Writing Reusable Code Feedback at Scale with Mixed-Initiative Program Synthesis

Anonymized for review

ABSTRACT
In large classes, teacher feedback on individual student coding mistakes is often infeasible. Program synthesis techniques can generate potential fixes for student code, as well as hints about them. However, these fully automatic approaches lack a teacher’s deep domain knowledge and can generate technically correct but subjectively bad fixes. We contribute a new mixed-initiative approach which combines a teacher’s expertise with program synthesis techniques that learn code transformations from examples. In the MISTAKE BROWSER system, transformations are learned from examples of students’ own bug fixes. In contrast, the FIXPROPAGATOR system interactively learns transformations from the teacher, as they fix bugs in student code. The teacher adds domain knowledge in the form of feedback and hints for each transformation. Feedback can be returned to the original students and can also be used for future students whose buggy solutions can be fixed with the same transformation. Two studies suggest that this approach helps teachers better understand student bugs and provide automatically reusable feedback to a larger number of students.

Author Keywords
programming education; program synthesis

INTRODUCTION
One of the most scalable, common forms of instantaneous debugging assistance in Computer Science (CS) classes is the autograder. Typical autograders display the results of running student code on teacher-written test suites, but this leaves students with a large gulf of evaluation in how failed tests or errors relate to underlying problems in their code. Recent advances in program synthesis and data-driven techniques have greatly expanded the kinds of automated feedback that are possible to generate. Multiple new methods can automatically fix bugs in student code. The fix can be suggested to the student as a bottom-out hint, or hinted more abstractly, e.g., just telling the student which line of their code should be changed, as described in [18].

These recent advances in fully automatic code feedback suffer from two key flaws. First, the hints that can be automatically generated from synthesized fixes lack a teacher’s deep domain knowledge. They do not address the student’s underlying misconception or point students to relevant principles or course materials. Second, automatically generated fixes can be technically correct but demonstrate poor coding style and be potentially misleading when used as the basis of a hint. For example, Fig. 2 shows a code fix generated with Refazer [16], an existing program transformation technique. The suggested fix obfuscates the location of the student’s bug by inserting a nearly identical correct base case immediately before the student’s incorrect base case. Hinting at this fix by its location (e.g., ‘add code before line 2’) would be misleading because the student’s bug is in the return statement on line 3. This is poor coding practice and any hints generated from this fix

ACM Classification Keywords
H.5.m. Information Interfaces and Presentation (e.g. HCI): Miscellaneous
would be misleading by obfuscating the actual location of the mistake. This is not a one-time occurrence: our first study shows that teachers rejected, on average, 19% (σ = 18%) of the synthesized fixes suggested by Refazer.

While recent work in UI design like OverCode [6] helps programming teachers understand and compose reusable feedback on clusters of hundreds or thousands of correct student solutions, the techniques do not yet work well for buggy solutions that do not pass all the test cases [19]. CodeOpticon [7] helps teachers give more synchronous, over-the-shoulder feedback to students debugging their solutions, but it does not cluster similar buggy solutions, help teachers identify bugs in solutions, or automatically suggest the same feedback to future students struggling with the same bug.

In contrast, we introduce a mixed-initiative approach that leverages program synthesis to help teachers write high-level, scalable, and reusable feedback and hints for clusters of buggy solutions. Our systems allow teachers to compose and deliver feedback to current and future solutions with the same bug or misconception. Based on examples of bug fixes, a data-driven program synthesis algorithm learns transformations that correct bugs in student solutions. These transformations are helpful to teachers in several ways: (1) Buggy solutions are clustered by which transformation makes them correct, so cluster members share a common bug. (2) The transformation defines a synthesized fix for each cluster member, which can be visualized as a diff that focuses teachers on the bug and a fix that resolves it. (3) The transformation can be used to apply the fix and any associated high-level teacher feedback to unseen, future buggy student solutions. Clustering buggy solutions by applicable transformations and creating mixed-initiative interfaces based on data-driven program synthesis are two key technical contributions of this paper. We present two tools, MistakeBrowser and FixPropagator, to demonstrate our approach. To implement these tools, we extend an existing program transformation technique [16] and add user interfaces that illustrate two different interaction mechanisms.

In our MistakeBrowser system, transformations are learned from examples of student-written bug fixes. In this workflow, shown on the left in Figure 1, the teacher reviews clusters based on these transformations. Every cluster originates from one code transformation, and, for each cluster, the teacher can see the synthesized fixes that result from applying that transformation to each buggy solution. Even if the synthesized fix is of poor quality, the buggy code in that cluster all may share a misconception because it is corrected by the same fix. Fixes may accurately reflect students’ debugging practices, and if some of those practices are poor, the teacher will not want to propagate them to other students. Therefore, the system allows the teacher to compose feedback for all the students placed by data-driven program synthesis into that cluster, past and future. This teacher feedback could include explanations, hints, or references to relevant course materials.

Rather than learn bug fixes from students, the FixPropagator system, shown on the right in Figure 1, learns transformations from teachers correcting buggy student solutions. The process is cyclic and iterative. First, the teacher demonstrates a stylistically good fix that corrects a single student submission. They annotate the fix with high-level feedback. Since this workflow does not depend on prior student data, the buggy student solutions could be exam submissions, and instead of high-level feedback, the teacher annotates the fix with a rubric item and a point deduction. For each correction the teacher completes, the system’s data-driven program synthesis algorithm learns and propagates pending fixes (and their associated feedback or rubric item) to more buggy student solutions in the pile or existing dataset. The teacher reviews these pending suggestions and accepts them or modifies them, which in turn kicks off another backend process to learn and propagate a fix (and its feedback) to more buggy student solutions. These learned transformations can be reused to apply feedback to students through autograders or apply rubric items to exam submissions in current and future semesters.

We evaluated both tools in user studies, where users were current and former teachers from the staff of a massive programming class. In MistakeBrowser, teachers appreciated the ability of seeing automatically generated fixes. The study also confirmed our suspicions that fixes learned from historical student data need a human in the loop to review, annotate, or remove transformations. Although all participants found fixes they would not recommend, they were still able to provide high-level feedback about the student’s mistake. The study also suggested that MistakeBrowser helped teachers understand what mistakes and solution strategies were common, equivalent to simultaneously looking over hundreds or thousands of shoulders as students each fix a bug. In FixPropagator, teachers appreciated the tool’s ability to automatically detect fixes to not yet graded attempts and some teachers found elegant ways to fix the student solution that they would have otherwise missed. The study also showed that most of the times FixPropagator suggested a fix to another solution the fix was accurate. These studies do not evaluate learning outcomes for students receiving such feedback. Instead, they demonstrate how the systems can help teachers understand common student mistakes and provide fixes and feedback for hundreds of students at a time. This paper makes the following contributions:

- A mixed-initiative approach which uses program synthesis to provide reusable teacher-written feedback at scale for introductory programming tasks.
- Two novel systems demonstrating this approach.
- Two user studies with 17 programming teachers from massive classrooms which evaluate the implemented systems.

RELATED WORK

To enable personalized, timely feedback in large classes, most research has fallen into one of two categories: automatic feed-

Figure 2. A correct but “poor” automatically synthesized fix.
back generation and tools for scaling human-generated feedback. In this section, we review the role of feedback in learning, techniques for clustering student solutions and bugs, and tools for automated and manual feedback generation at scale.

**Feedback and Learning** Feedback is critical to the learning experience [1]. Personalized, timely feedback significantly improves learning outcomes [11] and help students overcome learning barriers in programming [10]. Traditionally, teachers can be excellent sources of personalized feedback. In a small group setting, a teacher can look over the shoulder and provide a high-level hint, such as reminding a relevant principle or pointing to the approximate location of an error. However, as the class size grows, such personalized attention becomes infeasible. Even if multiple teaching assistants are provided, the instructors still need to monitor dozens of students and handle hundreds of student submissions at a time [7].

**Clustering Student Bugs** In large classroom, many students likely share common errors and misconceptions [1, 5]. However, identifying and clustering these common misconceptions is still a challenge. One learnersourcing workflow [5] allows students to augment or replace the help of teachers by writing their own personalized debugging hints for peers, indexed by failed test cases. Similarly, HelpMeOut [8] collects student bug fixes that resolve compiler errors and exception messages. When a student encounters a compiler error or exception, HelpMeOut can suggest edits that other students made to resolve the error or exception, and allow teachers to review and comment on these suggested edits. These systems benefit from both massive enrollment and the existence of common solutions and errors. However, the feedback is only indexed by test cases, and students may fail a test case due to different errors and misconceptions. Our tools also automatically identify and cluster semantically similar mistakes, but still allow instructors to manually correct or rewrite the clustering rules. Kaleeswaran et al. [9] propose a technique to cluster student solutions for dynamic programming (DP) problems by solution strategy. They use static analysis to detect how students manipulate arrays that store subproblems’ results in a DP solution. Although the clusters may contain different types of bugs, the solutions share a similar strategy making it easier to use formal methods to do equivalence checking between them and synthesize fixes. While their clustering technique is limited by the solution type (e.g., DP), our technique clusters solutions independently of their types and strategies, focusing just on the bug fix.

**Tools that Support Instructor Feedback at Scale** There are several user interfaces and systems designed to empower teachers to manage large numbers of student submissions. OverCod [6] and Foobaz [4] normalize and cluster students’ correct solutions so that teachers do not need to read thousands of student solutions in order to identify common and uncommon student choices about syntax and style. AutoStyle [12] clusters correct student solutions using a metric of code complexity so that teachers can write hints for each cluster about how the code in that cluster can be written more simply. Singh et al. [17] define a problem-independent grammar of features; a supervised learning algorithm trained on teacher-graded examples can map new student code submissions to grades. CodeOpticon [7] enables instructors to monitor many students simultaneously and provide situated help on code-in-progress.

**Algorithmically Generating Debugging Feedback** Intelligent Tutoring Systems (ITS) seek to emulate one-on-one tutoring and provide personalized feedback by using rule-based or constraint-based methods [20]. However, traditional rule-based feedback requires much time and expert knowledge to construct [15]; it does not scale well to the complexity of programming problems.

Data-driven methods have recently been introduced to augment existing techniques. Rivers et al. [15] use student data to incrementally improve ITS feedback, within the domain of Python programming assignments. Codewebs [13] and Codex [3] use statistical machine learning to analyze large volumes of code, extract patterns, flag anomalies as possible errors, and, if deployed in an educational context, could deliver feedback. These techniques can leverage the statistical properties of large numbers of student solutions, but they suffer from the cold-start problem. Even when large amounts of student data is available, they may not fail gracefully on novel inputs far outside what they have been trained on.

**Program Synthesis for Feedback Generation** Recent advances in program synthesis can help programming teachers and students in verifiably correct ways that statistical or rule-based techniques cannot. AutomataTutor [2] uses program synthesis to generate conceptual hints in the domain of automata constructions. Synthesized bug fixes have also been used to generate personalized hints for introductory-level programming assignments [9, 18]. AutoGrader [18] can find a minimal sequence of “repairs” that transforms a student’s incorrect solution into a correct one; however, it requires that instructor manually write down an error model of possible local modifications ahead of time. Instead of requiring a hard-coded error model, Rolim et al. [16] take an example-based approach to learn code fixes from previous students’ debugging activity.

Techniques like program synthesis can produce pedagogically bad feedback. If the generated code fixes are poor in quality (e.g., Fig. 2), the corresponding hints can potentially distract or confuse students. Propagating discovered student fixes to new students without oversight may eventually spread poor coding practices. To avoid such adverse effects, feedback needs to be carefully considered before revealing it to students. Thus, we leverage teachers’ expertise to ensure the quality of feedback as well as to provide high-level, conceptual hints.

Our work is motivated by the insight that systems backed by program synthesis can allow teachers to deliver their own corrections and feedback to students in a scale-able, reusable way.

**MIXED-INITIATIVE INTERFACES FOR WRITING REUSABLE CODE FEEDBACK**

We established the following design goals for our systems from the literature review and our understanding of current pain points in large programming courses: (1) Give teachers
a better understanding the distribution of common student bugs for introductory coding assignments. (2) Aid teachers in understanding the nature of the bugs and how to fix them. (3) Give teachers leverage to scale teacher-authored feedback to large numbers of students in a way that is reusable across semesters. We first briefly review how program synthesis enables these new interfaces, then describe both systems.

Using Program Synthesis To Cluster Submissions

To reduce teacher burden, our systems automatically find groups of student solutions that exhibit the same underlying problem. We extract code transformations from pairs of buggy and correct student solutions. We then check if a transformation can be successfully applied to other buggy student solutions. Success is defined relative to an assignment’s test suite: a transformation is successful if applying the transformation makes the corrected solution pass all tests.

In **MISTAKEBROWSER**, the pairs of buggy and correct solutions come from histories of students’ submissions to an autograder, culminating in a correct submission; in **FIXPROPAGER**, buggy solutions comes from students and are manually fixed by teachers (see Figure 1).

Naive extraction of code transformations, e.g., through simple text differencing or abstract syntax tree differencing, does not work well. Consider the two student solutions in Figure 3, center column: while they are conceptually similar, and indeed exhibit the same underlying problem, they differ both in variable names and in code structure — one uses a loop with an index variable, and the other uses list iteration. Thus, it is important to find *abstract* transformations that capture edits at a level that can be reused across different students. In our example, an abstract transformation might express that a student replaced the 0 on the right-hand-side of an assignment with function parameter base; and the function call inside the return statement should be replaced with the second argument inside that call.

We generate abstract code transformations using the Refazer system [16]. Refazer uses a Domain-Specific Language (DSL) to specify transformations, and synthesizes transformations as programs in that DSL that map from buggy solutions to correct solutions. The language allows abstracting nodes in the Abstract Syntax Tree (AST) of a solution using a tree pattern matching language. It then offers common tree edit operations to modify nodes in the AST, such as Insert, Delete, Update, and Move. Returning to our example, the transformation synthesized by Refazer to fix submissions 10 and 11 in Figure 3 has 3 AST operations: (i) Update a constant value to base; (ii) Delete a function call with two name arguments located in a return statement and (iii) Move the second argument of this call to the beginning of the return statement.

Browsing Student Bugs

Consider the teaching staff of a massive introductory programming class, CS1, that have been using the same programming assignments for weekly ‘finger exercises’ every semester for years. These exercises are intended to reinforce new concepts introduced in class each week. Since teaching staff are not present when students attempt these exercises, they do not know what bugs and misconceptions are most common in student code, except through students’ forum posts.

Before the semester starts, Jamie, the lead teaching assistant (TA) loads student code snapshots from the prior semester from the course’s autograder into **MISTAKEBROWSER**, shown in Figure 3. The back-end learns reusable transformations from the bug fixes made by students in previous semesters. The **MISTAKEBROWSER** front-end displays, one at a time, clusters of buggy solutions that are corrected by the same transformation, along with their synthesized fixes. The center pane lists all student submissions, showing incorrect code fragments in red, and fixes in green in a common code difference view (Figure 3D). The lead TA would review each cluster and write down conceptual feedback for each cluster. To help the teacher understand the cluster, the interface shows...
a compact representation of the fix for the cluster (Figure 3A), how many student solutions comprise the cluster (Figure 3B), and information about the first test case they all fail in the teacher’s test suite by returning the same unexpected value (Figure 3C). There are two clustering variants we consider in the user study that follows, and Figure 3 shows the CLUSTERBYFIXANDTESTCASE variant. In the CLUSTERBYFIX variant, dynamic information, i.e., the results of running the solution on the teacher’s test suite, is ignored and only the solution’s AST is taken into consideration. These two variants are both considered in the user study, in case it has a significant impact on the ease of the teacher’s task: identifying a common shared bug in each cluster and writing feedback about it.

After reviewing a cluster, the TA composes high-level feedback that applies to all submissions within the cluster in a free-form text (Figure 3E). In the example shown in Figure 3, the teacher may provide the hint “Assign the correct initial value to your accumulating total, and make sure you return that value on completion.”

When the TA is satisfied that the most common and interesting clusters have been annotated with explanations, hints, or references to relevant course materials, MISTAKEBROWSER would be left running as part of the course autograder’s back-end, where it could deliver the TA’s feedback to students during current and future semesters, along with the test case successes and failures, whenever a buggy student solution falls into an annotated cluster in MISTAKEBROWSER.

System design As we mentioned, MISTAKEBROWSER clusters are based on program transformations synthesized by Refazer. For each homework assignment, the back-end keeps a list of Refazer transformations and the assignment’s test suite. Given a buggy solution, the system iterates over the list of transformations, and for each transformation, tries to apply it and checks whether the code is fixed according to the test suite. As soon as the system finds a transformation that fixes the solution, it adds the solution to the cluster represented by this transformation. In the CLUSTERBYFIXANDTESTCASE, the system uses additional information provided by the tests related to the actual and the expected output to create clusters.

Propagating a Teacher’s Bug Fixes
Sam, the lead TA of a massive introductory programming class at another school wants to deploy the same kind of high-level feedback on buggy solutions that MISTAKEBROWSER enables. However, their course infrastructure only saves the most recent submission from each student, so there is no history of student bug fixes from which MISTAKEBROWSER could learn transformations. Instead, Sam uploads the buggy solutions he has for each student into FIXPROPAGATOR. Figure 4 shows the FIXPROPAGATOR user interface.

In the FIXPROPAGATOR interface, Sam looks at buggy solutions by selecting them (Figure 4A), iteratively edits and executes the buggy solution from an interactive code editor (Figure 4B) against the teacher’s test suite (Figure 4C), and adds some high-level feedback for the student, such as explanations, hints, or pointers to relevant course materials. Ideally, this feedback should be worded such that future students in need of a similar fix would also find beneficial (Figure 4E).

When Sam submits feedback, the original buggy solution, the fixed solution, and the high-level feedback are uploaded to the FIXPROPAGATOR back-end to synthesize possible generalized transformations based on Sam’s correction. FIXPROPAGATOR applies each transformation to the buggy solutions that do not yet have feedback. Transformations that fix buggy solutions turn into suggested fixes (along with its corresponding feedback) in the FIXPROPAGATOR interface (bottom of Figure 4A) that can be accepted with a single click (Figure 4D). If accepted, the tests are run automatically and Sam sees that, indeed, this fix is just what the student needs to correct their submission. Sam clicks on a button to reuse the feedback from the submission that generated the fix (Figure 4E). If Sam judges the fix or the feedback as not appropriate, it can be modified in place. Changes to synthesized fixes become new bug fix examples that spur the generation of new transformations in the back-end. Sam alternates between reviewing suggestions and manually correcting more buggy solutions. After a while, most submissions have suggestions.

System design Given the high cost of debugging students’ submissions, a teacher should be able to fix few submissions and see feedback propagate to many other students. We improved the Refazer back-end so that it can synthesize generalizable fixes from just one fix. To improve generalization, Refazer produces six transformation rules of varying generality for each submitted fix. All generated rules are applied to all submissions that have not yet been fixed by another rule. In our evaluation, we report on which hyperparameters appeared to produce the most effective, generalizable fixes.

Furthermore, FIXPROPAGATOR needs to support online fix generation at interactive speeds. However, effectively searching a space of program transformations can be time-consuming; with the current version of Refazer, it could take minutes to synthesize and apply fixes learned from an example fix to all other submissions. The user interface was decoupled from the Refazer back-end so that teachers can continue to fix and test code, produce feedback, and move on to other submissions while the back-end discovered fixes. All communication with the synthesis back-end aside from initialization was asynchronous: background threads were used to upload example fixes to Refazer and to query Refazer for recent fixes.

Our implementation anticipates future modifications to support collaborative production of fixes and feedback. Communications with the Refazer back-end are moderated as “sessions” sharing a common set of submissions and synthesis results. Small changes to our code could enable multiple users to connect to the same session to share fixes and feedback, and divide the task of reviewing many students solutions to produce generalizable feedback. We have made the code for both the web server and front end available under an open source license 1

USER STUDIES
We ran two in-lab user studies with teaching staff, one for each tool. The main goal of the studies was to evaluate how effective

1. URL redacted for anonymized submission.
our interfaces are at helping teachers understand common student bugs and bug fixes, and write reusable feedback.

Participants
We recruited 17 participants from the pool of current and former CS1 teaching staff members (‘teachers’). CS1 is the local massive introductory programming class at our institution. All the participants in our study are over 18 years old (average: 19.76 years old, $\sigma = 1.39$) and current or former teaching assistants of the same programming class at a large US university. 16 of the 17 participants are currently serving on the class teaching staff, and the remaining participant was previously a class teaching assistant for many semesters. We split the participant pool into two groups of size 9 and 8; the first group tested MistakeBrowser, while the second group tested FixPropagator. All participants were qualified to try both tools, but we limited each participant to one tool due to the time required by the evaluation. Additionally, the performance of a participant on FixPropagator might improve after having seen many examples of synthesized fixes when using MistakeBrowser.

Dataset
Whenever a CS1 student submits code to be tested by the course autograder against the teacher-written test suite, the system logs the code, student id, and test results. From one homework assigned in Spring 2015, we selected the three programming exercises below. For each exercise, we extracted the student’s final correct solution and the earliest buggy submission that Refazer could fix.

Product (data from 549 students): takes as parameters a positive integer $n$ and a unary function $term$, and returns the product of the first $n$ terms in a sequence: $\text{term}(1) \star \text{term}(2) \star \ldots \star \text{term}(n)$.

Accumulate (668 students): takes as parameters the same $n$ and $term$ as Product as well as a binary function $\text{combiner}$ for accumulating terms, and an initial value $\text{base}$. For example, $\text{accumulate}(\text{add}, 11, 3, \text{square})$ returns $11 + \text{square}(1) + \text{square}(2) + \text{square}(3)$.

Repeated (720 students): takes as parameters a unary function $f$ and a number $n$, and returns the $n$th application of $f$. For example, $\text{repeated}(\text{square}, 2)(5)$ returns $\text{square}(\text{square}(5))$, which evaluates to 625.

Refazer generated mostly small synthesized fixes for the dataset: on average, the tree edit distance between the Abstract Syntax Tree (AST) of a buggy student solution and the fixed solution was $4.9$ ($\sigma = 5.1$). However, Refazer did learn some larger fixes, with a maximum tree edit distance of 45.

Shared Protocol: Setup and Training
Participants were invited into the lab for one hour and offered twenty dollars in exchange for their time and expertise. After obtaining informed consent and collecting information about their age, year, and experience as a teaching staff member, the experimenter walked the participant through the features of the interface they would see, demonstrating actions on one of the students’ buggy solutions that the participant will be working on. This walk-through included a few minutes of explanation about the synthesis back-end. We chose to give this light explanation because, during pilot studies, subjects who did not receive an explanation were distracted from the task by their own curiosity about the back-end’s inner workings. The tutorial takes no more than five minutes. Finally, the experimenter shows the specifications of the first problem the participant will see buggy solutions for, i.e., a short description of its purpose, the test cases used to check its correctness, and brief explanations for each expected return value. As soon as the participants indicate that they have refreshed their memory of the assignment, the study-specific task begins.

**STUDY 1: MistakeBrowser**

The purpose of this user study is to evaluate MistakeBrowser and we ask the following research questions: (1) How do teachers perceive quality of synthesized fixes? (2) How do teachers perceive quality of explanations that help them understand common bugs?
Do synthesis-based clusters help teachers write feedback? (3) How reusable is the cluster-based feedback? To evaluate these research questions, we describe and quantify how many synthesized fixes teaching assistants find inappropriate; measure how conceptually coherent teachers perceive generated clusters to be; and observe how well teachers can annotate clusters with higher-level feedback.

Study Protocol
Participants had 40 minutes to view student mistakes and review and providing feedback for each cluster. They viewed two clustering variants, CLUSTERBYFIX and CLUSTERBYFIXANDTESTCASE, for 20 minutes each. The order of interface variants and choice of assigned problems from the three in our dataset were counterbalanced across subjects. After the second twenty period minute, subjects reflected on their experiences in a final survey. Participants were assigned a problem, cluster, and interface variant to start with. First, they marked all the bad synthesized fixes for student solutions. They then answered a few questions about the semantic coherence of the cluster, e.g., “Do these submissions share the same misconception?” Finally, they were asked to complete a survey to “write the most precise short description [they] can of the fix [they] would suggest,” which need not match the synthesized fix, and answer a few Likert scale questions about their confidence in their descriptions and the degree of the domain knowledge they added, in the process. As soon as they finished these tasks for the cluster, they could advance to the next of the largest three clusters in their assigned problem and interface.

Results
Refazer generated fixes for 87% of the students in our dataset, resulting in an average of 549 fixes across all three problems. On average, these fixes were clustered into 134 clusters in the CLUSTERBYFIX interface and 198 clusters in the CLUSTERBYFIXANDTESTCASE interface. Within the top three clusters for all problems the largest contains, on average, 109 solutions and the smallest contains 32 solutions.

Participants saw an average of 3 (σ = 1.4) clusters containing 145 (σ = 80.9) buggy solutions per hour-long session, where they spent 20 min in each cluster condition. They saw, on average, 72.9 (σ = 53.4) buggy solutions, in CLUSTERBYFIX condition, and 78.7 (σ = 43.9) buggy solutions in CLUSTERBYFIXANDTESTCASE condition. During that time, all participants reported bad fixes among the synthesized fixes. They were informed that most submissions in each cluster and interface variant to start with. First, they marked all the bad synthesized fixes for student solutions. They then answered a few questions about the semantic coherence of the cluster, e.g., “Do these submissions share the same misconception?” Finaly, they were asked to complete a survey to “write the most precise short description [they] can of the fix [they] would suggest,” which need not match the synthesized fix, and answer a few Likert scale questions about their confidence in their descriptions and the degree of the domain knowledge they added, in the process. As soon as they finished these tasks for the cluster, they could advance to the next of the largest three clusters in their assigned problem and interface.

Participants described how the synthesized fixes, shown as highlighted diffs, helped during the task: “highlight[ing] the part of the code that was incorrect ... made it much easier to quickly learn what was wrong with the code and how to fix it” (SS7). These diffs were “fast and easy to review” and “familiar” (SS3).

Subject SS1 wrote, “I thought it was interesting how grouping student answers by their common mistakes actually revealed something about the misconceptions they shared!” The utility of this clustering was apparent to SS3: “Seeing all of the similar instances of the same (or nearly the same) misconception was very useful, because it suggested ways to address common issues shared by many students.” They agreed with the statement “These interfaces gave me insight into student mistakes and misconceptions” at the level of 6.2 (σ = 0.44) on a scale from 1 (strongly disagree) to 7 (strongly agree), and no participant rated their agreement as lower than a 6.

Several participants’ responses support the hypothesis that MISTAKEBROWSER gives a high-level view of student mistakes and incomplete approaches to solving the problem, like OverCode [6] but for incorrect solutions. SS9 liked that “it had a wide variety of student responses to the same problem.” SS1 wrote, “I felt that being able to compare many different solutions (i.e. iterative, recursive, tail-recursive) was insightful as to how the students approached the problem.”

Reusability of the feedback To evaluate the reusability of the feedback wrote using synthesis-based clusters, we asked partic-
ipants to rate how many buggy solutions in each cluster shared the same misconception. Figure 5 shows participants’ answers using the two cluster conditions, CLUSTERBYFIX and CLUSTERBYFIXANDTESTCASE. In both conditions, they reported most submissions share the same misconceptions. However, they reported a greater proportion of CLUSTERBYFIXANDTESTCASE clusters as “100%” or “100% with a few exceptions”, compared to CLUSTERBYFIX clusters. Seven out of nine participants also mentioned in the final survey they preferred CLUSTERBYFIXANDTESTCASE cluster because the combination of fixes and test cases makes it easier to check if the buggy solutions share the same misconception. The CLUSTERBYFIXANDTESTCASE cluster where participants reported only 50% was due to a missing test case that did not reveal another bug in the solutions.

STUDY 2: FIXPROPAGATOR

The purpose of this user study is to evaluate FIXPROPAGATOR. Specifically, we have the following research questions: (1) Can FIXPROPAGATOR propagate fix suggestions based only on a small number of examples provided by a single teacher? (2) Can FIXPROPAGATOR’s back-end perform fast enough to support an interactive interface? (3) Can teachers reuse their written feedback associated with the propagated fixes?

Study-specific protocol

Each participant is assigned to interact with solutions to a single programming exercise out of the three. After their five-minute tutorial on the FIXPROPAGATOR system, they are given thirty minutes to interact with the system to teach the system the best code fixes. This thirty-minute period is broken up into alternating five-minute tasks. During the first five minutes, the participant fixes as many bugs as possible, to optimize the number of suggested fixes the system generates. The experimenter tells the participant that the simpler the bug fix, the more suggestions they may get per fix. During the second five minutes, the participant reviews pending fixes and then accepts or modifies them, so that they can check whether the system has learned acceptable transformations for fixing future, unseen buggy solutions. After completing the thirty-minute period, the participants were asked to fill out a post-study reflection survey and to rate the system and their experience with the 7-point Likert scale. We also asked them about qualitative feedback as free response questions.

Results

During our studies, due to unforeseen circumstances, Refazer was not enabled for the entirety of two participants’ study sessions (ST1, ST4), data described here was collected from the remaining six participants. Next, we describe the results.

Bug fix propagation

Participants provided examples of bug fixes in two ways: (1) fixing buggy solutions from scratch and (2) editing suggested fixes. They fixed a median of 10 (σ = 2.7) solutions from scratch and fixed 3 (σ = 2.9) more after editing suggested fixes. During the time of both the study session and a period of up to 40 minutes after the study, Refazer was able to fix a median of 201 submissions (σ = 47.7). Figure 6 shows the propagation of the fixes over time. By the end of the study, a large portion of the submissions (average = 34.7%, σ = 10.19%) had either been corrected by the participant or fixed by a synthesized transformation.

Performance

It took a median of 2 minutes and 20 seconds (σ = 7m34s) to successfully find a bug-fix suggestion after the teacher provides an example of a bug fix. This includes the time needed by Refazer to learn transformations based on the examples and try them on other buggy solutions. Although the current performance of Refazer does not allow teachers to immediately see the suggestions generated by their examples, teachers were able to work on other solutions while waiting for synthesized fixes. Figure 7 shows the interaction of one of the participants with FIXPROPAGATOR. The participant alternates back and forth between fixing and reviewing, but the effort invested in manual fixes results in accepting a large number of auto-propagated fixes.

The value of synthesized fixes

Transformations learned from the examples manually authored by the participants can fix new buggy solutions in unexpected ways. They help participants better understand the space of bug fixes and approaches to implementing the solution. For example, ST3 came across a buggy solution which was very close to being correct. She did not see the simple fix and instead wrote an elaborate fix that was fundamentally different from the student’s approach. Later, a simpler synthesized fix to a similar buggy solution was suggested, and she realized that these buggy solutions were using an alternative but valid approach to solving the problem. After accepting the suggested fix, she admitted she had learned something herself about the space of solutions to the homework assignment.

Reusability

Participants generally reused fixes and feedback they made on past submissions when reviewing fixes generated for other submissions. However, the participants were more likely to reuse the proposed fixes to the code verbatim (median = 17 times, σ = 8.9) than reuse feedback verbatim (median = 11, σ = 6.3). Intuitively, this makes sense: generated code fixes were exact, and only returned if they allowed code to pass test cases that it could not before the change was made. However, participants’ feedback could be specific (i.e., “Your starting value of z should be a function, not an int”), mentioning variable names and values that might occur only in one student’s code. When the proposed fixes and feedback were not enough, participants made modifications after applying the suggested fixes (median = 3, σ = 2.9), and altered the feedback for student submissions (median = 6, σ = 2.7).

In the survey, most participants rated pending fixes as acceptable 100% of the time, with a few exceptions. The rate of pending feedback acceptability was one category worse: participants rated the suggested feedback (linked with the more acceptable synthesized fix that corrected the buggy solution) as accurate only 75% of the time. This may be due to the fact that participants wrote feedback one solution at a time, and had to iteratively revise their feedback to make it specific to the bug but general with respect to student implementation choices, e.g., variable names. This suggests that new interaction techniques are necessary to allow teachers to write feedback that is abstract enough to help many students with the same bug and specific enough to be helpful to each one of those students,
 accused (ST7) product (ST5) repeated (ST8) generated feedback reused generated feedback modified generated feedback ignored feedback written from scratch feedback transferred to a new submission (each shade of gray is caused by a different fix) Time (min) Time (min) Time (min) 15 30 15 30 15 30

TA submits a correction for a buggy solution
TA accepts a synthesized fix
TA modifies and re-submits a synthesized fix
A transformation for a correction fixes a buggy solution

Figure 6. Top row: The number of buggy solutions for which participants provided feedback, shown for three participants (ST5, ST7, ST8). Bottom row: The number of buggy solutions to which a participant’s feedback was matched using the transformations learned from their corrections.

Figure 7. Timeline of the corrections a participant (ST8) made to buggy solutions, and the subsequent synthesized fixes that were generated by each correction.

despite varying variable names and code structure. The results highlight the promise of propagating feedback at scale to many students after a teacher demonstrates several fixes.

DISCUSSION
Our first design goal is to give teachers a better understanding of the distribution of common student bugs for introductory coding assignments. MISTAKEBROWSER achieved this goal by clustering buggy solutions by the transformation that corrects the common underlying bug. Given that the teachers in our studies had no comparable view of buggy solutions, this was a significant added value. FIXPROPAGATOR achieved this goal indirectly, by helping teachers discover how many different buggy solutions could be fixed with the same transformation.

Our second design goal is to aid teachers in understanding of the nature of the bugs and how to fix them. Both MISTAKEBROWSER and FIXPROPAGATOR achieve this goal by visualizing the synthesized fix as a diff for every buggy solution in each cluster. By seeing the variety of solutions fixed by a common transformation, teachers begin to understand the essence of the underlying misconception, as well as the variety of buggy solutions it can appear in.

Our third design goal is to give teachers a tool for composing high-level feedback and debugging assistance that scales to large numbers of students and can be automatically reused in future semesters. Teachers can achieve this goal with either system, depending on the availability and quality of archives of student debugging activity. As seen in the first user study, the existence of bad synthesized fixes does not prevent teachers from composing high-level feedback per cluster that can be propagated to current and future buggy solutions. Clustering by transformation and test cases reduced cluster size but increased cluster purity, as in, there was more likely to be a single bug shared across all buggy solutions in the clusters. Despite the smaller size of clusters in the CLUSTERBYFIXANDTESTCASE variant of MISTAKEBROWSER, teachers still reviewed as many or more buggy solutions. In the FIXPROPAGATOR system, after only a few minutes of manually fixing and providing feedback on a few buggy solutions, teachers can receive suggested bug fixes and feedback for tens of hundreds of additional buggy solutions. Even if only a large minority of students receive high-level feedback in which teachers can remind the student of relevant principles and course content, it is still a major advance over the existing status quo in feedback for students in massive programming classes.

One participant (SS3) mentioned that they used to hand-grade homework submissions, giving feedback as well as grades, until their class became too large. Now they only evaluate student homework based on a proxy for student effort, test cases passed, and spot-checks for composition. He thought that MISTAKEBROWSER could help the staff grade their massive class the same way they used to grade homework when the class was smaller. The FIXPROPAGATOR system can also be used for grading-through-debugging. Debugging a student exam solution is not a trivial activity, but FIXPROPAGATOR can potentially learn reusable transformations from every successful correction, simplifying adjustments to the grading rubric and point deduction midway through the process of grading and potentially automatically grading submissions to coding problems that are reused on exams in future semesters.

Limitations Our studies do not evaluate the impact of these systems on learning outcomes for students. The study results showed that instructors believed generated fixes and feedback applied to other buggy solutions. However, we have not shown whether students believe the fixes and feedback are relevant or whether they improve learning outcomes. There may be a trade-off between the generality and the relevance of the feedback instructors provide. Future studies can shed light into how propagated feedback and fixes impact student learning and inform how the systems might best encourage feedback that is both general and pedagogically useful.
So far, our systems have only been shown to propagate feedback for small fixes. For MistakeBrowser, this is due to the constraints of the training data: to build the training examples, we used only pairs of correct student solutions and the last incorrect student submission before the correct submission. It may be possible to synthesize larger fixes by training with earlier incorrect solutions. While larger fixes may allow instructors to give feedback on more problems, there is a tension: larger fixes may be harder for teachers to understand and provide feedback for.

Our datasets have thus far only focused on fixing short programs typical of early assignments in an introductory CS class. The student solutions usually consist of just one main function, sometimes including a few helper functions. We have not tested how our the systems’ online performance will scale with more complex programs. Intuitively, the time to apply a fix will depend on transformation size, the size of a buggy student solution, and the runtime of an assignment’s test harness. We note that our synthesis engine is capable of learning and applying transformations for complex code bases (150K-1500K lines of code) [16]. However, we expect a synthesis backend may need to be heuristically tuned to apply fixes quickly to many complex programs.

CONCLUSIONS AND FUTURE WORK

We presented two mixed-initiative systems for providing reusable feedback at scale with program synthesis. MistakeBrowser learns transformations to fix buggy solutions from examples of student-written bug fixes, and uses them to cluster buggy solutions. Teachers can then review these clusters and write reusable feedback to be used for future student submissions. When examples of student fixes are not available, FixPropagator allows teachers to write example bug fixes themselves. The system then learns from such fixes in real-time. We conducted two user studies with teaching assistants to evaluate our systems. Our results suggest that synthesized fixes, either from teachers‘ examples or previous students‘ bug fixes, can be useful for providing reusable feedback at scale.

As future work, we plan to deploy these systems in a massive programming course and evaluate the quality of the generated feedback from the students‘ perspective as well as student learning outcomes as a result of receiving this feedback. For example, we can quantify the number and severity of unhelpful hints generated by teachers using MistakeBrowser, to confirm that is low, in practice. To increase flexibility, we plan to combine workflows of FixPropagator and MistakeBrowser, allowing teachers to edit transformations learned from students‘ fixes by providing new examples. To improve the interpretability of learned transformations, we plan to combine visual and natural language to better describe the output of Refazer to teachers, and, additionally, allow teachers to modify the transformations. Finally, there are open user interface design questions about how a combination of teacher-authored feedback and synthesis-generated fixes should best be presented to a student.

REFERENCES


