Pedal: An Infrastructure for Automated Feedback Systems

Luke Gusukuma
Department of Computer Science
Virginia Tech
Blacksburg, Virginia
lukesg08@vt.edu

Austin Cory Bart
Department of Computer Science
University of Delaware
Newark, Delaware
acbart@udel.edu

Dennis Kafura
Department of Computer Science
Virginia Tech
Blacksburg, Virginia
kafura@cs.vt.edu

ABSTRACT

This paper describes Pedal, an innovative approach to the automated creation of feedback given to students in programming classes. Pedal is so named because it supports the PEDAgogical goals of instructors and is an expandable Library of components motivated by these goals. Pedal currently comes with components for type inferencing, flow analysis, pattern matching, and unit testing to provide an instructor with a rich set of resources to use in authoring and prioritizing feedback. The larger vision is the loosely-coupled architecture whose components can be readily expanded or replaced. The Pedal library components are motivated by a study of contemporary automated feedback systems and our own experience. Pedal's components are described and examples are given of Pedal-based feedback from three different introductory classes at two different universities. The integration of Pedal into several programming and autograding environments is briefly described.

CCS CONCEPTS

• **Social and professional topics** → **Computing education**;

KEYWORDS

CS Education; Immediate Feedback; Student Model; Feedback Architecture

ACM Reference Format:

Luke Gusukuma, Austin Cory Bart, and Dennis Kafura. 2020. Pedal: An Infrastructure for Automated Feedback Systems. In *The 51st ACM Technical Symposium on Computer Science Education (SIGCSE '20)*, March 11–14, 2020, Portland, OR, USA. ACM, New York, NY, USA, 7 pages. <https://doi.org/10.1145/3328778.3366913>

1 INTRODUCTION

Automated feedback on programming is important to students and instructors in introductory programming classes [20, 23, 27, 29]. Automated feedback facilitates student learning by providing timely indications of progress or problems; it also enhances instructors' effectiveness, especially when the system requires moderate instructor effort to support a large number of students.

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.

SIGCSE '20, March 11–14, 2020, Portland, OR, USA

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-6793-6/20/03...\$15.00

<https://doi.org/10.1145/3328778.3366913>

A variety of automated feedback tools have been studied [17], and recognized as critical open questions for practioners [7]. These studies focus on the impact of the feedback on student learning. We ourselves have explored an innovative form of feedback and evaluate its impact on the learning of a specific topic by novice learners [11].

What has received less attention is the systematic study and intentional design of the *infrastructure* of automated feedback systems. Such a study creates a better understanding of the infrastructure required by different forms of feedback and enables open source community-supported feedback systems. The example presented in this paper is an initial, imperfect step toward this vision.

The architecture and vision described in this paper is named Pedal (<https://github.com/pedal-edu/pedal/>), a Pedagogical Library developed as an automated feedback system for Python programs in introductory courses. Pedal is designed with the following goals:

- (1) enabling instructors to author a wider variety of feedback types.
- (2) delivering feedback in a deterministic manner using instructor specified priorities, and
- (3) supporting the interaction of library components within a flexible and extensible .

The authors have used Pedal to integrate automated feedback systems for three different courses at two different universities and in feedback research [11]. This paper makes four contributions:

- (1) an abstract model of and requirements for the infrastructure of feedback systems,
- (2) an example , that of Pedal, satisfying the requirements,
- (3) examples of Pedal-generated feedback in multiple classes, and
- (4) examples of integrating Pedal with different editors.

We also use our experience to highlight related practices of testing feedback before deployment and relating delivered feedback to students' instruction. Finally, we advocate for an open, extensible community infrastructure for automated feedback systems and the related tools that support development and assessment of feedback.

2 AUTOMATED FEEDBACK

In this paper, an automated feedback system refers to any system that takes a syntactically correct program and generates information intended to be delivered to the student programmer. Excluded by this definition are: (1) systems that have a "human in the loop" and are not fully automated, (2) feedback for syntactically incorrect programs in the form of compiler error messages [4], (3) feedback given without an explicit submission as in intelligent tutoring systems [1], (3) off-line components that analyze prior student submissions, and (4) user interface components that are more properly

part of the editing environment. While feedback from human instructors remains the gold standard, only fully automated systems can scale to large class sizes. Excluding compiler error messages focuses attention on feedback about semantic mistakes. Because intelligent tutoring systems tend to be problem-specific, excluding them emphasizes that the systems of interest should be usable for a wide range of problems. Off-line and user-interface components are, of course, valuable and necessary. We exclude them only to focus attention on the core elements of a system that produces the feedback for a given program. Finally, there are industry tools that provide feedback to professional programmers in various domains such as Fortify [9] for computer security. While Pedal has much in common with these tools, there are differences in the target audience, goals (products vs. pedagogy), and the need to adapt to diverse learning environments.

This paper attempts to provide an initial answer to the question "What infrastructure enables a robust, open-ended and flexible automated feedback system?" This focus on the feedback system's infrastructure is orthogonal to other papers on feedback systems that address different kinds of feedback, the impact of the feedback on student learning, of the specific analysis needed to produce feedback. While these studies are critical, they do not focus on the infrastructure itself.

In answering the above question, we are guided by (1) an abstract model of feedback systems, and (2) an analysis of exemplar feedback systems. The abstract model defines the essential nature of a feedback system. The model's framework and terminology acts as a lens through which we can view previous feedback systems. The abstract model is described in Section 2.1 While previous feedback systems did not focus on the infrastructure issues, they provide requirements for the infrastructure. For example, an output checking feedback system imposes the requirements that the student code must be executed, provided with specified inputs, and the outputs must be captured for analysis. To understand these requirements it is necessary to examine only a single exemplar output checking system that is representative of the entire class of such systems. The exemplar systems are reviewed in Section 2.2; the goal is not to present a comprehensive review of all feedback systems (see [15]) but to characterize the space of feedback systems and begin to understand various architectural requirements and the feedback they enable.

2.1 Modeling a Feedback System

Having an abstract model of a feedback system is useful for framing and understanding the requirements of the feedback system's. From a review of various feedback systems, presented next, our own experience, and interviews with practitioners, we have defined the model shown in Figure 1.

In this model, an interface is used to enable interactions of condition-response pairs with an underlying infrastructure. The infrastructure consists of components for executing and analyzing the student code. Each condition is stated in terms of observable characteristics of the input student code. The conditions may be simple, such as: "the student code produces a given output when supplied a given input". The conditions may also be arbitrarily complex, such as: "the student code defines a function with a given

Figure 1: Feedback System Abstract Model

*API is optional

signature and uses this function to control an iteration whose body involves appending an item to a list". When a condition holds, the corresponding response becomes (at least part of) a possible output feedback message. A response may be parameterized so that it can, for example, refer to the specific variable names used in the student code. A response may also contain information such as priorities to guide the infrastructure's selection of the single output feedback. We assume that there is a single feedback because this conforms to best practice [4].

The infrastructure must be capable of making the observations stated in the conditions. The examples above implied that the infrastructure could execute the input code, examine its name space and make type inferences (i.e., there is a function with a given signature), and look for structural patterns in the code (an iteration whose body appends to a list). This connection between the conditions and the infrastructure provide a way to organize and inform a study of exemplar systems.

Category	Exemplars	Condition Expression
Common Mistakes	PyTA	Program Analysis
Hint Generation	ITAP	Program Synthesis
Instructor Authored	MistakeBrowser/ Fix Propagator	*Various
Unit Testing	WebCAT	SandBxing

Table 1: Overview of Automated Feedback Systems

*Instructor authored feedback tends to use many methodologies

2.2 Exemplar Systems

Based on the abstract model, we examine a number of exemplar systems to determine what conditions on observable behavior they require. The exemplars are chosen from a variety of different types of feedback systems and organized into these categories: Common Mistakes, Hint Generation, Instructor-authored, and Unit Test. The below discussion is summarized in Table 1.

Common Mistakes systems deliver feedback about frequently occurring semantic mistakes. Exemplars of this include Espresso[15], FrenchPress[5], and PyTA[21]. These systems have little to no API interaction for the instructor beyond possibly suppressing certain mistake responses. The conditions are predefined as a set of common mistakes and have automatically generated responses. The

conditions are found/expressed using program analysis. These systems typically don't allow for overriding responses, but avoids the effort of giving customized responses over a large set of possible conditions.

Hint Generation systems give suggestions for code edits that bring an incorrect program closer to a reference solution. Exemplars of this include ITAP [25] (Intelligent Teaching Assistant for Programming) and SourceCheck [24]. This type of feedback system also typically has little to no API interaction for the instructor beyond allowing either locations or full edits to be revealed. The conditions in a hint generation are finding the next possible edit and are defined using program synthesis; the response is either a location or an edit. This approach avoids the effort of giving customized responses over a large set of possible conditions.

Instructor-authored feedback systems enable feedback tailored to mistakes that indicate a significant breakdown in learning. Exemplars of this include Mistake Browser/Fix Propagator [14], CSF² [13], [28], and our own work [11]. Instructor-authored systems typically have a more in-depth instructor API interaction. The conditions are either defined by the instructor, or a set of conditions are defined by the system, and the instructor uses an API to author appropriate responses. The conditions use a variety of tools which may include unit testing, program synthesis, program analysis, etc. These types of systems allow instructors to give a variety of custom responses, but they do not cover as many conditions as common mistakes and hint generation methods do and can be time consuming to increase the number of conditions.

Exemplars of Unit Testing systems are WebCAT [26] and GradeScope. In Unit testing systems, the API is typically a set of assertions and output checks. These assertions and output checks are used to compose conditions. These APIs also allow for the instructor to write feedback for the failing of a particular assertion. These types of systems strike a middle ground between instructor authored feedback systems and common mistakes systems as they allow instructors to use assertions to cover a wider variety of conditions, but they cannot get to the specificity of Instructor authored feedback nor the reusability of common mistakes.

2.3 Infrastructure

From the review of exemplar systems we can define three infrastructure requirements.

2.3.1 Support of many services and condition specifications. There are a variety of services that each type of feedback approach requires even though they may use some of the same condition-finding mechanisms. For example, ITAP (hint generation) uses program synthesis, but requires little in terms of available API, but MistakeBrowser/FixPropagator uses a more verbose way of specifying responses.

2.3.2 Communication between tools. Communication between tools is important so that analyses don't have to be re-run multiple times. Instructor authored feedback in particular can leverage analyses from various different condition-finding methodologies.

2.3.3 Ability to Override. Many hint generation and common mistakes feedback systems have builtin responses. However, when

conditions have multiple contending potential responses, it is necessary to prioritize/override responses when delivering feedback.

3 PEDAL

The Pedal (Figure 2) draws inspiration from the Blackboard design pattern [6] that consists of three components: a set of modules (feedback systems, tools, and inspectors), a blackboard, and a control. In Pedal, each module provides feedback-related functionality motivated by our analysis of automated feedback systems: syntax checking (Source), program analysis (TIFA), code inspection (CAIT), mistake patterns (Mistakes), safe execution (Sandbox), and unit testing (Assertions). The blackboard (Report) is a shared read-write entity through which the modules communicate and store feedback messages. A related component (Resolver) selects from feedback messages written to the blackboard. The control is an instructor-written element that orchestrates the execution of the modules, respecting that one module may expect the blackboard to contain artifacts produced by another module. An example of a control implementation is shown in Figure 4. Each component is described in more detail below.

3.1 Source

The Source module loads student code and writes an Abstract Syntax Tree (AST) to the blackboard (Report module). This module can also recognize files divided into sections by commented headers. Sections can be analyzed in successive phases (for progressive lab-style assignments) or in isolation (for exam-style questions). An example of a single section is given in Figure 3. This example is a sample solution to the following problem in a CS1 course.

Define a function named `summate` that consumes a plus-separated string of numbers from the user, and produce a float representing the sum of all the numbers. Call the function on a list and print the result.

This example will be used in the description of other modules.

3.2 TIFA

The *Type-inferencing Flow Analyzer*, or TIFA, statically analyzes the Abstract Syntax Tree (AST) looking for common issues related to variable liveness, scope, and types. A majority of these issues relate to the tracing of identifiers: variables being read before defined, being defined or overwritten without being read, or taking on a new type. Because the tracing respects branches, TIFA also takes issue with initialization that might only occur in only one branch even if a variable is read in every branch. Scope rules for function definitions are respected, so an issue is reported when a global variable is modified or read out of scope. If line 3 (`total = 0`) was moved to the beginning of the code in Figure 3, TIFA would detect that the variable `total` was modified out of scope. TIFA recognizes a number of other errors, such as students attempting to iterate over a non-sequence type (e.g., an `int`) or statements after a `return`.

TIFA's type system can account for primitive types (`Num`, `Bool`, `None`, `Str`), composite types (`Tuple`, `List`, `Dict`, `Set`, `Generator`), and has a simple model for mutable namespaces (e.g., classes). As a program is analyzed, the system tracks the current type of each variable and detects type mismatches in function calls, operators, and other kinds of expressions; this includes static type annotations supported

Figure 2: Pedal

```

1 def summate(plus_str):
2     numbers = plus_str.split("+")
3     total = 0.0
4     for number in numbers:
5         total += float(number)
6     return total
7 print(summate("1+2+3"))

```

Figure 3: Sample Student Code

in newer versions of Python. In Figure 3, if `summate(45+10)` was run, TIFA would recognize the fact that the variable `plus_str` was not a string, and could not therefore be `split`. TIFA follows the control flow of the program according to Python evaluation rules for sequencing, precedence, branching, and scoping.

TIFA's assumptions are similar to those made by other static Python analyzers [2, 8, 10, 22], not attempting to handle dynamic code execution like `eval`. TIFA writes issues identified and type information about variables to the blackboard. The system comes with a large number of default responses (which can be suppressed).

3.3 CAIT

The CAIT (Capturer for AST Included Trees) module allows inspection of the AST. CAIT's interface can be used for arbitrary traversal of the AST. More importantly, CAIT can also detect a specified code pattern in the student's AST. CAIT detects subtrees in the student's program that include the AST of the given pattern, using a variation on the Ordered Tree Inclusion Problem adapted to ASTs [18] (described further in [12]). CAIT provides ways for instructors to capture variables, variable usage, and expressions in student code, using the following identifiers:

- “`___`”: Triple underscores are a wild match card. It will match to any node or subtree.

- “`_var_`”: A variable (or function) name enclosed in single underscores is a place holder for variables instructors wish to reference. Otherwise variables are made with exact name matches.
- “`__expr__`”: A name enclosed in double underscores is a place holder for a subtree of an “expression”.

3.4 Mistakes

The Mistakes module is a collection of functions typically using both TIFA and CAIT; each function detects a mistake pattern. This module allows easy reuse of the detection of mistake patterns that occur across multiple assignments. Part 4 of Figure 4 illustrates an example of detecting a mistake pattern that occurred in several assignments in one of our classes. This example detects the mistake of placing a list in the place of the iteration variable. CAIT is used to pattern match the structural elements and TIFA is used to determine the data type of `_item_`. For simplicity, the detection code is shown directly in the instructor code. In practice detection code is implemented as a function in the Mistakes module and imported by the instructor code.

3.5 Sandbox

The Sandbox module provides functions to safely evaluate student code independently from the instructor code's namespace. Certain functions are overridden to avoid delays or to capture execution data. For example, the `Matplotlib` library is overridden so that any plotted data is stored in an accessible data structure. The input built-in function can be overridden to return an instructor-specified list of strings. The `open` function is restricted to opening non-source code files so that students' code cannot access the instructor grading scripts. Any printed output is captured to a list.

Execution is, by default, performed in a separate thread. If the students' code takes longer than some defined threshold, execution

```

1  from pedal import *
2
3  ## (1) Source
4  # Load the student code
5  set_source(open("student.py").read())
6
7  ## (2) TIFA
8  # Check for common TIFA issues
9  tifa_analysis()
10 # But ignore type changes, e.g., line 5
11 suppress("Type change")
12
13 ## (3) CAIT
14 # Students sometimes try to use input
15 # instead of passing in a parameter
16 if find_matches("input(____)":
17     explain("Do not use the input function.")
18
19 ## (4) Mistakes (CAIT + TIFA)
20 # Students use the list as the iteration target
21 # instead of the iteration value
22 matches = find_matches("for _item_ in ____: "
23     "    pass")
24 for match in matches:
25     if match['_item_'].was_type("list"):
26         explain("Iteration item was a list.")
27
28 ## (5) Sandbox
29 # Execute the student's code
30 student = run()
31 # Check that they called the function
32 if student.calls['summate'] < 1:
33     gently("You have not called summate.")
34 # Check that they printed something
35 if not student.output:
36     gently("You are not printing")
37
38 ## (6) Assertions
39 # Some basic unit tests
40 assertSignature(student, 'summate', str)
41 assertEquals(student.call('summate', "0"), 0)
42 assertEquals(student.call('summate', "1+1"), 2)
43 assertEquals(student.call('summate', "10+5"), 15)
44
45 ## (7) Resolver
46 # Select what feedback to show
47 resolve()

```

Figure 4: Sample Instructor Code

can be interrupted. Exceptions are also safely captured, to be augmented by the instructor. If an error occurs in student code, the traceback is modified to remove calls made by the infrastructure to execute the student's code. Function calls (including function names, arguments, and returned values) are traced so that instructors can track execution (e.g., to determine if a function was called recursively).

3.6 Assertions

Pedal's Assertion module provides an API that mimics the interface of Python's built-in `unittest` module, allowing the instructor to

write assertions about the students' code. This API has extensions such as checks for verifying the correctness of function and class definitions, the existence and type of variables and attributes, and that certain text has been printed as a result of a specific function call. This module can optionally use the data collected from the Sandbox module to improve its feedback. The Assertions module also provides a collection of decorators and context managers to help organize, sequence, and execute instructor tests.

3.7 Resolver

The Resolver determines which feedback message should be shown to the user. Different implementations can be used. The default implementation of the Resolver shows a single piece of feedback. Feedback can be given a specific priority level, and then within a priority level, a single piece of feedback is selected based on what order it entered the queue. The built-in priority levels follow from the default ordering of tools: source validation, static analysis issues, runtime errors, and then issues identified by the instructor.

4 USING PEDAL

4.1 Instructor Code Example

In Pedal, the instructor provides the Blackboard model's control, element. The example shown in Figure 4 has 7 numbered steps. The first step uses Source to read and verify that the student's code is syntactically correct. The second step uses TIFA to perform type inferencing but to suppress feedback related to changes in types. CAIT is used in the third step to inspect the code and insure that the student has not used an input statement. Step 4, also using CAIT, illustrates a mistake pattern - finding any occurrences of an iteration variable whose type is a list (this is a common mistake made by CS0 students). The type information for this pattern was produced by TIFA in step 2. Step 5 uses Sandbox to safely execute the student's code and verify run-time requirements: that a particular function was called and that output was produced. Step 6 uses Assertions to perform basic unit tests. Finally, in step 7 the Resolver selects the single, highest priority feedback message for the student.

This example is, of course, simpler than those used in practice where any number of type errors, mistake patterns, run-time checks and unit tests may be used to create feedback.

4.2 Integrating Pedal

We have integrated Pedal into three different courses: a CS0 course on Computational Thinking for non-STEM majors [16], a CS0.5 course on Python Programming for non-Computer Science majors and a CS1 course for Computer Science majors. The CS0 and CS0.5 courses were at one university and the CS1 course was at a different university. These courses integrated Pedal into environments ranging from IDEs to autograders, and scale from small programming problems to large projects.

The CS0 and CS0.5 courses used a dual block/Python programming environment[3] where the blocks are directly translatable to Python. JavaScript performs client-side execution of students' code (and Pedal). These courses have many short exercises, leading to a large set of preconfigured options. For example, TIFA automatically runs for every problem by default, and the results are typically used by other components.

In the CS0 course, students transitioned to small programming exercises in Jupyter Notebooks before tackling more open-ended projects. Therefore, we augmented Jupyter Notebooks with a custom Pedal extension. The feedback provided by this tool and for this level of student was shorter and more tactical to encourage critical thinking as they advanced through the curriculum.

A similar strategy was used for the second half of the CS0.5 course, where students submitted code to the Web-CAT grading platform. Although only a prototypical form of Pedal was used, similar organizing principles were leveraged. The feedback at this level focused more on unit testing and assertions, but other kinds of program analysis were integrated to check for project requirements and to improve the contextualization of the feedback.

The CS1 course used the autograding platform VPL in the last third of the course. Aspects of VPL's integration into its Learning Management System necessitated sectional grading so problems be worked on independently but stored within the same file. To improve the organization of the instructor-written code, separate modules were created for different problems, sections, and groups of related tests. These modules were then be imported as needed.

Integrating Pedal is a matter of simply importing it as a Python library. Pedal can be incorporated into most infrastructures designed to deliver feedback, as long as source code can be passed to Pedal.

5 BEYOND PEDAL

5.1 Evaluating Automated Feedback

Our experience with Pedal leads us to advocate for two feedback-related practices: unit testing of feedback and feedback-based instructional evaluation.

The complexity of the Pedal components and the volume of feedback cases made it difficult to reason about what feedback would be produced. This led us to create a set of sample student solutions and their expected feedback expected. In effect, this is a set of unit tests for the Pedal modules and the instructor's control code. These unit tests helped us think strategically about the kinds of mistakes students make and assure that the feedback from Pedal was what we expected. This approach can also help evaluate alternative implementations of the Resolver and assist in repurposing, by examining the feedback for a new reference solution or new control code.

Feedback informs students about mistakes in their solutions or gaps in their understanding; it also informs instructors about weaknesses in their curriculum. Frequency of feedback occurrences can signal instructional success or need for instructional changes. Relating the feedback to the instruction is best done if the instruction resulted from a principled instructional design process [12].

5.2 Beyond Source Code

In this paper, our abstract model was student source code going into an infrastructure, being analyzed for meeting a set of conditions, which then triggers a response and possibly feedback. This model can be generalized to using conditions as ways to observe student performance and producing responses that can be either turned into feedback, like in our model in figure 1, or can be used by instructors to revise curriculum. Framing the model in this way means feedback systems can be aligned with works such as [19], [12], and other educational literature that uses knowledge components in either

student analysis or instructional design. This approach lends itself to robust and deliberate revision of curriculum.

5.3 Evaluating Pedal

Being relatively new, Pedal as an infrastructure has little evaluation on its performance and extensibility. This paper demonstrates feasibility, and our local experience with its extensibility. However, evaluating Pedal's extensibility requires external adopters.

We cannot formally assess Pedal's current performance, as Pedal is a library whose performance depends greatly on its embedded environment. However, we can say that with Pedal we have supported over 1000 students across two different programming environments at two different institutions and three courses.

From classroom usage, we have used Pedal and shown learning gains in our students [11]. While the infrastructure of the feedback system doesn't necessarily relate to students' performance, it shows that using this infrastructure to tie together multiple different types of feedback together is feasible and effective.

5.4 Future of Pedal

Pedal is an evolving project. Pedal could benefit from the addition of modules for Code Coverage, Linting/style checking, and mistake detection as in PyTA [21]. Simpler programmatic interfaces or specification languages could lower the complexity of using Pedal. Pedal has been deployed in several environments including Jupyter Notebooks, BlockPy [3], and VPL; integration into other platforms (Web-CAT or GradeScope) that support package installation is viable. While Pedal was developed for Python, it can also serve as a model for a similarly constructed library for other languages.

We advocate for a larger community-based approach in two mutually supporting directions. First, Pedal serves as an initial step toward an extensible, modular infrastructure and from which tailored, more robust automated feedback systems could be efficiently created. This direction is similar to community projects oriented towards the infrastructure of learning management systems.

Second, some components of an automated feedback system require off-line tool support. In Pedal, the Mistakes module is a collection of detectable patterns that were discovered through analysis and classification of prior student solutions. Hint generation and instructor-authored systems rely on similar tools while unit testing may rely on program slicing or code coverage tools. The ecosystem of these off-line support tools is a needed counterpart to the extensible infrastructure of online modules.

The essential contribution of Pedal is the idea of an open architecture that highlights thinking systematically about the nature of feedback and more structurally about how that feedback is generated. We believe that greater attention to developing, tracking, and revising feedback is a key research problem - developing a Feedback model and architecture is therefore critical and timely. We hope that Pedal can serve as a cornerstone for this new path forward.

USING PEDAL

Pedal is free, open-source software available through regular Python installation tools (e.g., PyPi/Pip) and on GitHub (<https://github.com/pedal-edu/pedal>). Extended examples can also be found on GitHub.

ACKNOWLEDGMENTS

This work is supported in part by National Science Foundation grants DUE 1624320, DUE 1444094, and DGE 0822220.

REFERENCES

- [1] Vincent Alevan, Bruce M McLaren, Jonathan Sewall, and Kenneth R Koedinger. 2009. A new paradigm for intelligent tutoring systems: Example-tracing tutors. *International Journal of Artificial Intelligence in Education* 19, 2 (2009), 105–154.
- [2] Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. 2007. RPython: A Step Towards Reconciling Dynamically and Statically Typed OO Languages. In *Proceedings of the 2007 Symposium on Dynamic Languages (DLS '07)*. ACM, New York, NY, USA, 53–64.
- [3] Austin Cory Bart, Javier Tibau, Eli Tilevich, Clifford A Shaffer, and Dennis Kafura. 2017. BlockPy: An Open Access Data-Science Environment for Introductory Programmers. *Computer* 50, 5 (2017), 18–26.
- [4] Brett A Becker. 2015. *An exploration of the effects of enhanced compiler error messages for computer programming novices*. Master's thesis. Dublin Institute of Technology.
- [5] Hannah Blau and J Eliot B Moss. 2015. Frenchpress gives students automated feedback on Java program flaws. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*. ACM, New York, NY, USA, 15–20.
- [6] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. 1996. *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*. Wiley Publishing, Hoboken, NJ, USA.
- [7] Paul Denny, Brett A. Becker, Michelle Craig, Greg Wilson, and Piotr Banaszkiewicz. 2019. Research This! Questions That Computing Educators Most Want Computing Education Researchers to Answer. In *Proceedings of the 2019 ACM Conference on International Computing Education Research (ICER '19)*. ACM, Toronto ON, Canada, 259–267. <http://doi.acm.org/10.1145/3291279.3339402>
- [8] Tiancong Dong, Lin Chen, Zhaogui Xu, and Bin Yu. 2014. Static Type Analysis for Python. In *Web Information System and Application Conference (WISA), 2014 11th*. IEEE, New York, NY, USA, 65–68.
- [9] Micro Focus. 2019. Fortify Static Code Analysis Tool: Static Application Security Testing. <https://www.microfocus.com/en-us/products/static-code-analysis-sast/overview>
- [10] Aymeric Fromherz, Abdelraouf Ouadjaout, and Antoine Miné. 2018. Static Value Analysis of Python Programs by Abstract Interpretation. In *NASA Formal Methods Symposium*. Springer, New York, NY, USA, 185–202.
- [11] Luke Gusukuma, Austin Cory Bart, Dennis Kafura, and Jeremy Ernst. 2018. Misconception-driven feedback: Results from an experimental study. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*. ACM, ACM, Espoo, Finland, 160–168.
- [12] Luke Gusukuma, Austin Cory Bart, Dennis Kafura, Jeremy Ernst, and Katherine Cennamo. 2018. Instructional Design+ Knowledge Components: A Systematic Method for Refining Instruction. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. ACM, ACM, Baltimore, Maryland, USA, 338–343.
- [13] G. Haldeman, M. Babes-Vroman, S. Bartos, J. Shah, D. Yucht, A. Tjang, and T. Nguyen. 2018. Providing Meaningful Feedback for Autograding of Programming Assignments. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. ACM, Baltimore, MD, 278–283.
- [14] Andrew Head, Elena Glassman, Gustavo Soares, Ryo Suzuki, Lucas Figueredo, Loris D'Antoni, and Björn Hartmann. 2017. Writing Reusable Code Feedback at Scale with Mixed-Initiative Program Synthesis. In *Proceedings of the Fourth (2017) ACM Conference on Learning@ Scale*. ACM, New York, NY, USA, 89–98.
- [15] Maria Hristova, Ananya Misra, Megan Rutter, and Rebecca Mercuri. 2003. Identifying and correcting Java programming errors for introductory computer science students. In *ACM SIGCSE Bulletin*, Vol. 35. ACM, New York, NY, USA, 153–156.
- [16] Dennis Kafura, Austin Cory Bart, and Bushra Chowdhury. 2018. A computational thinking course accessible to non-stem majors. *Journal of Computing Sciences in Colleges* 34, 2 (2018), 157–163.
- [17] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2016. Towards a systematic review of automated feedback generation for programming exercises. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*. ACM, New York, NY, USA, 41–46.
- [18] Donald E Knuth. 1969. *The art of computer programming*. Vol. 1: Fundamental algorithms. Second printing.
- [19] Kenneth R Koedinger, Albert T Corbett, and Charles Perfetti. 2012. The Knowledge-Learning-Instruction framework: Bridging the science-practice chasm to enhance robust student learning. *Cognitive science* 36, 5 (2012), 757–798.
- [20] Nguyen-Thinh Le. 2016. A classification of adaptive feedback in educational systems for programming. *Systems* 4, 2 (2016), 22.
- [21] David Liu. 2017. PyTA Help Documentation. <http://www.cs.toronto.edu/~david/pyta/>
- [22] Éva Maia, Nelma Moreira, and Rogério Reis. 2012. A static type inference for python. *Proc. of DYL* 5, 1 (2012), 1.
- [23] David J Nicol and Debra Macfarlane-Dick. 2006. Formative assessment and self-regulated learning: A model and seven principles of good feedback practice. *Studies in higher education* 31, 2 (2006), 199–218.
- [24] Thomas W Price, Yihuan Dong, and Dragan Lipovac. 2017. iSnap: Towards Intelligent Tutoring in Novice Programming Environments. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. ACM, New York, NY, USA, 483–488.
- [25] Kelly Rivers and Kenneth R Koedinger. 2017. Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. *International Journal of Artificial Intelligence in Education* 27, 1 (2017), 37–64.
- [26] Anuj Ramesh Shah. 2003. *Web-cat: A web-based center for automated testing*. Ph.D. Dissertation. Virginia Tech.
- [27] Valerie J Shute. 2008. Focus on formative feedback. *Review of educational research* 78, 1 (2008), 153–189.
- [28] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated feedback generation for introductory programming assignments. *ACM SIGPLAN Notices* 48, 6 (2013), 15–26.
- [29] Marieke Thurlings, Marjan Vermeulen, Theo Bastiaens, and Sjeef Stijnen. 2013. Understanding feedback: A learning theory perspective. *Educational Research Review* 9 (2013), 1–15.