Board 421: Using a Timeline of Programming Events as a Method for Understanding the Introductory Students' Programming Process

Dr. Phyllis Jean Beck, Mississippi State University

Phyllis Beck is a blend of art and science having completed an undergraduate degree in Fine Arts at MSU and a PH.D in Computer Science where she focused on applying Artificial Intelligence, Natural language Processing and Machine Learning techniques to the engineering education space. Currently, she is working as a post-doctoral researcher at Mississippi State University in the Bagley College of Electrical and Computer Engineering. She has worked for companies such the Air Force Research Laboratory in conjunction with Oak Ridge National Labs and as an R & D Intern for Sandia National Labs conducting Natural Language Processing and AI research and was been inducted into the Bagley College of Engineering Hall of Fame in 2021.

Dr. Mahnas Jean Mohammadi-Aragh, Mississippi State University

Jean Mohammadi-Aragh is the Director of Diversity Programs and Student Development for the Bagley College of Engineering and Associate Professor in the Department of Electrical and Computer Engineering at Mississippi State University. Through her interdependent roles in research, teaching, and service, Jean is actively breaking down academic and social barriers to foster an environment where diverse and creative people are successful in the pursuit of engineering and computing degrees. Jean's efforts have been recognized with numerous awards including the National Science Foundation Faculty Early Career Development award, the American Society for Engineering Education John A. Curtis Lecturer award, and the Bagley College of Engineering Service award. Jean earned her B.S. and M.S. in computer engineering from Mississippi State University, and her Ph.D. in engineering education from Virginia Tech.

Using a Timeline of Programming Events as a Method for Understanding the Introductory Students' Programming Process

Abstract

Due to the difficulty in assessing programming skills that arise from the open-ended nature of programming, in 2017, researchers conducted a major literature review on IDE-based learning analytics. The results of this review led researchers to put forth a call to action to expand the ability of IDEs to collect and analyze different types of data. Through the development of Instrumented IDEs, we can acquire complex programming process data, however, this approach is hindered by the complexity of developing and deploying an API for multiple IDEs. This complexity and the cross-compatibility of APIs is the primary limitation in conducting cross-IDE research, followed by the inconsistent structure and collection of data and a lack of variety in the types of metrics used to instrument IDEs.

In response to the call to action, we developed a web-based IDE known as the Archimedes Platform for capturing flowcharts and a persistent trace of student programming and design data. Using this application, we conducted an investigation of intermediate students' programming process patterns using the Python programming language. Student programming event data was collected based on a custom event compression system for capturing events such as CREATE, UPDATE, DELETE, RUN_SUCCESS, RUN_FAIL, and various browser-based events for detecting external behavior, such as copying and pasting from external sources. Using this data, we seek to validate an additional IDE-based metric called the Timeline of Program Development. We define this as a sequence of events for categorize programming skills by looking at students' programming behavior and actions taken over time. A timeline of events records events such as time spent designing, writing, updating, running, or deleting code. This poster illustrates the programming process patterns captured and analyzed through the Archimedes platform. It is our hope that this data will be used as a method to better understand student's programming behavior.

1. Introduction

This poster presents research that seeks to address a call-to-action for better methods for capturing students' programming process data by expanding the type and variety of data collected, and making an effort to create a platform with an accessible unified infrastructure that uses a standardized data schema. The primary motivation for this research an overarching goal of developing a model of programming skill estimation for introductory programming that is rooted in appropriate learning theories and utilizes artificial intelligence (AI) and IDE-based learning analytics to automate the collection and analysis of the student programming process. By improving the types of data we can collect and assess, we can provide better feedback to students and instructors on one's current level of programming skill and capacity to design programming solutions. The currently proposed model of programming skill estimation consists of five dimensions, thinking processes (TP), organizational strategy (OS), design cohesion (DC), the timeline of program development (TD), and skill mastery (SM) [Beck 2020]. The primary

purpose of these five components is to facilitate the development of six metacognitive strategies: metacognitive scaffolding, reflective prompts, self-assessment, self-questioning, self-directed learning, and graphic organizers [Rum 2017]. This poster focuses on presenting the event compression system used to capture programming events within a web-based IDE and generate the timeline of program development.

2. Methods

To address the development of a timeline of programming events we developed an event compression system that generates meaningful programming events that operate across multiple levels of data collection *granularity* as given by [Ihantola 2015]. The event system captures four levels of granularity: key-stroke level data, line-level edits, execution and submissions data. File saving is automatic, and compilation is not considered at this time as the programming exercises are completed in Python.

2.1 Timeline of Events

As a student develops a program solution for a given exercise, a *timeline of events* is generated that captures the event type, the current state of the code editor, the start and end time of the event sequence, the current line number and the value of the line associated with the event. Using this data we can from a *Timeline of Program Development* that we define as a sequence of events used to categorize programming skill by capturing a sequence of students' programming events taken over time. A timeline of events is used to record events such as: create, update, delete, copy, paste, run_success and run_fail [Beck 2020].

The event system takes low-level keystroke events such as *insert, remove, carriage returns* and *cursor changes* along with code execution events, and window focus events into account to create a set of compressed events as seen in *Table 1*. A single keystroke event captures the *time* it occurs, the event type (*etype*), the *value* of the line at the time the event occurred, and the current line number (*linenum*). These events are compressed into the higher-level events as defined in *Table 1* and stored in a global events data structure when certain criteria are met to transition to a new event state such as running the code to create a *run_success* or *run_fail* event or when a cursor change is detected for *create* and *update* events.

While some events vary in the data they capture, each compressed event is recorded with event id (e*id*), a start time (*stime*), and the current state of the code editor (*state*). *Figure 1* gives an example of a CREATE event. *Figure 2* gives an example of an UPDATE event. *Figure 3* shows an example of a RUN_SUCCESS event. *Figure 4* shows an example of a RUN_FAIL event. As events are captured, compressed, and stored in the global events data structure, the sequence of events forms a time series dataset that can be structured as a timeline of programming events.

To briefly demonstrate the intended sequence of events, we can examine a simple sequence as a case study, to understand what the intended sequence of compressed events should. This example shows the editing of a simple error where the user creates and runs the code successfully, intentionally introduces an error, runs the code unsuccessfully, corrects the code,

Event Name	Event Description
CREATE	When a user first completes the creation of a new line.
UPDATE	When a user updates a previously created line.
DELETE	When a user fully deletes a line.
COPY	A user copies one or multiple lines.
PASTE	A user pastes code or comments into the editor, without doing a copy directly prior to the paste, such as copy code from another source into the editor.
COPY-PASTE	A user directly copies and pastes in the the editor with out registering any additional events in-between.
RUN_SUCCESS	The user runs the python code and no exception occurs.
RUN_FAIL	The user runs the python code and an exception occurs.
ENTER_FOCUS	The user's active window returns to the code editor.
EXIT_FOCUS	The user's focus leaves the code editor and is no longer the active window.

Table 1: List of Editor Events and their descriptions

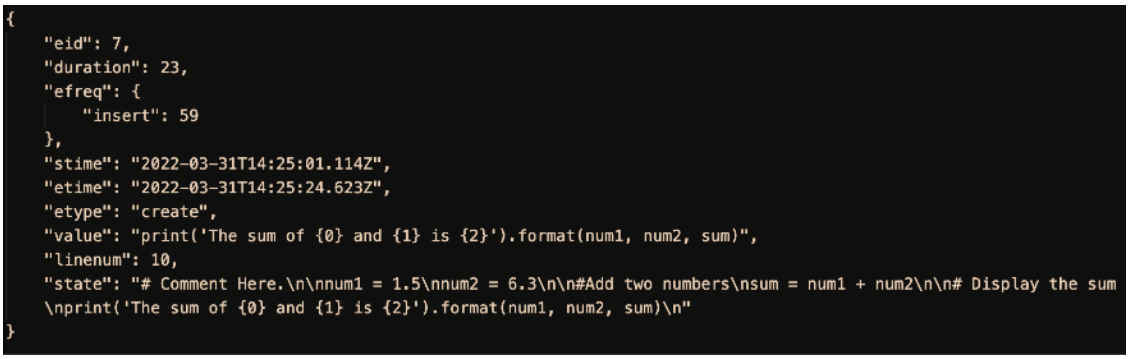

Figure 1: Compressed CREATE event example

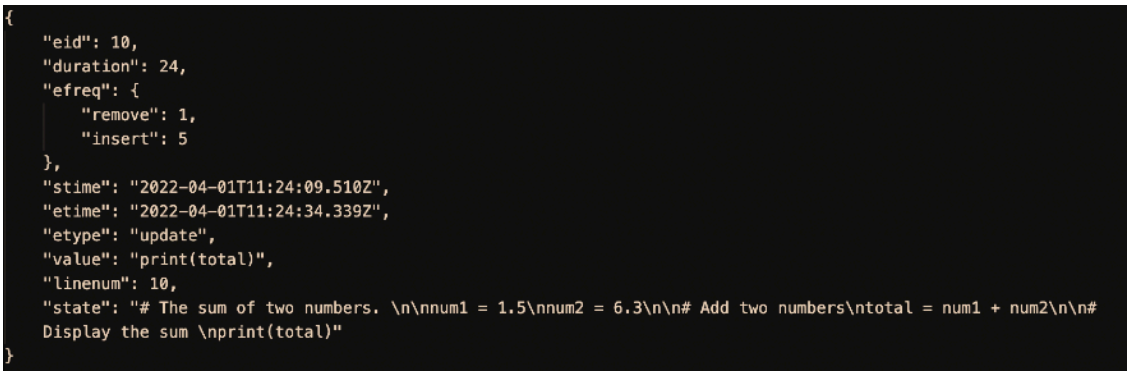

Figure 2: Compressed UPDATE event example

Figure 3: Compressed RUN_SUCCESS Event example

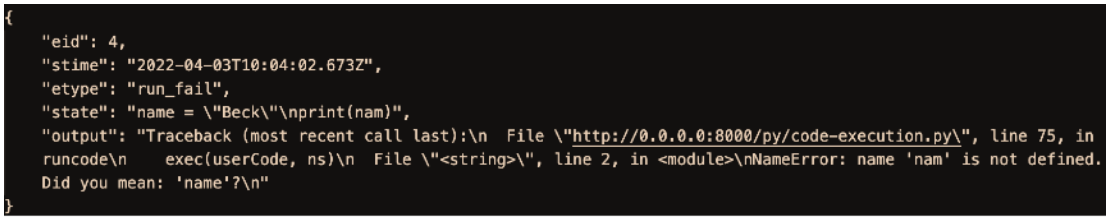

Figure 4: Compressed RUN_FAIL Event example

and then runs successfully again. Sequences like this are simple but critical for evaluating the consistency of the outcome of the compressed event sequence generated by the state machine. *Figure 5* shows the final code, *Table 2* shows the step-by-step procedure, and *Figure 6* shows the resulting compressed events data structure.

Figure 5: Final Code for Events Data Sequence Example: Correcting a simple error.

Action	Expected Event	Output
type: `name = "Beck"`		
press: ENTER	CREATE	
type: `print(name)`		
press: ENTER	CREATE	
press: RUN	RUN_SUCCESS	Beck
Edit: `print(nam)`	UPDATE	
Press: RUN	RUN_FAIL	NameError
`edit: print(name)`	UPDATE	
press: RUN	RUN_SUCCESS	Beck

Table 2: User Procedure for Correcting a Simple Error Sequence

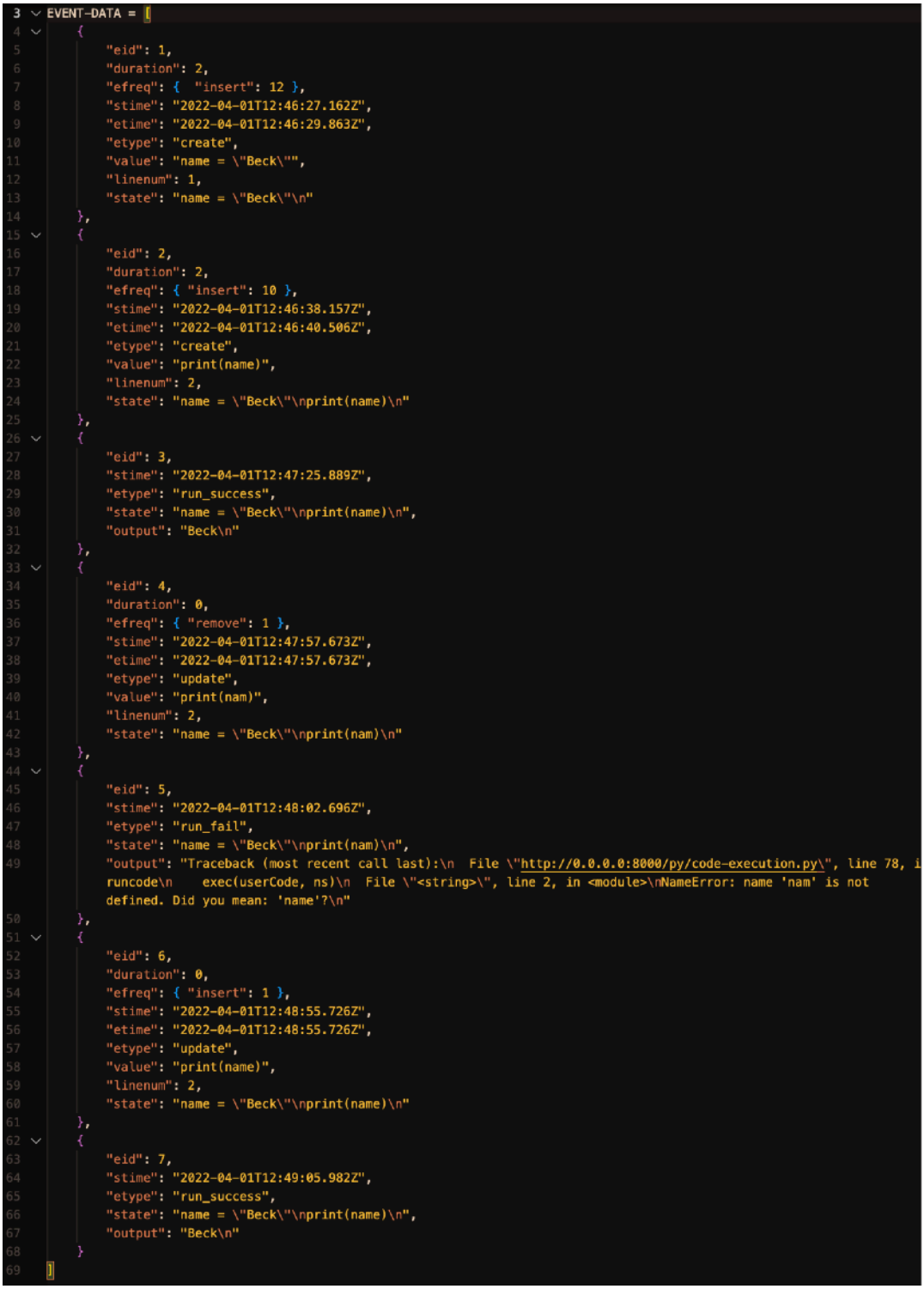

Figure 6: Sequence of events generated for Correcting a Simple Error.

This procedure generated two lines of code and seven events: two CREATE, two UPDATE, two RUN_SUCCESS, and one RUN_FAIL event. We can see that each event contains a snapshot of the code editor state (*state*) at that time. In addition to the event id, event type, and state, the CREATE and UPDATE events contain a start time (*stime*) and end time (*etime*) to capture the start and end of *insert* and *remove* events that are generated by the code editor. In the compressed

event, the frequency of those events is stored in the (*efreq*) property. Additionally, the value of the line that was either created or modified is stored in the *value* property along with its associated line number at that time. When the user runs the code there is only a single time stamp collected (*stime*). If there is any output to the console from the code execution it is captured in the *output* property, including errors when the run is unsuccessful. With this data, we can determine which programming process events occurred and when and use it to reconstruct a timeline to better understand programming behaviors and the path a user takes to arrive at a solution.

To further illustrate the sequence of compressed events in a student programming context this poster presents, as a case study, a timeline of events generated by a sample of participants for two introductory programming tasks: *commission rate* and *alternating cipher*.