An Interaction Design for Machine Teaching to Develop AI Tutors

Daniel Weitekamp III
Carnegie Mellon University
Pittsburgh, USA
weitekamp@cmu.edu

Erik Harpstead
Carnegie Mellon University
Pittsburgh, USA
harpstead@cmu.edu

Kenneth R. Koedinger
Carnegie Mellon University
Pittsburgh, USA
koedinger@cmu.edu

ABSTRACT

Intelligent tutoring systems (ITSs) have consistently been shown to improve the educational outcomes of students when used alone or combined with traditional instruction. However, building an ITS is a time-consuming process which requires specialized knowledge of existing tools. Extant authoring methods, including the Cognitive Tutor Authoring Tools' (CTAT) example-tracing method and SimStudent's Authoring by Tutoring, use programming-by-demonstration to allow authors to build ITSs more quickly than they could by hand programming with model-tracing. Yet these methods still suffer from long authoring times or difficulty creating complete models. In this study, we demonstrate that Simulated Learners built with the Apprentice Learner (AL) Framework can be combined with a novel interaction design that emphasizes model transparency, input flexibility, and problem solving control to enable authors to achieve greater model completeness in less time than existing authoring methods.

Author Keywords

Simulated Learners; Interaction Design; Programming-by-Demonstration; Machine Teaching; Intelligent Tutoring Systems

CCS Concepts

•**Human-centered computing** → **Graphical user interfaces;** *User studies;* •**Software and its engineering** → **Programming by example;**

INTRODUCTION

Intelligent tutoring systems (ITSs) are a type of computerized educational technology which tutor students through scaffolded practice problems and provide correctness feedback, next-step hints, and adaptive feedback messages [27]. ITSs also typically track student knowledge at a granular level to intelligently pick practice problems that will help students learn new skills [5]. In several studies ITSs have been shown to benefit learners when used alone or in combination with traditional instruction [7, 24, 25, 12].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CHI '20, April 25–30, 2020, Honolulu, HI, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6708-0/20/04 ...$15.00.
http://dx.doi.org/10.1145/3313831.3376226

ITSs are, however, notoriously difficult and time consuming to build [22]. For example, consider the case of an author creating and ITS to teach multi-column addition. To build a model-tracing version of this ITS, a type of ITS built by programming production rules that encode a space of correct student solution paths, a programmer would need to write production rules for adding numbers, carrying 1's digits and putting their answers into appropriate interface fields. Typically this process takes 200-300 hours of developer time per hour of instruction time. The Cognitive Tutor Authoring Tools (CTAT) introduced example-tracing as an alternative method of ITS authoring, which does not require programming and reduces authoring time down to 50-100 hours per hour of instruction [2]. ITS authors who use example-tracing can program the behavior of an ITS by simply demonstrating all of the correct action(s) that can be taken at each step of a problem. For example, an instructional designer could make a single multi-column arithmetic problem with example-tracing by simply setting up and solving a problem every possible way in order to generate a behavior graph, a directed graph specifying all possible solution paths. However, example-tracing is often not the best tool when it comes to problems with complex behavior or highly variable solution spaces, such as in a complex algebra problem [16]. Additionally, with CTAT example-tracing scaling from a single problem to several problems (termed mass production) is still often a time consuming process which requires programming Excel spreadsheets [1].

CTAT exhibits a very simple form of programming-by-demonstration (PBD), a broad class of machine learning techniques which compose programs capable of replicating user demonstrations [6, 23]. Traditionally, PBD is used to help users automate repetitive tasks. The simplest PBD systems, like CTAT example-tracing, reapply the exact actions demonstrated by users. More complex systems contain a learning agent capable of inducing generalized programs from user demonstrations. These induced programs can then be reapplied in new situations at the user's command [10, 11]. Other PBD systems additionally learn the conditions under which each of several induced programs should be applied, allowing for the creation of complex interactive applications [19].

Some systems use a broader set of interactions beyond demonstration to learn from users [9, 14]. In this work, we refer to these approaches as machine teaching. In machine teaching systems the user may reinforce or correct an agent's actions by providing clarifying demonstrations or correctness feedback [14]. For example, in the Gamut system users could build

whole games by demonstrating and correcting the movements of various objects [19]. Another system which operates by machine teaching is SimStudent, a computational model of human learning which learns to solve problems in an ITS [18]. For the purposes of ITS authoring SimStudent could be used to interactively author ITS behavior in conjunction with CTAT using an interaction design called Authoring by Tutoring [17]. One of the goals of Authoring by Tutoring was to further reduce ITS authoring times beyond the reduction achieved by CTAT example-tracing, while at the same time reintroducing some of the generality of model-tracing tutors.

In general, AI agents which mimic the inductive learning process undergone by students are known as Simulated Learners. Following the work of SimStudent, the Apprentice Learner (AL) Framework was created to build Simulated Learners similar to SimStudent, but modularized by their several interworking learning mechanisms [15]. Collectively these learning mechanisms generate skills in the form of production rules. Through user demonstrations and correctness feedback these learning mechanisms inductively refine the rules specifying the conditions that will cause a skill to fire and the way in which it will fire. Each learning mechanism in the AL framework can be changed in and out to create unique agents primed to serve particular purposes. For example, the desired efficacy with which an agent learns might differ by use case. For the purposes of ITS authoring one would want an AL agent to learn as much as possible from each interaction to limit the tedium of the authoring process. By contrast for the purposes of student modeling agents would be taught without a human in the loop against an existing tutoring system, and the ideal agent would learn at the same, relatively slow, rate per opportunity evident in the logs of human students working in the same tutoring system.

Simulated Learners like SimStudent and AL agents can be evaluated differently depending on their intended purposes [8]. For example, for the goal of producing accurate models of human learning one might assess whether a Simulated Learner is able to reach mastery from similar activities as human learners. SimStudent and AL agents have both been demonstrated to reach mastery performance at solving problems in ITSs [15]. However, for the purpose of acting as an ITS authoring tool, mastery performance is not a sufficient condition for success. An ITS must be able to check the correctness of any possible student input, not only produce a correct input. Instead, a Simulated Learner used to drive an ITS should exhibit **model-tracing completeness**, defined as recognizing all intended correct actions as correct, and no incorrect action as correct for all possible states in a problem's intended solution space. If a Simulated Learner achieves mastery performance in an ITS, then it can at least recognize a particular correct solution path through a problem. However, only a Simulated Learner exhibiting the stronger condition of model-tracing completeness can recognize all incorrect actions as incorrect, and support every solution path intended by the ITS author.

To date, no Simulated Learner has been well suited to the task of achieving model-tracing completeness. As we will demonstrate in this work, this failure is not due to technical limitations of existing Simulated Learners, but to the interaction techniques used to train them [14]. At the heart of the issue is the fact that current interaction techniques including SimStudent's Authoring by Tutoring treat the ITS author like they are an ITS and treat the Simulated Learner like a real human student using that ITS. However, there is no inherent requirement that a Simulated Learner must learn like a human [26], and framing interactions this way limits the sort of feedback that users can give in training the system. For example, if the ITS author is required to act like an ITS to the Simulated Learner than when any correct action is proposed they can only mark that action as correct. Such an interactions design would not afford, for example, the opportunity to demonstrate alternate solution paths, or query for and correct other actions the Simulated Learner believes can lead to a solution. Ultimately, this perspective prevents authors from demonstrating full solution spaces for different types of problems and checking them for completeness. In order to address this issue we have created a novel authoring interface for AL agents designed to be more transparent and flexible than previous interfaces.

In this study we test the new interaction design of our AL authoring interface with 10 instructional design students with varying backgrounds. The objective of our study is to assess the degree to which our interaction design supports more efficient authoring of model-tracing complete ITSs compared to CTAT example-tracing. The contributions of this work are:

1. A novel interaction design for authoring Intelligent Tutoring Systems using Simulated Learners that emphasizes model transparency, input flexibility, and problem solving control.

2. A user study demonstrating the efficacy of this interaction design toward training Simulated Learners that exhibit model-tracing completeness.

3. Design recommendations for future Simulated Learner based ITSs authoring tools.

THE APPRENTICE LEARNER FRAMEWORK

Apprentice Learner agents are Simulated Learners comprised of sets of modular interconnected learning mechanisms [15]. Each learning mechanism serves a particular role in skill induction or refinement. A skill is a collection of learning mechanism instances and a production rule induced by those learning mechanism instances. A production rule consist of a left-hand side which specifies the conditions sufficient for it to fire, and a right-hand side which specifies what occurs when it fires [3]. The left-hand side of each skill is learned by the *where-learning* and *when-learning* mechanisms. Similarly, the right-hand side is induced and refined, by the *how-learning* mechanism. There is also a *which-learning* mechanism which is a conflict resolution strategy for choosing which skill to fire should multiple skills have their *where* and *when* conditions satisfied at a particular step in problem solving.

A single AL agent uses different machine learning algorithms for each of its various learning mechanisms. For example, the *where-learning* mechanisms learns the conditions for matching interface elements pertaining to the application of a skill by inductive logic programming [21]. These conditions that the

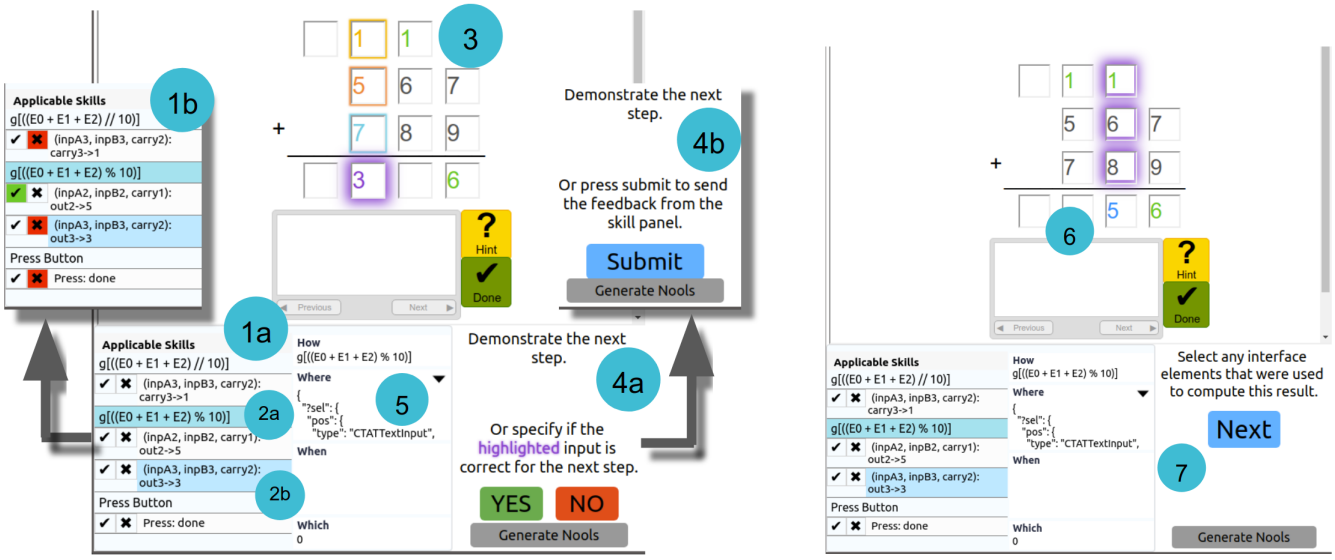

Figure 1: Screenshots of our AL authoring interface when skill applications are proposed (left) and after a user demonstration (right). 1a) The skill window with several proposed skill applications. 1b) The skill window with ✖ and ✔ buttons toggled. 2a) The selected skill and its formula. 2b) The selected/staged skill application. 3) The tutor interface with *where* match highlighted. 4a) The Yes and No button dialog. 4b) The Yes and No buttons are replaced with a Submit button which when clicked will send the ✖ and ✔ button feedback encoded in 1b. 5) Debug information describing the state of the selected skill's learning mechanisms. Not intended for users. 6) The state of the tutoring system after a user has made a demonstration (blue 5) and selected foci-of-attention (purple glow). 7) Yes and No buttons replaced with Next button.

where-learning mechanism learns match to both the interface element where a skill will be applied and the interface elements from which the value of that skill application is derived. In this study we use a variation of the version space algorithm for the *where-learning* mechanism [20]. The *when-learning* mechanism determines the conditions when a skill should be applied. *When-learning* mechanisms can be implemented using any form of binary-classification algorithm. For example, in this study we use the decision tree algorithm [4]. The *how-learning* mechanism induces and refines the behavior that is applied when a skill fires. The agents in this study use a search-based planner to form one or more formulas that explain each user provided demonstration. This planner searches over a number of numerical functions such as addition and division by ten which the agent is provided as prior knowledge.

An important feature of the AL agents used in this study is that their *where-learning* and *when-learning* mechanisms support skills that are not tied to particular interface elements. Skills can be applied between steps in an interface and even between interfaces. Skills are applied when a particular set of interface elements match learned *where* and *when* patterns.

AL agents support two primary functions, *request*, which returns a set of actions that an agent thinks it can take given a tutoring interface state, and *train*, which engages the agent's learning mechanisms from a state-action pair and reward. In this study we also support a *train_explicit* function which uses a skill application instead of an action for fitting. A skill application is associated with a particular instance of a skill firing while an action is not. For example, two skill applications from two different skills might apply the same action. *Train* differs from *train_explicit* in that *train* gives feedback to all skills that could have produced an action. Readers may find additional details in the following prior publications [15] [13].

Prior to this study AL agents learned interactively from the user via the same interaction design used in SimStudent's Authoring by Tutoring. In Authoring by Tutoring when an agent produces an action in the interface it requests for correctness feedback from the user, and if the agent cannot produce an action it asks for a demonstration of correct behavior. When users provide demonstrations they also specify the interface elements from which that action was computed to make skill induction less ambiguous.

INTERACTION DESIGN

Most PBD systems, including those that use the broader set of machine teaching interactions [14] like SimStudent, put the user in the perspective of teaching an agent as it takes steps along a path through a problem, to ensure that the agent produces correct actions. We refer to this perspective as the *performance-model perspective*, as its aim is to ensure that an agent performs correctly along some path to a goal.

When specifying the behavior of an ITS one must define behavior that can respond to the whole space of possible student solutions. Teaching an agent to understand this entire space is

a more complex machine teaching task than teaching an agent to perform correctly through just a single solution path per problem. When using Simulated Learners for ITS authoring we would prefer for them to achieve model-tracing completeness, defined as the ability to identify all intended correct next actions as correct, and all other actions as incorrect for all possible problem states. Importantly, the condition of model-tracing completeness can be applied to any tutoring system given an intended set of behaviors regardless of the underlying implementation of the tutoring system (example-tracing or model tracing). Model-tracing completeness is an evaluation of the behavior of the tutoring system thus all systems which operate as intended are equally model-tracing complete. An ITS author can evaluate the local model-tracing completeness of each step of a problem, by simply considering whether or not the set of actions currently allowed by the tutoring system at that step agree with their understanding of how the tutoring system should work.

Our novel interaction design seeks to support users in taking a *model-tracing completeness perspective* where users evaluate agents on the set of all skill applications an agent produces at each step instead of on a single sufficient action that the agent chooses at each step. To support this perspective our new interaction design makes skill applications transparent to the user, is flexible to a wide range of user interactions, and gives the user more control over the problem solving process.

Model Transparency

From the *performance-model perspective* a user only needs to ensure that an agent produces a correct action at each step in a problem. However, for a user to confidently train a Simulated Learner to manifest model-tracing completeness they must be able to see whether or not the agent can produce all intended correct actions at each step. In order to give the user access to this information we created a skill window (Figure 1.1a), which lists all of the skills that an agent believes can be applied on the current step.

This skill window provides users with considerably more information than they would see in a typical *performance-model perspective* interaction design. Instead of a single action the user sees all skill applications that the agent considers correct for a given step (Figure 1.1a). This allows them to assess not just whether the agent would have taken a correct action at that step, but whether the agent exhibits complete model-tracing behavior for that step. In seeing skill applications, the user is made aware of information concerning how the agent's associated actions came about. This includes the formula induced by the agent on the first application of each skill (Figure 1.2a) and the particular interface elements matched by the conditions learned by the *where-learning* mechanism for each proposed application of the skill (Figure 1.2b). The *where* match includes the interface element that the action would be taken on and the elements used as arguments for the formula from which the value of that action was derived. For example, in Figure 1.1a, a skill application is selected with formula: "(E0 + E1 + E2) % 10" (i.e., take the modulus 10 of the sum of elements E0-E2) and a *where* match consisting of the arguments inpA2, inpB2, carry1 and the selection element out2. The user can also see that the formula evaluated on these interface elements yields 5.

It is important to note that the skill applications displayed in the skill window are not representations of the underlying AL agent's AI or its derived production rules. The displayed skill applications are simply the firing of the AL agent's derived skills and are comprised only of an action, highlights indicating the values from which that action was computed and a mathematical formula. Thus, the information presented to the user requires no additional expertise to understand beyond knowledge of the particular domain being covered by the tutoring system. No checking of the underlying program or knowledge of AI is required by the user. The skill window adds transparency beyond extant interaction designs for Simulated Learners in that it provides an interactive display of all possible next actions that a Simulated Learner currently believes are correct.

By clicking each proposed skill in the skill window users can select each skill application as the staged skill application. When a skill application is staged (Figure 1.2b) its *where* match elements are highlighted in various colors (Figure 1.3). The state change caused by the action is prominently highlighted in purple, and the argument elements are highlighted less prominently in several other unique colors.

Input Flexibility

When a proposed skill application is staged the user can give it positive or negative feedback by pressing the Yes or No buttons respectively (Figure 1.4a). When Yes is pressed the action is applied putting the tutoring system in a new state, then a new set of skill applications are proposed by the agent and displayed in the skill window. Authoring by Tutoring had equivalent positive and negative feedback buttons, however our interaction design lets users give feedback to any of the skill applications proposed by the agent by staging them from the skill window. Additionally, our interaction design allows users to directly give feedback to all proposed skill applications at once by toggling the ✖ and ✔ buttons associated with each item in the skill window (Figure 1.1b). When any of these toggle buttons are selected the Yes and No buttons are replaced with a Submit button which, when pressed, sends all of the positive and negative feedback encoded in these toggle buttons to the agent (Figure 1.4b). Since the next state that the user wants to enter into may be ambiguous (e.g., there are multiple skill applications that have been marked as correct), pressing the submit button does not apply any of the associated actions to change the state of the tutoring system.

These new interactions give users the ability to provide a Simulated Learner with the feedback necessary to construct a complete model-tracing model. At each step the user can give feedback on all proposed skill applications instead of on only the skill application which the agent would choose to execute. By seeing and responding to all proposed actions the user can be more confident that a particular step exhibits complete model-tracing behavior.

Problem Solving Control

By choosing which skill applications to give feedback on, the user can also choose what states the tutoring system enters into. If Simulated Learners were to always choose a single skill application for a user to give feedback on then it would also be choosing the state the tutoring system would go into for the next round of feedback. When this is the case, the paths through problems that the user can test and give feedback on are heavily dependant on the Simulated Learner's conflict resolution strategy (i.e. *which-learning* mechanism). By default AL agents prioritize skills which have received the greatest proportion of positive feedback, meaning the first applicable skill is often the same for similar states. SimStudent's default behavior by contrast was to choose a single skill randomly among applicable skills [18]. Our interaction design reduces the role that an agent's conflict resolution plays in training. A user is allowed to override a Simulated Learner's default action by staging any skill application they choose. Consequently, users can make informed decisions about where to give the Simulated Learner feedback.

In addition to being able to give correctness feedback to the skill applications proposed by a Simulated Learner, our system allows users to demonstrate actions at any point in authoring regardless of whether or not the Simulated Learner has suggested an action (Figure 1.6). When demonstrating, users directly take actions in the interface. Giving the user this freedom means that they can set about building a complete solution space for a problem domain without being constrained to choosing among the Simulated Learner's induced skills at each step. Combined with the transparency to see all applicable skills at each step this flexibility orients the interaction design toward supporting a *model-tracing completeness perspective* as opposed to a *performance-model perspective*.

METHODS

In order to test whether our interaction design supports users in making model-tracing complete ITSs we had 10 participants author three-digit multi-column addition problems using our new AL authoring interface, and CTAT example-tracing interface. All participants were masters or PhD students studying educational technology. Users were paid $30 per hour for a total of an hour and a half to two hours. 8 participants were familiar with the CTAT authoring tools and 2 were not. 4 participants did CTAT first, and the rest used the AL interface first.

Participants were given the same ready made HTML interface (Figure 2) to use with both authoring modes. Each participant was asked to author each problem in a set of 11 specially selected multi-column addition problems (e.g., 543 + 678 = 1221). At each step in a multi-column addition problem, all of the numbers in a column may or may not add to more than 10, requiring the ten's digit to be carried to the next column. Our 11 problems were selected such that the solutions for the first 8 problems exhibited each of the 8 carry patterns possible in three-digit addition problems. The last three problems captured situations where the carry pattern would be incorrect if a student had forgotten to add the carry from the previous column (Figure 2b). We used our first two participants to

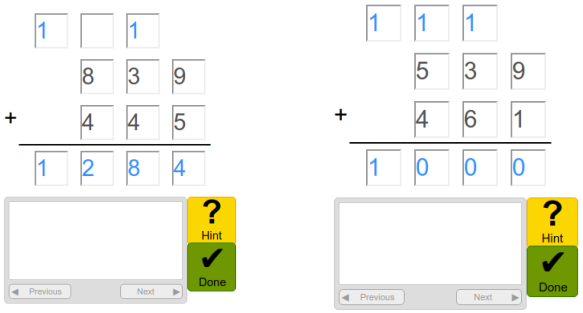

Figure 2: The multi-column addition tutor interface completed on two problems with different carry patterns. a) The middle carry is not present. b) All carries are present. Two columns exhibit a special case where they add to 10 only if the carry derived from the previous column is added.

estimate the appropriate amount of time to give each user. These two participants primarily used one authoring type, and spent little or no time with the other type. The remaining eight users spent at most 45 minutes authoring with each of the two tools. For all participants, one of the authors was available to provide guidance and answer questions on both authoring interfaces.

When using CTAT example-tracing, participants were asked to first make a behavior graph (Figure 3) to handle each of the eight carry patterns, and then for each one proceed as if they were going to use CTAT's mass production feature to make dozens of practice problems of each type. For novice users, we explained each of the steps involved, and demonstrated steps where necessary. The steps for mass producing a CTAT example-tracing tutor include demonstrating multiple solution paths to create a behavior graph, replacing the values for each edge in the demonstrated behavior graph with variables, and creating a problem table complete with Excel spreadsheet equations. In practice, a user might then use this spreadsheet to mass produce a large number of problems of a particular form. We only asked users to replicate the given problem and another of their own creation manifesting the same carry pattern. Participants were encouraged to test their mass produced problems to make sure that their formulas and choice of problems were correct and consistent with the carry pattern for the provided problem.

When using the AL authoring interface users were first given a brief demo of the interface. Then the agent was reset and the participants started again from scratch. Participants entered each of the provided problems into the interface one at a time, and then demonstrated steps and provided correctness feedback to train an AL agent. After going once through the provided problems participants continued redoing problems from those provided or of their own choosing until they felt that the agent had achieved full model-tracing completeness. At the point when the participants decided that they were finished or at the 45 minute mark we used a grading script to assess the

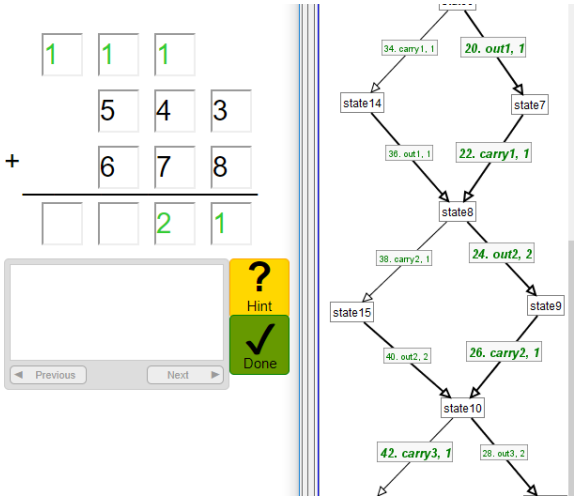

Figure 3: CTAT example-tracing. An intermediate solution state (left), and part of a behavior graph (right) specifying legal actions at each step. The graph branches so that adding and placing carries can be done in either order.

User	*First*	*AL Complete*	*AL Time*	*CTAT Complete*	*CTAT Time*
1	CTAT	30%	15min	11%	45min
2	AL	85%	55min	N/A*	N/A*
3	AL	98%	45min	13%	45min
4	CTAT	83%	30min	7%	45min
5	CTAT	92%	42min	50%	45min
6	AL	91%	25min	27%	45min
7	CTAT	92%	35min	11%	45min
8	AL	98%	41min	23%	45min
9	AL	85%	41min	13%	30min**
10	AL	99%	45min	23%	45min
Average		85%	37min	20%	43min
Median		92%	41min	13%	45min

Table 1: Performance of AL agents for each user by proportion of grader problems where the agent's skill applications were model-tracing complete (AL Complete). In addition, completeness progress in CTAT (CTAT Complete) and total times for both authoring types. *User 2 had to leave before they had the opportunity to start with CTAT. **User 9 did not enjoy using CTAT and elected to leave early.

model-tracing completeness of the user's trained Simulated Learner. The grading script measured the proportion of model-tracing complete steps in the 11 given problems, in addition to 12 problems from a holdout set with a diversity of carry patterns. As participants worked, a researcher occasionally intervened to remind them of the criteria for reaching 100% completeness. There was no formal condition for intervening; however, we typically intervened whenever the participant had questions, finished going through all the problems for the first time, and at any point where they seemed uncertain about the behavior the ITS was intended to exhibit. For example, we often reminded participants that if a carry was absent then the box should be left blank (instead of for example inserting a zero). Our intention in choosing to intervene was to be able to get a qualitative sense of how having varying degrees of mastery with the interface might change a user's interaction with the software. By partially scaffolding the participants' experience in this way we could see them progress from novice to expert users.

RESULTS

Our goal was to test whether our novel interaction design would enable users to train a Simulated Learner exhibiting full model-tracing completeness with three-digit multi-column arithmetic problems in much less time than it would take them to directly author the eight mass produced behavior graphs necessary to completely build problems in this domain with CTAT example-tracing.

Quantitative

The results of our user study are summarized in Table 1. For the AL authoring interface, our grading script measured completeness as the proportion of legally enterable states in which a tutor's behavior is model-tracing complete (i.e. only marks the intended correct next actions as correct). The states graded by our script included all of the unique steps along each of the legal solution paths in the 11 given problems and in the 12 problems in the holdout set. Completeness for the CTAT authoring mode is measured, more leniently, as the total progress the user made in producing the eight mass production ready behavior graphs. Each complete behavior graph and set of mass production formulas counted toward 1/16th of total completeness, and partially completed graphs, and Excel formulas received partial credit. For example, in the course of 45 minutes, users 1 and 7 completed an example tracing graph for one of the carry patterns and made it half way through writing mass production formulas for it in Excel, yielding a total of 11%, just short of the 13% (rounding up from 12.5%) which they would have earned had they fully completed one of the eight carry patterns. Although model-tracing completeness is measured differently between conditions, 100% model-tracing completeness means the same thing for both model-tracing and example-tracing tutors—that the tutor exhibits the intended correct behavior along all solution paths for all problem types.

The average model-tracing completeness achieved by users with AL across conditions was 85%. Excluding user 1, who had to leave after working for just 15 minutes, the average completeness was 91%. By contrast users achieved an average of 20% completeness with CTAT which corresponds to demonstrating, variablizing, and mass producing one behavior graph, then demonstrating and variablizing a second graph but not beginning the Excel spreadsheet to mass produce it. There was no appreciable difference in performance on our interface between users who were and were not familiar with CTAT. Additionally, there was no appreciable difference between users who used CTAT first or the AL interface first. Estimating completeness progress to be linear with respect to

time, participants achieved an average of 2.2% completeness per minute with AL and .5% completeness per minute with CTAT example-tracing. This constitutes slightly more than a 4-fold speedup in authoring time for equivalent levels of completeness. No participants reached 100% model-tracing completeness in either interface; however, three participants achieved completeness scores within 2% of full completion.

These results show us that although users can achieve greater levels of completeness using AL agents with our interaction design than with CTAT example-tracing, they fall short of building fully complete model-tracing behavior. The majority of users believed that they had reached full model-tracing completeness before their 45 minutes were up. Our own internal tests among ourselves show it is possible to achieve 100% model-tracing completeness in the course of about 20 minutes. Even though our users did not reach 100%, we found that they were supported in approaching full model-tracing completeness. Had users been constrained to authoring from a *performance-model perspective*, like in SimStudent's Authoring by Tutoring, they could have achieved at most 55% model-tracing completeness since 45% of the states in the holdout set can only be reached by supporting alternate paths. However, all users that authored for 45 minutes or up to the point that they believed they had achieved model-tracing completeness reached at least 83% model-tracing completeness.

Qualitative

In addition to the quantitative measures we also informally observed some trends among participants. When using CTAT, several users encountered a few common interaction difficulties. Many users had a hard time telling the difference between edges (denoting actions) and nodes (denoting problem states) in behavior graphs (In Figure 3 edges are shown in green text while nodes are in black). They would click an edge hoping to navigate to a state, but nothing would happen. A few participants who were more familiar with CTAT became sidetracked by trying to use features which they had once used successfully but no longer remembered how to use. These features included CTAT's formula function language (an alternative approach to using Excel equations), and creating unordered groups of steps. One user reported that it was difficult to fix errors with CTAT because the process of mass production required using two different applications (Excel and CTAT). Although the majority of our users were familiar with how to use CTAT, we often had to intervene to remind them of how to do certain steps.

Likewise, when working with the AL authoring interface users generally needed guidance in order to help them recognize model-tracing completeness at each step, and provide feedback and demonstrations when it had not been achieved. Although our initial demo covered these points, participants' initial behavior often indicated that they did not fully grasp the completeness objective. For example, some participants fell into a pattern of only giving feedback on the first skill application proposed by the AL agent. Other participants forgot to demonstrate steps if not all of the intended correct actions were proposed. These behaviors are indicative of the *performance-model perspective*. When participants consistently showed one of these behaviors, we helped them to understand how to use the interface to reach completeness through the *model-tracing-completeness perspective*. The fact that we consistently needed to intervene means that our interaction design could use some improvements to better induce this perspective in first time users. We discuss this further in the future work section.

There were a few patterns that we observed among participants. Most participants began interacting with the interface by only responding Yes or No to the first proposed skill application. However, after we reminded participants that they needed to give negative feedback to erroneous proposed skill applications, they tended to interact in one of two ways. Either participants would stage skills and press Yes or No, or they would primarily use the ✖ and ✔ buttons. When primarily using the Yes and No buttons users would sometimes give positive feedback before responding with negative feedback causing them to change the problem state without achieving model-tracing completeness on the step they were in. When participants predominantly used the ✖ and ✔ buttons they tended to give feedback to all of the proposed skill applications even if they were both correct. However, since only the Yes button could be used to navigate forward through a problem this meant they usually gave redundant feedback. In both these cases the fact that the Yes button was used for both navigation and feedback was associated with user errors or inefficiencies.

Generally, participants reported that they enjoyed working with the AL agent more than CTAT example-tracing. Some participants even thought training the AL agent was fun. A few participants reported that although the initial authoring steps were difficult it was rewarding to see the AL agent learning from their feedback to the point that they usually just had to click Yes at each step. Participants felt more confident that the progress that they had made with CTAT was complete and found that although it was relatively quick to train an AL agent it was hard to be certain of when the agent had reached completeness.

DISCUSSION

The fact that our interaction design supported users in reaching model-tracing completeness is a promising result. Although there were some limitations to our study, we believe these results have important implications for PBD and ITS authoring.

Limitations

In this study we only tested our interaction design on a single type of arithmetic problem. Multi-column addition was chosen because it is a type of problem with multiple solution paths which can be authored with both CTAT example-tracing and a Simulated Learner. We would, however, would like to see if these results extend to other types of problems. In practice Simulated Learners can induce a wider range of behaviors than are easily achievable with example-tracing [13]. We did not flex this functionality. Additionally we did not test our AL agents' abilities to generalize to problems of different sizes. For example, if an agent achieves model-tracing completeness on three-digit problems it should in principle

be model-tracing complete on two-digit, four-digit, and larger addition problems.

Our users were restricted to current educational technology students whereas in future work we would like to work with a greater diversity of users, including teachers and professional instructional designers. We would have also liked to test our users' progress toward model-tracing completeness at every step rather than only at the end. This would have helped us measure places where their progress slowed. Additionally, it would be interesting to see how users fare with our interaction design with considerably less scaffolding—no predefined problem sequence, and no interventions.

Implications For ITS Authoring

Efficient, easy-to-use ITS authoring tools backed by Simulated Learners have the potential to have a large impact on making ITSs widely available inside and out of the classroom. Although our participants were relatively skilled with the technologies involved, we believe that the significant speedup we observed in authoring times relative to CTAT example-tracing bodes well for future studies of both teachers and instructional designers. Unlike CTAT which requires some light programming in Excel for its mass production step, our interaction design requires neither prior programming knowledge nor an understanding of our tool's underlying AI. Ultimately, we see this study as a step toward using Simulated Learners to make ITS authoring easy enough that teachers could author tutoring systems without any special training, and fast enough that instructional designers could build the core functionality of tutoring systems in as little time as it takes to write a worksheet and grade a few students' work.

Implications For Programming-by-Demonstration (PBD)

Generally, PBD systems support a *performance-model perspective*, where the goal is to train an agent to correctly perform tasks. The field of PBD may benefit from adopting interaction designs which support the *model-tracing-completeness perspective* as well. Consider a case where an agent taught via PBD must choose among several possible actions. For example, one might train an agent to treat possible instances of cancer. The agent could be responsible for diagnosing and prescribing treatment via the same iterative process that an oncologist might go through—ordering several rounds of diagnostic tests, checking the progress of treatments, and changing strategies when appropriate. A well trained agent would choose an action at each interaction in alignment with the positive feedback of its human teacher. Usually, this teaching would result in the agent learning a single course of action or single strategy. However, it may be the case that unforeseen circumstances make this strategy impossible or undesirable. For example, the appropriate course of action may be to prescribe a particular drug, but the drug has run out. In this case, the agent will need a plan-B. If the agent has only been trained from a *performance-model perspective* then its second choice of action may be erroneous or even catastrophic to the user. For example, the agent may instead prescribe a dangerous and unnecessary surgery. However, if the agent was trained using an interface which encouraged a *model-tracing-completeness perspective* then the oncologist training the agent could demonstrate alternate solution strategies for various situations and cull out any misconceptions the agent may temporarily acquire about those strategies. In this way, the agent would accurately learn alternative diagnosis and treatment strategies to more flexibly deal with unforeseen circumstances.

FUTURE WORK

Visual Features

The results of our user study revealed several considerations for future revisions to our current design. One unanticipated issue was that some users reported having trouble switching focus between the skill window and the tutoring interface. Although these participants understood that their objective was to ensure model-tracing completeness at each step, their focus was often directed only on the current staged skill. In cases when the staged skill was correct, participants often simply pressed the Yes button to navigate to the next step without considering any other proposed skill applications. One solution to this issue would be to have the authoring tools show all of the applicable skills directly on the tutoring system interface. For example, the staged skill could be highlighted in color, and the others in gray. This would allow users to quickly assess the model-tracing completeness of a particular step without requiring them to switch their focus to the skill window and manually toggle between the proposed skill applications. An alternate solution would be to remove the Yes and No buttons entirely to slow the user down and force them to use the skill window to consider all proposed skill applications.

A related aspect of the AL authoring interface that users reported having difficulty with was understanding the description of skills in the skill window. The content of the skill window is organized into a nested list with skills as major items and applications of those skills as minor item. Skills are written out as mathematical formulas with variables. Skill applications contain both the identifiers for the interface elements which bind to those variables (the where parts), in addition to the input value derived from evaluating the skill formula on those interface elements. Since interface element identifiers are sometimes named arbitrarily, for example 'div64', users did not find this format particularly useful for identifying skills. Users often opted to click through the skill window manually, staging items, to see the action of each skill application visually on the tutoring interface. To improve the readability of the skill window it may be helpful to make use of letter or symbol identifiers in the description of skill applications. These identifiers could have counterparts displayed on the tutoring interface to help users associate the content of the skill window with the content of the tutoring interface.

Recovering From Mistakes

A core usability issue which we anticipated among our users was the lack of means to recover from mistakes. In our current design, mistakes of providing erroneous positive or negative feedback can be remedied by outweighing these errors with several instances of correct feedback. However, some errors cannot be easily recovered from. For example, some forms of erroneous demonstration can cause the *where-learning* mechanism to considerably overgeneralize and propose a very large

number of mostly incorrect skill applications. In our current system, recovering from this sort of mistake would require tediously giving negative feedback to all of these skill applications across the whole problem space.

We have identified a few design strategies that might help users recover from errors. A simple undo button is a common means for recovering from mistakes, and we hope to include this in future work. However, we found that our users were usually unaware of their mistakes, so an undo button would only have solved a small part of the problem. We found that a few issues with our *where-learning* and *when-learning* mechanisms made it hard for users to recover from mistakes easily and made our system vulnerable to unrecoverable failure.

For example if a user incorrectly gave positive feedback to an incorrect action and then tried to correct this error later, our current *when-learning* mechanism would weight the correction equally to the erroneous feedback, meaning users would need to repeatedly reinforce the good behavior to overcome the error. As a general design recommendation we suggest that new user demonstrations should override old ones should a conflict arise. This way feedback mistakes, including ones that users have repeatedly reinforced can be remedied without having to repeatedly specify correct behavior.

Additionally, we found that the current *where-learning* mechanism in our AL agents would sometimes cause catastrophic overgeneralization errors, where an erroneous demonstration would lead to the when conditions binding to many more sets of interface elements than the user intended. As a general design recommendation, we think user demonstrations should cause conservative changes in case they need to be undone. An improved version of our *where-learning* mechanism would be cognizant of the number of new sets of interface elements it would bind to by generalizing its matching rules. This alternate mechanism might hold off on performing generalizations that result in large changes to the matching behavior until the generalization can be substantiated by subsequent examples. A *when-learning* mechanism like this could give the user the opportunity to correct erroneous demonstrations well after they were produced.

Prior Knowledge and Skill Induction

One aspect of authoring which we have overlooked for the purposes of this study is prior knowledge specification. The search-based planner that induces skill formulas requires a set of functions that it can chain together to explain demonstrations. In our study, we preloaded a few functions sufficient for multi-column addition. In a full authoring suite however, a very large number of functions could be available. Savvy users may even be inclined to author their own functions. Thus, we still must determine how users might select a subset of these functions for the purposes of authoring in a particular domain. This could be done with a simple menu with toggle buttons displayed at the beginning of authoring. However, it may be possible to skip this step altogether by having a Simulated Learner induce skills from all (or at least a large subset) of the available functions, and then have the user narrow down the set of explanations for a demonstration by searching for their intended formula. This process could be assisted by including a search bar where users could type keywords associated with their intended formula, or input these keywords multi-modally with speech-to-text.

Another approach would be to allow skill formulas to be refined through the training process instead of forcing users to zero in on a skill's formula at its first demonstration. Currently, when multiple explanations are available for a demonstration an AL agent will instantiate a skill with a formula taken from the most parsimonious of the conflicting explanations. If this formula proves to be incorrect in a later situation a new one may be induced. SimStudent has demonstrated a potential solution to this problem in its ability to revise skill formulas. For example, a demonstration of 2 and 2 makes 4 might induce a skill with formula Add(x,y); however, if later there is a demonstration in a similar situation of 3 and 3 makes 9, then the formula may be revised to Multiply(x,y) since this generalizes to both 3*3=9 and 2*2=4. An advantage of this approach is that the user never actually needs to inspect the formulas induced from demonstrations. A challenge to this approach is that determining when a skill formula should be changed and when a new skill should be induced is not always clear. In SimStudent users explicitly express the connection between each demonstration and the skill it should refine or induce. While this helps to address generalization problems, it would introduce an additional step to the authoring process.

Supporting Model-tracing Completeness

It was not always easy for our users to identify whether or not they had trained an AL agent to the point of model-tracing completeness. In CTAT, users can visually see the demonstrated structure of a problem's solution space through its behavior graph. By contrast, it is not always clear where a Simulated Learner needs additional demonstrations or correctness feedback without explicitly going through problems, perhaps several times.

One way of making a user's progress toward model-tracing completeness more transparent would be to display an induced behavior graph for each problem. This behavior graph could be constructed by proceeding from the start state to the done state in a breadth first fashion by adding edges for each action suggested by a Simulated Learner at each state. Immediate candidates for user feedback would be actions which result in states with no path to the done state. However, after these cases have been eliminated there will be extraneous paths which lead to the done state but are incorrect. However, the interface could support the user in identifying these paths.

In our study, users had difficulty keeping track of all the situations where they had provided feedback or not. For problems with exponentially larger solution spaces reaching model-tracing completeness with our current interaction design may prove elusive. However, we believe it would be possible to support users in these sorts of cases. For example, to aide users in finding paths which could use explicit feedback, a Simulated Learner could keep track of which paths it had implicitly constructed and which were specified explicitly by a user. In principle, an AL agent could even construct a continuous sense of its confidence in a particular skill application by gauging how similar the situation is to states and skill applications

which have been provided explicit feedback. This information could be visualized on the problem level by annotating each edge in an induced behavior graph with these confidences. Similarly, at the step level this information could be used to create a skill application conflict resolution strategy which places low confidence skill applications higher in the skill window, with the least confident, and thus most likely to be incorrect, skill application staged by default. Employing this strategy would make likely incorrect skill applications more salient to the user. A secondary confidence heuristic could even be aware of low skill application confidences downstream from the current state, leading the user to give feedback along paths which they have not already traversed. In principle, users do not need to give explicit feedback in every conceivable situation since AL agents generalize across situations. Thus a well crafted confidence heuristic could significantly cut down on the number of situations that a user needs to check to reach model-tracing completeness.

Making Smarter Agents

One of our users referred to the agent as 'dumb' because after many demonstrations it was still making mistakes. There are a few ways that an agent could more quickly converge to the behavior a user intended for it to learn. SimStudent included a feature in its *when-learning* mechanism that made positive feedback to one skill count as implicit negative feedback to all other skills. Implicit negatives were overridden by explicit positive feedback to avoid interference between skills. This implicit negative feedback caused skills to be applied much more conservatively. For the purposes of achieving model-tracing completeness, implementing implicit negatives in AL agents could reduce the amount of time users spend searching for erroneous skill applications. A drawback of this approach is that it would require the users to make a larger number of explicit demonstrations, which are slightly more time consuming and prone to error than providing correctness feedback.

Another concern with *when-learning* mechanisms in extant machine teaching systems is that they tend to commit to a single set of conditions that separate negative examples from positive ones. In reality, there is often a large space of condition sets which could separate all of the positive and negative examples produced so far by the user. Consequently, sometimes the set of conditions that a *when-learning* mechanism picks in this large space results in false negatives where the agent does not produce an intended skill application, or a false positives where the agent produces an incorrect skill application. The Gamut system's strategy for reducing the size of this space was to prompt the user to select interface elements on which the actions produced by the system depended [19]. Similarly, AL uses foci-of-attention to make skill induction easier, but a similar interaction could be implemented to aide the *when-learning* mechanism specifically. Alternatively, an AL agent's *when-learning* mechanism could use spatial and temporal heuristics to pick conditions sets which are more likely to be consistent with future demonstrations. These heuristics would encode the fact that steps in procedural tasks tend to be done roughly in some order and roughly spatially close together (e.g., problems may generally be solved from left-to-right and down).

CONCLUSION

In this paper, we presented a novel interaction design for creating intelligent tutoring systems by training Simulated Learners. We demonstrated that our novel interaction design supported users in creating nearly model-tracing complete ITSs in less than a quarter of the time it would take to author the same ITSs with CTAT example-tracing. Finally, we provided several design recommendations for future work in Simulated Learner based authoring tools.

ACKNOWLEDGEMENTS

The research reported here was supported in part by a training grant from the Institute of Education Sciences (R305B150008). Opinions expressed do not represent the views of the U.S. Department of Education. Big thanks to Google for providing us with a 2018 Google Faculty Research Award supporting this work.

REFERENCES

[1] Vincent Aleven, Bruce M McLaren, Jonathan Sewall, and Kenneth R Koedinger. 2006. The Cognitive Tutor Authoring Tools (CTAT): Preliminary Evaluation of Efficiency Gains. In *International Conference on Intelligent Tutoring Systems*. Springer, 61–70.

[2] Vincent Aleven, Bruce M McLaren, Jonathan Sewall, Martin Van Velsen, Octav Popescu, Sandra Demi, Michael Ringenberg, and Kenneth R Koedinger. 2016. Example-Tracing Tutors: Intelligent Tutor Development for Non-Programmers. *International Journal of Artificial Intelligence in Education* 26, 1 (2016), 224–269.

[3] John R Anderson. 1996. ACT: A Simple Theory of Complex Cognition. *American Psychologist* 51, 4 (1996), 355.

[4] Leo Breiman. 2017. *Classification and Regression Trees*. Routledge.

[5] Albert T Corbett, John R Anderson, and Alison T O'Brien. 1995. Student Modeling in the ACT Programming Tutor. *Cognitively Diagnostic Assessment* (1995), 19–41.

[6] Allen Cypher and Daniel Conrad Halbert. 1993. *Watch What I Do: Programming by Demonstration*. MIT press.

[7] Kenneth R Koedinger, John R Anderson, William H Hadley, and Mary A Mark. 1997. Intelligent Tutoring Goes To School in the Big City. (1997).

[8] Kenneth R Koedinger, Noboru Matsuda, Christopher J MacLellan, and Elizabeth A McLaughlin. 2015. Methods for Evaluating Simulated Learners: Examples from SimStudent.. In *AIED Workshops*.

[9] John E Laird, Kevin Gluck, John Anderson, Kenneth D Forbus, Odest Chadwicke Jenkins, Christian Lebiere, Dario Salvucci, Matthias Scheutz, Andrea Thomaz, Greg Trafton, Robert E Wray, Shiwali Mohan, and James R Kirk. 2017. Interactive Task Learning. *IEEE Intelligent Systems* 32, 4 (2017), 6–21. DOI: http://dx.doi.org/10.1109/MIS.2017.3121552

[10] Tessa Lau, Steven A Wolfman, Pedro Domingos, and Daniel S Weld. 2001. Learning Repetitive Text-editing Procedures with SMARTedit. In *Your Wish is My Command*. Elsevier, 209–XI.

[11] Toby Jia-Jun Li, Amos Azaria, and Brad A Myers. 2017. SUGILITE: Creating Multimodal Smartphone Automation by Demonstration. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. ACM, 6038–6049.

[12] Wenting Ma, Olusola O Adesope, John C Nesbit, and Qing Liu. 2014. Intelligent Tutoring Systems and Learning Outcomes: A Meta-Analysis. *Journal of Educational Psychology* 106, 4 (2014), 901.

[13] Christopher J MacLellan. 2017. *Computational Models of Human Learning: Applications for Tutor Development, Behavior Prediction, and Theory Testing*. Ph.D. Dissertation. Carnegie Mellon University.

[14] Christopher J MacLellan, Erik Harpstead, Robert P Marinier III, and Kenneth R Koedinger. 2018. A Framework for Natural Cognitive System Training Interactions. *Advances in Cognitive Systems* 6 (2018), 1–16.

[15] Christopher J Maclellan, Erik Harpstead, Rony Patel, and Kenneth R Koedinger. 2016. The Apprentice Learner Architecture: Closing the Loop between Learning Theory and Educational Data. *International Educational Data Mining Society* (2016).

[16] Christopher J MacLellan, Erik Harpstead, Eliane Stampfer Wiese, Mengfan Zou, Noboru Matsuda, Vincent Aleven, and Kenneth R Koedinger. 2015. Authoring Tutors with Complex Solutions: A Comparative Analysis of Example Tracing and SimStudent. In *The 2nd AIED Workshop on Simulated Learners*. CEUR-WS.org, Madrid, Spain.

[17] Christopher J MacLellan, Kenneth R Koedinger, and Noboru Matsuda. 2014. Authoring Tutors with SimStudent: An Evaluation of Efficiency and Model Quality. In *International Conference on Intelligent Tutoring Systems*. Springer, 551–560.

[18] Noboru Matsuda, William W Cohen, and Kenneth R Koedinger. 2015. Teaching the Teacher: Tutoring SimStudent Leads to More Effective Cognitive Tutor Authoring. *International Journal of Artificial Intelligence in Education* 25, 1 (2015), 1–34.

[19] Richard G McDaniel and Brad A Myers. 1997. Gamut: Demonstrating Whole Applications. In *Symposium on User Interface Software and Technology: Proceedings of the 10 th annual ACM symposium on User interface software and technology*, Vol. 14. 81–82.

[20] Tom M Mitchell. 1982. Generalization as Search. *Artificial Intelligence* 18, 2 (1982), 203–226.

[21] Stephen Muggleton. 1991. Inductive Logic Programming. *New Generation Computing* 8, 4 (1991), 295–318.

[22] Tom Murray. 2003. An Overview of Intelligent Tutoring System Authoring Tools: Updated Analysis of the State of the Art. In *Authoring Tools for Advanced Technology Learning Environments*. Springer, 491–544.

[23] Brad A Myers. 1986. Visual Programming, Programming by Example, and Program Visualization: a Taxonomy. In *ACM SIGCHI Bulletin*, Vol. 17. ACM, 59–66.

[24] John F Pane, Beth Ann Griffin, Daniel F McCaffrey, and Rita Karam. 2014. Effectiveness of Cognitive Tutor Algebra I at Scale. *Educational Evaluation and Policy Analysis* 36, 2 (2014), 127–144.

[25] Steven Ritter, John R Anderson, Kenneth R Koedinger, and Albert Corbett. 2007. Cognitive Tutor: Applied Research in Mathematics Education. *Psychonomic Bulletin & Review* 14, 2 (2007), 249–255.

[26] Herbert A Simon. 1983. Why Should Machines Learn? In *Machine Learning*. Elsevier, 25–37.

[27] Kurt VanLehn. 2006. The Behavior of Tutoring Systems. *International Journal of Artificial Intelligence in Education* 16, 3 (2006), 227–265.