fully incorrect SPSes (57% were "Accurate") than partially incorrect SPSes (36% were "Accurate"). We believe our system's accuracy on partially correct SPSes was lower because it is generally harder for our system to identify misconceptions if not all problems in the SPS exhibit the misconception. This happens frequently with partially correct SPSes. Systematic procedural misconceptions are also significantly rarer to find in partially correct SPSes. For the MetaMetrics data, we identified only 22 out of 547 total partially correct SPSes as having a misconception.

Our approach assumes that a student solves each problem in a SPS in exactly the same manner. However, students are frequently not self-consistent; they make mistakes that are seemingly unrelated to a misconception or are complex combinations of many misconceptions. Differentiating between systematic and random errors is important future work.



Student Submission: First, the student calculated that the selected cell is not empty.

Correct Algorithm: First, add 2 and 6 together and select the ones digit, resulting in 8.

Figure 8. GUI Demonstration with a conditional event for the student and an update event for the correct algorithm

VISUALIZATION OF MISCONCEPTIONS TO EDUCATORS

This section presents the design and evaluation of a GUI that explains the output of our misconception identification system. We were motivated to build a visualization to make our complex system output (i.e. reconstruction programs) easily interpretable by educators. The challenge was to build an interface that could work for any problem type in K-8 math.

Preprocessing

To display a reconstruction program via the GUI, we had to apply the program to each individual problem in the SPS. From a computational perspective, a reconstruction program represents the student's behavior for *every* problem in the original SPS. However, this abstraction runs counter to how educators think about math problems. Educators are accustomed to assessing a student's work via a series of individual examples on a worksheet or exam. Furthermore, generating natural language descriptions of student misconceptions is a very challenging problem. We therefore build on recent work [38] that automatically generated step-by-step tutorials to explain procedures. To generate a step-by-step walkthrough of a reconstruction program we convert it into an *event sequence*. An event sequence is an application of a reconstruction program to one problem in a SPS, worked out step-by-step with loops unrolled and conditionals evaluated.

Comparing the Student Solution to the Correct Algorithm

In order to place a student's misconception in context, we generate a program and an event sequence for the correct algorithm as well and present it side-by-side with the student's (Fig. 8). When students solve a problem incorrectly, they may insert or delete several steps compared to the correct algorithm. To make the student solution advance with the correct algorithm, we needed to match up events in the student sequence with those in the correct sequence. To do this we compute an edit distance between two paired events using a heuristic based on the operator, operand locations, and the result location. We add an empty event if the distance is over a certain threshold. We add an empty event first to the shorter sequence and then, once the sequences equalize, we alternate.

Evaluation of the GUI

To evaluate the GUI (also called "the demonstration") we conducted a user study. The study was designed to compare the explanations provided by the GUI against expert descriptions of student misconceptions (Fig. 9). Our hypothesis was

A-W-3	"Carol misses examples in which one of the addends is written as a single digit. When working with such examples, she adds the three digits as if they were all units. When both addends are two-digit numbers, she appears to add correctly."
A-W-4	"She tries to use the regular addition algorithm; however, when she adds the tens column she adds in the one-digit number again."
E-F-3	Greg "considers the given numerator and denominator as two whole numbers, and divides the larger by the smaller to determine the new numerator (ignoring any remainder); then the largest of the two numbers is copied as the new denominator."
Cox	The student "subtracts the single digit of the subtrahend from both digits of the minuend."

Figure 9. Expert descriptions for all problems used in the user study. The Cox example is Table 6, Subtable 1, Misconception 3.

Q1	The demonstration accurately explains the student's misconception
Q2	The demonstration uses unambiguous language and terminology
Q3	The demonstration addresses every error that I found
Q4	The demonstration was easy to use

Figure 10. Rating questions asked for both the expert description and demonstration (our GUI). Users were asked to respond using a 5-point Likert scale. The ease of use question (Q4) was only asked for the GUI.

that we can explain students' misconceptions to an educator as successfully as a human expert.

To determine how well the GUI conveyed the student's misconception we asked participants to assess three different explanations. First, we had participants explain the misconception in their own words, which required them to identify and, hopefully, internalize the student's misconception. We then asked them to rate the expert description from the source material (Fig. 9) and the demonstration on a Likert scale individually using the first three and four questions, respectively, of Fig. 10. To prevent priming bias, we randomly determined whether the GUI or the expert description would come first. Finally, we had participants compare the expert description and demonstration directly (see questions in Fig. 12).

User Study Details

We chose 4 student misconceptions that exercise both the breadth and depth of our system for the study. The tested misconceptions were two addition examples from Ashlock [4] (**E1**: A-W-4 & **E2**: A-W-3, our running example), one subtraction misconception from Cox [12] (**E3**: Table 6, Subtable 1, Misconception 3), and a fraction reduction example from Ashlock (**E4**: E-F-3, Fig. 2, right). Reconstruction programs for the student's process for E2 and E4 contain conditionals and have event sequences of unequal lengths. E4 programs also contain conditionals for the correct algorithm. We did not use any data from MetaMetrics because we do not have expert descriptions available. The remaining two resources, Lankford's Report [26] and Van Lehn [46], do not provide full problem sets with their misconception descriptions.

Our recruitment goal was to reach as many US educators as possible. We recruited participants online through Reddit, Twitter, and Facebook and contacted multiple educators we knew. All participants were asked to sign-up for the study in advance and then the study was released to the participant pool for 2 weeks. 29 users completed the study in full, of which 17 self-reported as full-time K-12 educators. Participants were allowed to start the study and return at a later point. Four participants were not able to access the demon-

	μ_{demo}	SE_{demo}	μ_{expert}	SE_{expert}	**Mean Diff.**	**95% CI**	**BF**
E1, Q1	4.34	0.22	4.07	0.19	0.28	$(-0.33, 0.88)$	*0.29*
E1, Q2	3.79	0.27	3.14	0.24	0.66	$(0.03, 1.28)$	1.47
E1, Q3	4.38	0.19	4.03	0.22	0.34	$(-0.09, 0.78)$	*0.64*
E2, Q1	4.39	0.20	4.71	0.09	-0.32	$(-0.77, 0.13)$	*0.53*
E2, Q2	3.93	0.25	4.43	0.19	-0.50	$(-1.05, 0.05)$	*0.90*
E2, Q3	4.36	0.18	4.5	0.18	-0.14	$(-0.49, 0.20)$	*0.28*
E3, Q1	4.38	0.24	3.96	0.24	0.42	$(-0.13, 0.97)$	*0.62*
E3, Q2	4.50	0.15	4.31	0.21	0.19	$(-0.40, 0.78)$	*0.25*
E3, Q3	4.50	0.19	3.54	0.30	0.96	$(0.43, 1.49)$	35.55
E4, Q1	4.00	0.27	4.52	0.18	-0.52	$(-0.98, -0.06)$	1.86
E4, Q2	4.41	0.16	4.41	0.14	0.00	$(-0.32, 0.32)$	*0.20*
E4, Q3	4.41	0.18	4.24	0.20	0.17	$(-0.21, 0.55)$	*0.29*

Figure 11. Numerical results assessing examples E1 − E4 on questions Q1 − Q3 shown in Fig. 10. All data were collected on a 5-point Likert scale. Means (μ) are shown with standard error (SE) for the demonstration and the expert description. Mean Difference is defined as $\mu_{\text{demo}} - \mu_{\text{expert}}$ computed directly from raw, unrounded means, hence any variability. 95% confidence intervals are calculated for the mean difference. Bayes Factors (BF) are calculated using R standard priors.

	Demonstration	**Equivalent**	**Text**
What is more helpful to you as an educator?	50	23	39
Which do you find easier to use?	39	30	43
What do you think is more accurate?	34	61	17

Figure 12. Comparison data aggregated across E1-E4 (n = 112)

stration for a single example due to a server malfunction; we included their responses for the examples that worked.

Results

Our system can successfully explain misconceptions

Our data show that the demonstration is as good as the expert description at explaining the student's misconception to the user. We arrive at this conclusion by looking at the results of the individual rankings of the demonstration and expert description. The goal of our statistical analysis is to determine if there is support *for* the traditional null hypothesis ($H_0 : \mu_{\text{demo}} - \mu_{\text{expert}} = 0$). Such support would suggest there is no difference between the demonstration and the expert description; in other words, the demonstration explains the misconception as well as a human expert.

To determine if there is support for the null hypothesis, we calculated a series of Bayes Factors as recommended by Kaptein & Robertson [23]. We calculated Bayes Factor values using the R language standard priors for all combinations of questions (Q1-Q3) and examples (E1-E4). In addition, we calculated basic frequentist statistics, reported in Fig. 11.

The results of this analysis show that 9 out of the 12 Bayes Factors provide support (value < 1) for the null hypothesis. Using common cutoffs to interpret the size of the support [47], 5 values provide substantial support $\left(\frac{1}{10} \text{ to } \frac{1}{3}\right)$ and 4 values provide anecdotal support $\left(\frac{1}{3} \text{ to } 1\right)$. The 3 values that support a difference between the two treatments were computed for E1Q2 ($\mu_{\text{demo}} > \mu_{\text{expert}}$), E3Q3 ($\mu_{\text{demo}} > \mu_{\text{expert}}$), and E4Q1 ($\mu_{\text{demo}} < \mu_{\text{expert}}$).

Participants found the demonstration easy to use and as helpful and accurate as the expert description

We asked the participants to rate how easy the demonstration was for each example (Fig. 10, Q4). Overwhelmingly, they agreed or strongly agreed that the demonstration was easy to use ($n = 100$, 89%). As a companion to the individual ranking data, we also asked the participants to directly compare the demonstration with the expert description. The results, aggregated across examples, are presented in Fig. 12. These results are statistically significant for both helpfulness and accuracy ($\chi^2 = 9.875$, $p = 0.007$; $\chi^2 = 26.375$, $p = 1.87e-06$) with the caveat that the data are not necessarily independent.

Discussion

The most interesting trend in our results is that the expert description had higher mean scores on every question for E2 (which is our running example, A-W-3). We believe this is because E2 contains conditionals for its representative student program and the way booleans were translated from the program to the GUI was insufficient. For instance, one user specifically noted for E2: "The phrase "the student calculated that the selected cell is empty" is odd."

Users also ranked the expert description higher than the demonstration on accuracy (Q1) for E4. We believe multiple factors may have contributed. First, E4 is a fraction reduction misconception and thus the layout of the GUI differs considerably from the previous three examples. In addition, the misconception itself is, in our opinion and the opinion of some of our users, the most difficult or non-intuitive to identify of those included in the study. We therefore believe users may have been more confused when assessing the demonstration.

We received qualitative feedback from users indicating the potential use of our system in a classroom setting. One user wrote: "I could see both the text and animation being useful in different contexts, depending on who the target audience is and what their math background and strengths are."

CONCLUSION

We presented a system that automatically identifies incorrect procedural thought processes in K-8 mathematics. Our approach generates programs representing students' thought processes and explains misconceptions by visualizing those programs side-by-side with the correct algorithm. The evaluations in this paper focus on K-8 mathematics, but in future work we hope to evaluate whether our approach can generalize to other domains. We believe our mechanisms for thought-process reconstruction and visualization could work for math topics that involve deterministic step-by-step computations on a 2D spreadsheet. For example, if we were to add operators that transform variables and coefficients we could potentially extend our approach to solve linear equations. In future work, we will explore how our system can be used in classroom settings by integrating it into an online grading portal (see our website at: `https://goo.gl/DfyryD`).

ACKNOWLEDGEMENTS

We would like to thank our study participants, Audra Kosh and MetaMetrics for allowing us to use their data, and Wil Thomason for programming support.

REFERENCES

1. Rajeev Alur, Loris D'Antoni, Sumit Gulwani, Dileep Kini, and Mahesh Viswanathan. 2013. Automated Grading of DFA Constructions.. In *IJCAI*, Vol. 13. 1976–1982.
2. Erik Andersen, Sumit Gulwani, and Zoran Popovic. 2013. A trace-based framework for analyzing and synthesizing educational progressions. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 773–782.
3. John R Anderson, Albert T Corbett, Kenneth R Koedinger, and Ray Pelletier. 1995. Cognitive tutors: Lessons learned. *The journal of the learning sciences* 4, 2 (1995), 167–207.
4. Robert B Ashlock. 1990. *Error patterns in computation* (5 ed.). Simon & Schuster Books For Young Readers.
5. Lawrence Bergman, Vittorio Castelli, Tessa Lau, and Daniel Oblinger. 2005. DocWizards: a system for authoring follow-me documentation wizards. In *Proceedings of the 18th annual ACM symposium on User interface software and technology*. ACM, 191–200.
6. Stephen B Blessing. 1997. A programming by demonstration authoring tool for model-tracing tutors. *International Journal of Artificial Intelligence in Education (IJAIED)* 8 (1997), 233–261.
7. John Seely Brown and Richard R Burton. 1978. Diagnostic models for procedural bugs in basic mathematical skills. *Cognitive science* 2, 2 (1978), 155–192.
8. John Seely Brown and Kurt VanLehn. 1980. Repair theory: A generative theory of bugs in procedural skills. *Cognitive science* 4, 4 (1980), 379–426.
9. Kerry Shih-Ping Chang and Brad A. Myers. 2016. Using and Exploring Hierarchical Data in Spreadsheets. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. ACM, 2497–2507.
10. Pei-Yu Chi, Sally Ahn, Amanda Ren, Mira Dontcheva, Wilmot Li, and Björn Hartmann. 2012. MixT: automatic generation of step-by-step mixed media tutorials. In *Proceedings of the 25th annual ACM symposium on User interface software and technology*. ACM, 93–102.
11. Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. 2016. Programmatic and direct manipulation, together at last. *ACM SIGPLAN Notices* 51, 6 (2016), 341–354.
12. Linda S Cox. 1975. Diagnosing and Remediating Systematic Errors in Addition and Subtraction Computations. *Arithmetic Teacher* 22, 2 (1975), 151–157.
13. Jennifer Fernquist, Tovi Grossman, and George Fitzmaurice. 2011. Sketch-sketch revolution: an engaging tutorial system for guided sketching and application learning. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*. ACM, 373–382.
14. Floraine Grabler, Maneesh Agrawala, Wilmot Li, Mira Dontcheva, and Takeo Igarashi. 2009. Generating photo manipulation tutorials by demonstration. *ACM Transactions on Graphics (TOG)* 28, 3 (2009), 66.
15. Andrew Head, Elena Glassman, Gustavo Soares, Ryo Suzuki, Lucas Figueredo, Loris D'Antoni, and Björn Hartmann. 2017. Writing Reusable Code Feedback at Scale with Mixed-Initiative Program Synthesis. In *Proceedings of the Fourth (2017) ACM Conference on Learning@ Scale*. ACM, 89–98.
16. HeyMath! 2016. Online. (2016).
17. Vicki-Lynn Holmes, Chelsea Miedema, Lindsay Nieuwkoop, and Nicholas Haugen. 2013. Data-driven intervention: correcting mathematics students' misconceptions, not mistakes. *The Mathematics Educator* 23, 1 (2013).
18. Bernd Huber, Joong Ho Lee, and Ji-Hyung Park. 2015. Detecting User Intention at Public Displays from Foot Positions. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. ACM, 3899–3902.
19. Earl Hunt and Jim Minstrell. 1994. A cognitive approach to the teaching of physics. *Classroom Lessons: Integrating Cognitive Theory and Classroom Practice* (1994).
20. Matthew P Jarvis, Goss Nuzzo-Jones, and Neil T Heffernan. 2004. Applying machine learning techniques to rule generation in intelligent tutoring systems. In *Intelligent Tutoring Systems*. Springer, 157–178.
21. Garvit Juniwal, Alexandre Donzé, Jeff C Jensen, and Sanjit A Seshia. 2014. CPSGrader: Synthesizing temporal logic testers for auto-grading an embedded systems laboratory. In *Proceedings of the 14th International Conference on Embedded Software*. ACM, 24.
22. Shalini Kaleeswaran, Anirudh Santhiar, Aditya Kanade, and Sumit Gulwani. 2016. Semi-supervised verified feedback generation. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 739–750.
23. Maurits Kaptein and Judy Robertson. 2012. Rethinking statistical analysis methods for CHI. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1105–1114.
24. J Richard Landis and Gary G Koch. 1977. The measurement of observer agreement for categorical data. *biometrics* (1977), 159–174.
25. Pat Langley and Stellan Ohlsson. 1984. Automated Cognitive Modeling.. In *AAAI*. 193–197.

26. Francis Lankford, Jr. 1972. *Some Computational Strategies of Seventh Grade Pupils*. Technical Report. U.S. Department of Health, Education, and Welfare, University of Virginia. http://babel.hathitrust.org/cgi/pt?id=mdp.39015035510141; view=1up; seq=3

27. Tessa Lau, Steven A Wolfman, Pedro Domingos, and Daniel S Weld. 2003. Programming by demonstration using version space algebra. *Machine Learning* 53, 1-2 (2003), 111–156.

28. Björn B Levidow, Earl Hunt, and Colene McKee. 1991. The DIAGNOSER: A HyperCard tool for building theoretically based tutorials. *Behavior Research Methods, Instruments, & Computers* 23, 2 (1991), 249–252.

29. Nan Li, William Cohen, Kenneth R Koedinger, and Noboru Matsuda. 2011. A machine learning approach for automatic student model discovery. In *Educational Data Mining 2011*.

30. Noboru Matsuda, William W Cohen, Jonathan Sewall, Gustavo Lacerda, and Kenneth R Koedinger. 2007. Predicting students' performance with simstudent: Learning cognitive skills from observation. *Frontiers in Artificial Intelligence and Applications* 158 (2007), 467.

31. Noboru Matsuda, Andrew Lee, William W Cohen, and Kenneth R Koedinger. 2009. A computational model of how learner errors arise from weak prior knowledge. In *Proceedings of the Annual Conference of the Cognitive Science Society, Austin, TX*. 1288–1293.

32. Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Oleksandr Polozov, Rishabh Singh, Benjamin Zorn, and Sumit Gulwani. 2015. User interaction models for disambiguation in programming by example. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*. ACM, 291–301.

33. McGraw-Hill Education. 2016. Thrive. Online. (2016).

34. MetaMetrics, Inc. 2017. MetaMetrics: Bringing meaning to measurement by matching students to resources using a scientific, universal scale. (2017). https://metametricsinc.com/.

35. Tom M Mitchell. 1982. Generalization as search. *Artificial intelligence* 18, 2 (1982), 203–226.

36. Brad A Myers, David A Weitzman, Andrew J Ko, and Duen H Chau. 2006. Answering why and why not questions in user interfaces. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*. ACM, 397–406.

37. Jeffrey Nichols and Tessa Lau. 2008. Mobilization by Demonstration: Using Traces to Re-author Existing Web Sites. In *Proceedings of the 13th International Conference on Intelligent User Interfaces*. ACM, 149–158.

38. Eleanor O'Rourke, Erik Andersen, Sumit Gulwani, and Zoran Popović. 2015. A Framework for Automatically Generating Interactive Instructional Scaffolding. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. ACM, 1545–1554.

39. Kelly Rivers and Kenneth R Koedinger. 2015. Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. *International Journal of Artificial Intelligence in Education* 27, 1 (2015), 37–64.

40. Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning syntactic program transformations from examples. In *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 404–415.

41. Arjun Singh, Sergey Karayev, Kevin Gutowski, and Pieter Abbeel. 2017. Gradescope: A Fast, Flexible, and Fair System for Scalable Assessment of Handwritten Work. In *Proceedings of the Fourth (2017) ACM Conference on Learning@ Scale*. ACM, 81–88.

42. Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated feedback generation for introductory programming assignments. *ACM SIGPLAN Notices* 48, 6 (2013), 15–26.

43. Masamichi Sison, Raymund an2d Shimura. 1998. Student modeling and machine learning. *International Journal of Artificial Intelligence in Education (IJAIED)* 9 (1998), 128–158.

44. Piyawadee Sukaviriya. 1988. Dynamic construction of animated help from application context. In *Proceedings of the 1st annual ACM SIGGRAPH symposium on User Interface Software*. ACM, 190–202.

45. Piyawadee Sukaviriya and James D Foley. 1990. Coupling a UI framework with automatic generation of context-sensitive animated help. In *Proceedings of the 3rd annual ACM SIGGRAPH symposium on User interface software and technology*. ACM, 152–166.

46. Kurt VanLehn. 1990. *Mind bugs: The origins of procedural misconceptions*. MIT press.

47. Ruud Wetzels, Dora Matzke, Michael D Lee, Jeffrey N Rouder, Geoffrey J Iverson, and Eric-Jan Wagenmakers. 2011. Statistical evidence in experimental psychology: An empirical comparison using 855 t tests. *Perspectives on Psychological Science* 6, 3 (2011), 291–298.