A Methodology for Analyzing the Temporal Evolution of Novice Programs Based on Semantic Components

Christopher D. Hundhausen, Jonathan L. Brown, Sean Farley, and Daniel Skarpas
Visualization and End User Programming Lab
School of Electrical Engineering and Computer Science
Washington State University
Pullman, WA 99164–2752
+1 (509) 335-6602
{hundhaus, jbrown, sfarley, dskarpas}@eecs.wsu.edu

ABSTRACT
Empirical studies of novice programming typically rely on code solutions or test responses as the basis of their analyses. While such data can provide insight into novice programming knowledge, they say little about the programming *processes* in which novices engage. For those interested in improving novice programming environments, a key research question arises: How can we collect and analyze data on novice programming that will enable us (a) to analyze and compare the programming processes promoted by alternative novice programming environments, and (b) ultimately to build better novice programming environments? To address this question, we have collected a large video corpus of novices as they construct code solutions in various versions of ALVIS Live! [14], a novice programming environment. Through detailed post-hoc analyses of our video corpus, we have developed a methodology for compiling the moment-by-moment evolution of novice code solutions. Based on an analysis of an ideal code solution's key semantic components, our methodology enables one to document, on a second-by-second basis, (a) what part of a code solution a programmer is focusing on, and (b) where the semantic feedback provided by the programming environment is helping. Although it is time and labor intensive, our methodology provides researchers with a standard set of data and representations for comparing the programming processes promoted by alternative programming environments.

Categories and Subject Descriptors
K.3.2 [**Computers and Education**]: Computer and Info. Science Education H.5.2 [**Information Interfaces and Presentation**]: User Interfaces—*evaluation/methodology*.

General Terms
Algorithms, Experimentation, Human Factors

Keywords
Novice programming environments, Algorithm visualization, video analysis, programming process, semantic components

ICER '06, Oct. 1–2, 2005, Kent, UK.

1. INTRODUCTION
Beginning in the 1970s, a legacy of empirical studies have empirically investigated novice programming (e.g., see [13] for an eminent collection). Gilmore [10] identifies four main goals of this line of research:

i. to test hypotheses,

ii. to compare alternative programming environments or practices;

iii. to evaluate existing programming environments, and

iv. to explore programming behavior or technological alternatives that are not well understood.

To meet goals (i), (ii), and (iii), empirical studies have typically employed written tests or questionnaires of programming knowledge as dependent variables (see, e.g.,[25]). In contrast, meeting goal (i) has often involved analyzing think-aloud protocols of novices as they program (see, e.g., [13]).

While all of these analysis methods have their place, we have found them insufficient for studies in which one wants to test hypotheses (goal i), compare alternative environments (goal ii), or evaluate existing environments (goal iii), with respect to the novice *programming processes*, rather than *products*, promoted by novice programming environments. Indeed such investigations have the potential to shed light on several key research questions, including

RQ1. How are programmers spending their time within a given environment?

RQ2. How can a given programming environment enable a programmer to identify, fix, and avoid, semantic errors?

RQ3. How does a novice program evolve over time within a given environment?

As an illustration of the kind of analysis that could shed light on the above questions, consider Figure 1, which depicts a timeline visualization of a novice writing a solution to the problem of creating an array of random numbers, and then computing their sum. In this visualization, the x-axis depicts the time, in minutes, of the novice's programming session. The y-axis divides the

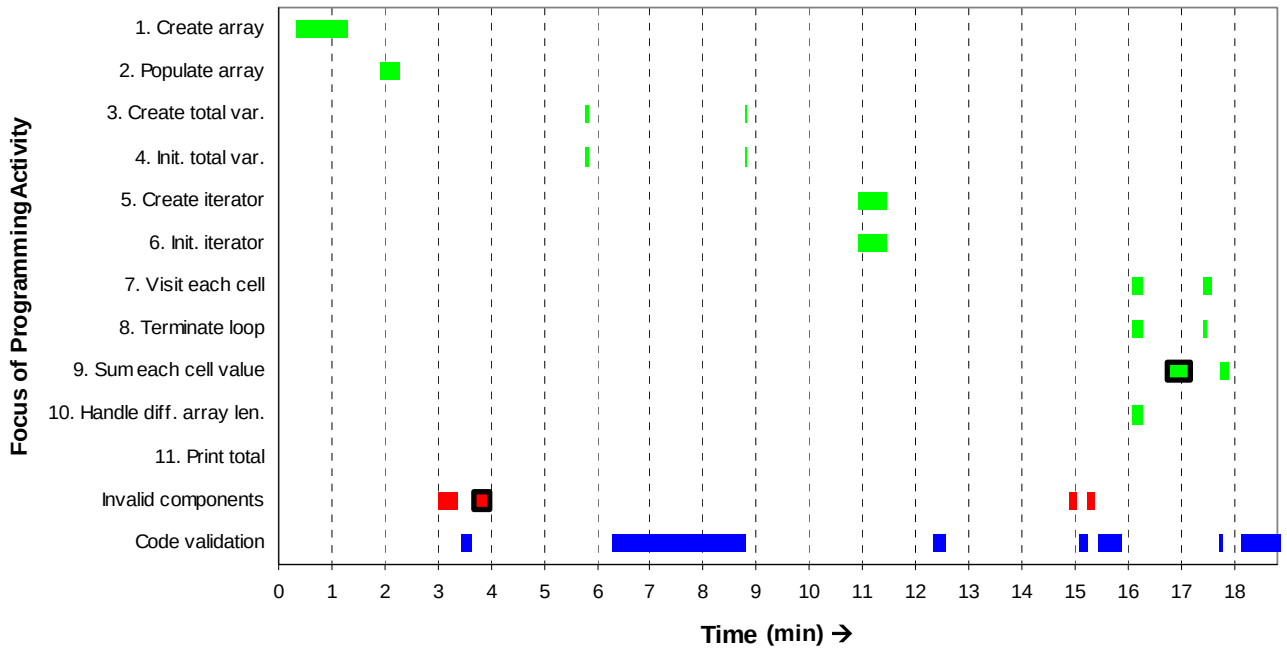

Figure 1. Timeline Visualization of a Novice's Programming Activities Leading to a Solution to the "Compute Sum" Problem

novice's programming activities into three key categories:

(a) programming activities that focus on one of 11 *semantic component*s into which we have partitioned an "ideal" solution to the "compute sum" problem (depicted as green bars)

(b) programming activities focused on semantically-invalid components (depicted as red bars); and

(c) activities in which code is explicitly validated for correctness (depicted as blue bars).

The length of each bar denotes the duration of the corresponding activity; spaces between bars (from left to right) represent periods of inactivity. Note that one of the red bars (at 3.5 min.) has a thick black outline. This denotes an invalid code component that was deleted with the help of semantic feedback from the programming environment. Note also that one of the green bars (near 17 min.) has a black outline. This denotes an episode in which semantic feedback provided by the programming environment assisted in the construction of a valid semantic component.

Thus, the timeline visualization presented in Figure 1 provides an overview of the temporal evolution of a novice's coding solution in terms of semantically correct and incorrect components. Such visualization can potentially shed light on the three research questions proposed above. More interestingly, the data from which such a visualization is generated can be used as the basis for (hypothesis-testing) studies that compare the programming processes promoted by alternative programming environments. Such studies, for which product-based and think-aloud methodologies prove inadequate, have the potential to inform, and ultimately to improve, the design of novice programming environments.

In this paper, we develop a novel methodology, based on semantic components, for collecting and analyzing videos of novice programming activity. We begin, in Section 2, by reviewing related work. Section 3 outlines the steps of the methodology, while Section 4 presents an actual case study that uses the methodology to test hypotheses regarding the appropriate frequency of semantic feedback in a novice programming environment. Finally, in Section 5, we summarize our contributions, and we propose directions for future research.

2. RELATED WORK

The difficulty of carrying out methodologically sound studies of programming has been widely lamented (see, e.g., [21]), and there have been several efforts to provide advice and guidance. In one of the earliest such efforts, Brooks [3] focused mainly on issues of experimental design, including the selection of participants, materials, tasks, and dependent measures. More recently, Shneiderman [23] described a variety of different research methods for studying programming, while Gilmore [10] presented guidelines for matching methodologies to research goals specific to empirical studies of programming. None of these methodological discussions, however, considers the problem addressed in this paper: namely, that of analyzing the actual *process* of programming for the purpose of gaining insight into how a programming environment might provide the programmer with better support.

We are certainly not the first to study the programming process in detail. For example, Gray and Anderson [11] studied programming activities in order better to understand how and

when programmers change their code. In order to characterize programming expertise, Ormerod and Ball [18] recorded experts as they wrote recursive algorithms in Prolog. Their analysis focused both on the evolution of the experts' programs, and on the experts' verbal protocols, which helped explain that evolution.

In order to study the programming process, the methodology presented here draws on two well-established research methodologies: *protocol analysis* [7] and *sequential analysis* [2]. Protocol analysis, in which single participants are asked to verbalize their thought processes as they perform tasks, has been extensively employed in the study of programming (see, e.g., [9]). As in protocol analysis, in our methodology, one records single participants as they complete programming tasks, and then performs detailed post-hoc analyses of the recordings. Unlike protocol analysis, our methodology does not have participants verbalize their thought processes, nor does it consider participants' verbalizations in the analysis process. Instead, we capture participants' programming behavior on video—in particular, their computer screens. The actions captured in such screen recordings become the focus of detailed post-hoc analyses.

Employed extensively in the study of human-human interaction, sequential analysis [2] involves coding human or animal behaviors, and then finding patterns in those behaviors. A key problem of sequential analysis is to develop coding schemes with sufficiently high inter-rater reliability [24], which ensures that different people who are trained in the scheme will code a given behavior into the same category. In our case, the behaviors to be coded are not human-human interactions, but rather human-computer interactions: namely, the editing operations within a novice programming session. (See [20] for a specialization of sequential data analysis for human-computer interaction research.)

In order to establish a common set of categories into which such editing operations are coded, our methodology requires the development of an "ideal" code solution that is broken up into fundamental semantic components. The idea of breaking up a code solution into semantic units is certainly not new (see, e.g., [5]); indeed, it is performed, in some shape or form, by programming instructors whenever they grade a programming assignment. The novelty of our approach lies in its use of a set of semantic components as a basis for analyzing novice programming *processes*, rather than products.

3. THE METHODOLOGY

Our methodology for documenting and analyzing programming processes includes five main steps:

1. constructing "ideal" solutions to the programming problems to be given to novices, and breaking those solutions up into semantic components;
2. making recordings of the programming sessions;
3. coding the recordings;
4. quantitatively analyzing the coding data in order to perform comparisons and test hypotheses; and
5. qualitatively analyzing the coding data by constructing and inspecting timeline visualizations.

In remainder of this section, we elaborate further on each of these steps.

3.1 Constructing "Ideal" Solutions

The foundation of our methodology is a means of constructing an "ideal code solution, and then breaking that solution up into semantic components. To construct an "ideal" code solution, we advocate the enlistment of a panel of at least three programming experts, who are given a problem description identical to the one that novices will receive when they perform the task. When we have applied this methodology, we have had the panel of experts construct the code solution collaboratively, arriving at the solution through a process of consensus-building. However, one could also have the experts construct their solutions independently, and then use a statistical model, such as that proposed by consensus theory [19], to derive the "culturally correct" solution.

To complete this step of the process, one needs to break the "ideal" solution up into semantic components. When we have applied this methodology, we have had the panel experts perform the semantic components analysis through a process of consensus-building. However, as with the development of the "ideal" solution, one could alternatively have the experts identify semantic components independently, and then hen use a statistical model, such as that proposed by consensus theory [19], to derive the "culturally correct" set of semantic components.

Whichever method one uses, we have found the following five guiding questions helpful in the process of identifying the semantic components in an "ideal" solution:

1. *What variable* roles [15] *are necessary in a correct solution?* In many cases, a semantic component needs to be defined in terms of a variable's role, rather than a specific line of code.
2. *To what values do variables need to be initialized?* Each variable (role) in a correct solution needs to be properly initialized in order to function properly. In many cases, we find it useful to break up a variable (role) into two semantic components: its creation and its initialization.
3. *Must the solution work for general input?* It is certainly possible to hard-code a solution that works for a specific input data set; however, an "ideal" solution most often must work for any input. To ensure that a solution is indeed general-purpose, we have found it useful to define a corresponding semantic component.
4. *How must iteration proceed?* The key elements of a correct iterative construct include (a) iterating over the correct range, (b) performing the correct operation(s) on each iteration, and (c) terminating properly. For each iterative construct in the "ideal" solution, we have found it useful to define semantic components that correspond to these key elements.
5. *What are the lines of code in the ideal solution?* This serves as the "catch-all" question. When in doubt, a semantic component will correspond roughly to a line of code.

3.2 Making the Recordings

The next step of the methodology is to collect the programming process data that will ultimately be analyzed. In many cases, such data collection will take place within the scope of an *experimental study* (see, e.g. [21]) in which one is comparing alternative versions of a novice programming environment; the programming process data will then be used to complement the programming outcomes data collected. Alternatively, one could collect process

data within the scope of a *usability study* (see, e.g., [6]) in which one is evaluating the effectiveness of a novice programming environment.

In either case, the essential data to collect are high-quality (lossless) *screen recordings* of each programmer's session. When we have applied the methodology, we have enlisted the Morae™ Recorder software [27] with great success, although any video codec capable of lossless screen recordings should work fine. Note that we regard the recording of participants' talk (audio) to be optional in our methodology; however, if recording audio is feasible in a given study, we highly recommend that it be captured as well, as it can provide a powerful resource for the coding step described next.

3.3 Coding the Recordings

Once recordings of the programming sessions of interest have been made, one is in a position to perform a post-hoc video analysis of the recordings. In this step, one carefully reviews each video recording in order to classify key participant programming actions into a set of mutually-exclusive categories that identify the following activities and events of interest:

- coding activities directed toward valid components of the "ideal" solution (i.e., the kind of coding activity that a novice programming environment would ideally encourage);
- coding activities that are *not* directed toward valid components of the "ideal" solution (i.e., the kind of coding activity that a novice programming environment would ideally discourage);
- coding activities directed toward explicitly validating the correctness of code; and
- points at which there is clear evidence that the programming environment contributed to the creation of a valid component (good), the removal of an invalid component (good), or the creation of an invalid component (bad).

Table 1 (next page) defines the 17 behavioral codes we have developed for precisely identifying the above events.[1] It is important to note that this coding scheme was developed collaboratively over an intensive two month period during which we (a) proposed possible codes; (b) independently applied the codes to sample video segments; (c) came together several times per week to discuss our experiences; (d) refined, added, and removed code definitions based on our discussions; and (e) iterated until we converged on a stable set of codes.

The actual coding of the video is the most time- and labor-intensive part of the methodology. To ease the coding process, we used Microsoft Excel® to develop coding sheet templates, along with a suite of utility scripts that ensure that only valid codes and sequences of codes are entered into coding sheets.

Figure 2 presents a sample coding sheet. Each entry in the first column denotes a primary code—one of the 17 codes depicted in Table 1. The second and third columns contain the secondary and tertiary codes, if any, associated with each primary code. For example, we identify each valid semantic component with an integer label, and we assign each invalid component a unique label, so that it can be referenced in future coding events. The fourth column provides the time at which the code was observed, rounded to the nearest second. Finally, the fourth column is reserved for coder observations.

As is the case in any type of sequential analysis [2], a key concern, in this step of the methodology, is that of ensuring the *reliability* of the coding scheme. To that end, we require at least two people who are trained in the coding scheme to independently code at least 15% of the video corpus being analyzed. One must then check the extent to which the independent coders agree by computing percent agreement and an associated kappa statistic (which factors out the extent to which the coders could have agreed by pure chance; see, e.g., [24]). In the social sciences, a kappa statistic value of 0.8 is generally considered sufficiently reliable. In our own use of the coding scheme, we achieved kappa values in excess of 0.9 after one month of training (see next section).

Given that each coding spreadsheet contains a potentially large number of temporally-ordered codes, we have found that computing the percent agreement and kappa statistic can be cumbersome. To streamline the process, we have built a set of Excel® scripts that output coding spreadsheets into a format that can be read by Coder, a publicly-available behavioral coding and analysis tool [17]. Coder then takes care of processing pairs of coding files and computing the corresponding percent agreement and kappa statistic.

3.4 Quantitatively Analyzing the Data

Recall that the primary of objective of the methodology presented here is to characterize novices' programming activities on a second-by-second basis, in order to illuminate patterns of coding activity that might indicate the effectiveness (or lack thereof) of a given novice programming environment. Toward that end, in the fourth step of the methodology, we use the set of coding sheets produced in the previous step as a basis for compiling various statistics on participants' coding activities.

Table 2 presents a set of statistics that we have found useful in answering key research questions regarding novice coding behavior with a given programming environment. Note that all of these aggregate statistics can be computed from the set of coding sheets produced in the previous steps. While computing these statistics is a relatively straightforward process, we certainly would not want to do it by hand, as it requires quite a few intermediate data structures. Hence, in our application of the methodology, we developed an Excel® script that processes a

PC	SC	TC	Time	Comment
IS	F3		0:07:28	
FI	F3		0:07:58	tooltip suggests "right" is a good name for an iterator!!
IE	F3		0:08:07	while a1[x] < a1[right]
IS	F4,F3		0:08:45	
ID	F4		0:10:08	
ID	F3		0:10:09	
IS	F5		0:10:51	
IE	F5		0:11:17	
CS	5,6		0:11:33	
CE	5,6		0:12:29	

Figure 2. Sample Coding Sheet

[1] On request, we would be happy to furnish interested researchers with (a) a 20 page coding manual that defines the coding system precisely, and (b) any of the software tools described in this section. Please contact the first author.

Table 1. Classification Scheme for Participant Programming Activities

CODE	DESCRIPTION
Task Start (TS)	The time at which the programming environment appears for the final time before the first edit of the script.
Task End (TE)	The last of (a) the time at which the participant has saved the code for the final time; or (b) the time at which the participant has run the code to the end for the final time.
Algorithm Start (AS)	The time at which the first character of algorithm code appears.
Algorithm End (AE)	The time at which the final *editing operation* on the algorithm code completes.
Valid Component Start (CS)	The time at which the participant performs the first *editing operation* directed towards the creation/modification of a statement that, in the participant's final solution, fulfills one of the valid semantic components.
Valid Component End (CE)	The time at which the participant performs the last *editing operation* directed towards the creation/modification of a valid semantic component; note that the beginning edit of this component must have been coded previously with 'Valid Component Start' code.
Valid Component Incomplete (CI)	The time at which the participant performs the last *editing operation* directed towards the creation and/or modification of an *incomplete* valid semantic component, where *incomplete* means that the component is partially satisfied by at least one *semantically correct* (full) component.
Invalid Component Start (IS)	The time at which the participant performs the first *editing operation* of an editing episode directed towards the creation of a semantic component that cannot be recognized as one valid semantic components.
Invalid Component End (IE)	The time at which the participant performs the final editing operation of an editing episode directed towards the creation and/or modification of a semantic component that cannot be recognized as a valid semantic component. **Note**: Every "Invalid Component Start" code must have either a corresponding "Invalid Component End" or "Invalid Component Delete" code, but not both.
Invalid Component Delete (ID)	The time at which the participant performs an editing operation that results in the deletion of a semantic component that cannot be recognized as a valid semantic component. **Note**: A full deletion of a statement should *always* be given an "Invalid Component Delete" code. If typing continues on the same line, this constitutes a new editing session, and should be coded as a component start.
Invalid to Valid Component Start (IVS)	*Invalid to Valid Component Start*— The time at which the participant performs the first *editing operation* of an editing episode in which an invalid semantic component is changed into a valid semantic component. **Note**: Use the "Valid Component Start" or "Valid Component Incomplete" to close this code.
Validation Start (VS)	The time at which the participant initiates the execution of at least one line of code.
Validation End (VE)	The time at which the participant stops code execution. **Note**: The number of semantic components that were validated during the validation session must be logged as well.
Feedback Leads to Error Correction	The time at which the last instance of semantic feedback appears that leads a participant to transform an invalid semantic component into a valid semantic component.
Feedback Leads to Deletion of Invalid Semantic component(FD)	The time at which the last instance of semantic feedback appears that leads a participant to delete an invalid semantic component.
Feedback Leads to Generation of New Semantic Component (FC)	The time at which an instance of semantic feedback appears that leads participants to generate a new, valid semantic component.
Feedback Leads to Generation of Invalid Semantic Component (FI)	The time at which an instance of semantic appears that leads participants to generate a new, invalid semantic component. Do not place anything in the SC or TC columns.

coding spreadsheet and builds a corresponding row of data in a master table. We then used the Analyse-It® statistical package [1] to actually compute aggregate statistics across all programming sessions in the master table. In cases in which one wants to compare alternative programming environments, the aggregate data gleaned from our coding spreadsheets can be used as the basis of statistical significance tests. It is important to note that, because most of the data we collect are percentages, they are not of interval scale, and hence violate the assumptions of parametric statistics. Hence, one must be careful to employ non-parametric tests. In most cases, non-parametric Kruskall-Wallis ANOVAs will be appropriate.

3.5 Qualitatively Analyzing the Data

The quantitative data compiled and analyzed in the previous step can be used (a) to identify general trends, and (b) to test hypotheses regarding the novice programming processes promoted by alternative programming environments; however, those data tell only part of the story. Indeed, one often wants to look for *patterns of behavior* that might shed further light on, or explain, the quantitative results. To that end, in the final step of the methodology, one generates and explores "timeline" visualizations of individual participants' coding sessions.

Table 2. Key Research Questions Posed by Our Methodology, along with Statistics That Can Be Gleaned from our Behavioral Coding Scheme to Address those Questions

RESEARCH QUESTION	SUPPORTING STATISTICS
How long do participants spend coding a solution within a given programming environment?	• *time-on-task*—the total length of the coding session, marked by the TS and TE codes.
Do participants spend their time focused on productive programming activities within a given programming environment?	• *% dead time*—the percentage of time on task during which participants neither coded nor validated code. • *% valid component editing time*—the percentage of time on task directed toward coding valid semantic components • *% invalid component editing time*—the percentage of time on task directed toward coding invalid semantic components
Are participants able to find and correct semantic errors in their code within a given programming environment?	• *% invalid components deleted or fixed*—the percentage of invalid semantic components that participants ultimately deleted or fixed
To what extent do participants explicitly validate their code's semantic correctness within a given programming environment? How long do they wait to perform such validation?	• *% validation time*—the percentage of time on task that participants explicitly validated their code. • *Avg. # components validated per validation session*—the average number of semantic components that participants validate within a validation session, where a component is considered to be "validated" if it reaches its final state prior to the validation session. • *Average validation lag time—the average time between the time a semantic component reaches its final state, and the time participants initiate a validation session.*
To what extent does semantic feedback provided by a given programming environment help or hinder coding progress?	• *% invalid components deleted or fixed via feedback*—the percentage of invalid semantic components that participants ultimately deleted or fixed with the help of semantic feedback provided by the programming environment • *% valid and invalid components generated with the help of feedback*—the percentage of valid and invalid components that were generated with the help of semantic feedback provided by a given programming environment.

In Section 1, we presented and described a sample "timeline" visualization (see Figure 1). Recall that a "timeline" visualization, as defined here, represents a participant coding session. In accordance with the distinctions made by our coding scheme, the visualization depicts each individual editing episode as a bar whose position on the x-axis indicates it start time, whose length indicates its duration, and whose color indicates its focus: either (a) a semantically correct component (green); (b) a semantically incorrect component (red), or (c) code validation (blue). In the case of semantically-correct components (green), a bar's y-axis position indicates the exact semantic component to which the editing episode is dedicated. In addition, the bar chart outlines in black any editing episode that was generated or deleted with the help of semantic feedback provided by the programming environment.

As illustrated by Figure 1, our "timeline" visualization offers a "gestalt" view of a participant's coding session. It thus provides a suitable basis for qualitative investigations of patterns of coding behavior. We will illustrate such an investigation in the case study presented in the following section.

Although the algorithm is non-trivial, it is important to note that one can generate a "timeline" visualization automatically from the coding data on any individual programming session. We have implemented an auto-generation algorithm as an Excel® script that takes an individual coding data spreadsheet as input, and generates an Excel® bar chart as output. Our algorithm was used to generate all of the sample timeline visualizations presented in this paper.

4. CASE STUDY

To illustrate the way in which this methodology might be applied in practice, we now turn to a case study from our own research. Within the context of our development of the ALVIS Live! programming environment for novices [14], we have been particularly interested in better understanding the potential for semantic feedback to benefit novice programming. To explore this issue systematically, we wanted to compare three alternative forms of semantic feedback:

- *Automatic*—semantic feedback, in the form of an updated visualization of a program's variables and data structures, is delivered automatically on every keystroke.
- *On Request*—semantic feedback, in the form of an updated visualization of a program's variables and data structures, is delivered only when explicitly requested by the programmer via an "Execute" button.
- *None*—No semantic feedback at all is available (the control condition used to establish a baseline).

We originally hypothesized that *Automatic* feedback would yield the best programming performance, both in terms of programming efficiency and semantic accuracy. Our hypothesis was based on three well-known models and theories of human-computer interaction—Norman's *seven stages of action model* [16], Shneiderman's principle of *direct manipulation* [22] and Green and Petre's 'cognitive dimension' of *progressive evaluation* [12]—each of which posit that, in order for users to be successful, they must be able to *continuously* evaluate their progress toward their goals. Because *automatic* feedback provides such evaluation

"for free," we reasoned that it would enable programmers to detect and correct semantic errors more readily.

In addition, we wanted to explore the programming processes promoted by these alternative forms of feedback. To focus our exploration, we posed the same three framing research questions as appear in Section 1, and elected to collect video data as part of an experimental study we conducted in the Spring of 2005.

Our between-subjects experimental study included three treatments defined by the three different levels of semantic feedback just described. We recruited participants for the study out of the "CS 1" course at Washington State University. The study was run within a regularly-scheduled lab period that took place near the beginning of the semester. Through a questionnaire, we screened participants for prior programming experience; any student who self-reported prior programming experience was disqualified from the study. We ultimately obtained 35 participants, whom we randomly assigned to the three experimental conditions.

Participants completed a simple programming task ("compute sum") with one of three versions of ALVIS Live! novice programming environment [14]. Each version of ALVIS Live! supported one of the three levels of semantic feedback described above. Note that participants wrote their solutions in SALSA, an interpreted "mini language" [4] with pseudocode-like syntax that supports single-procedure, array-iterative algorithms.

Having provided a brief background on our study, we use the remainder of this section to walk through the five steps of the methodology just presented. Our goal is not only to illustrate the specific ways in which those steps can be applied in practice, but also to provide empirical evidence that the methodology can actually do as it claims—that is, that it can actually provide insight into its three framing research questions.

4.1 Constructing "Ideal" Solutions

Study participants were given the following programming problem:

> *Using ALVIS, design an algorithm that creates an array containing n random integers between 1 and 100. Your algorithm should compute the sum of all the values in the array, and print out that sum.*

Applying the guidelines presented in Section 3.1, we collaboratively developed an "ideal" solution to the problem. Through much discussion and several iterations, we ultimately decomposed our "ideal" solution into the set of 11 semantic components and operational definitions presented in Table 1.

Note that we applied all four rules (see Section 3.1) in our development of the semantic components listed in Table 1. In particular, rule 1 yielded components 1, 3, and 5; rule 2 yielded components 4 and 6; rule 3 yielded component 10; rule 4 yielded components 7, 8, and 9; and rule 4 yielded component 11.

4.2 Making the Recordings

Because the computer lab in which we ran the study had only 12 computers, we ran participants through our study in three separate sessions, each of which included 11 to 12 participants. Each computer in our lab was equipped with Morae™ Recorder software [27], which captured lossless recordings of participants' computer screens as they worked.

Table 3. Semantic Components for "Compute Sum" Solution

#	DESCRIPTION	OPERATIONAL DEFINITION
1	Create array	The code must create an array with *n cells, e.g., "create array a with n cells"*
2	Populate array	The code must populate the array, e.g., "populate a with random ints between 1 and 100"
3	Create(role of) total	The code must create a variable to store the summation results. Clear evidence must exist that the variable is used to accumulate the sum.
4	Initialize (role of) total	ALVIS Live! automatically initializes the variable to 0; however, the code could set the variable to a non-zero value (incorrect)
5	Create (role of) iterator	The code must create a variable that acts an array index, e.g., "*set i to index 0 of a*"
6	Initialize (role of) iterator	ALVIS Live! automatically initializes indexes to 0; however, the code could initialize the index to another value (incorrect)
7	Looping visits each cell	The code must contain a loop that visits each cell of the array, e.g., *"while i < cells of a…add 1 to I…endwhile"*
8	Loop terminates correctly	The loop defined by the code must terminate.
9	Add cell value to total	The code must include a line that adds the current array cell to the total variable, e.g., "*add a[i] to sum*"
10	Iteration handles variable length array	The loop defined in the code must refer to "cells of a," rather than to a hard-coded value.
11	Print total	The code must contain a print command to print out the total variable.

We provided participants with directions on how to set up, start, and stop their recordings. For the most part, participants successfully recorded their work; however, a few participants failed to make proper recordings, so their data had to be discarded from our study. Ideally, the researcher would have full control over setting up, starting, and stopping the recordings. Unfortunately, in our case, this simply was not feasible, since we ran 12 participants at a time.

In addition, we would have ideally recorded participants' verbalizations (as in, e.g., [18]). Indeed, we believe that our video analysis of participants' activities would have been easier if we had had access to participants' verbalizations of what they were doing. Unfortunately, since we ran participants in groups of 11 to 12, participants would have disturbed, and possibly influenced, each other if we had requested them to think aloud. Therefore, we collected only their screen recordings.

4.3 Coding the Recordings

We collected 19.6 hours of video footage of participants as they programmed solutions to the "compute sum" problem. Three analysts (the second, third, and fourth authors) participated in a two-week training session, during which they (a) learned the coding scheme described in Section 3.3, (b) practiced coding selected video files not used in our analysis, and (c) came together to review and discuss their results. When we felt that we had reached sufficient agreement on the training videos, as evidenced by kappa values above 0.9, we had the three analysts independently code a random 20 percent sample of our 19.6 hours of video footage (roughly four hours of footage). We used the Excel®-based tools described in Section 3.3 to output our coding

sheets to the Coder software [17], which analyzed our coding files for agreement.

Across the 1,602 calibration codes in our 20 percent sample, we achieved 94.4 percent agreement, along with a kappa value of 0.936 on the primary codes, 0.964 on the secondary codes, and 0.994 on the tertiary codes. Having established that our coding scheme was sufficiently reliable, we randomly assigned an equal amount of the remaining video footage to each of our three analysts.

We estimate that the entire process—from training the coders to completing the coding of all 19.6 hours of video—required roughly one month of full-time labor. Moreover, for every hour of video, we estimate we needed 2 to 3 hours of analysis, not including the "start-up" time of training the analysts.

4.4 Quantitatively Analyzing the Data

We performed the quantitative analysis of our data in two phases, as described below.

4.4.1 Semantic Accuracy Analysis

In order to address the main hypotheses of our study, we graded participants' code solutions for semantic correctness. Our grading process had two key features in common with our video coding process. First, it was based on the 11 semantic components of our "ideal" solution. Participant solutions could receive up to a maximum of 11 points according to the number of semantic components they successfully embodied. Second, we performed a reliability check of our grading process by having two independent graders (the third and fourth authors) independently grade a 20% sample of the code solutions. Having achieved 95% agreement, we concluded that our grading system was reliable, and we proceeded to have a the third author grade the remaining 80 percent of the code solutions.

Table 4 presents our semantic accuracy results by treatment group. As can be seen, the Automatic group generally outperformed the On Request condition, which, in turn, outperformed the No Feedback condition. According to a Shapiro-Wilk test for normality, these data are not normally distributed, thus requiring us to use non-parametric Kruskall Wallis ANOVAs to test for significant differences. Consult column 5 of Table 4 for the *p*-values corresponding to each measure.

As can be seen from Table 4, although there exists no statistically-significant difference with respect to overall accuracy ($df = 2$, $H = 4.46$, $p = 0.108$, there do exist statistically significant differences with respect to three semantic components: namely creating the role of, and initializing, an iterator (SC #5 and 6), and adding each cell to the total value (SC #9). According to post-hoc Connover contrasts, the difference in all three cases lies between the no feedback condition and the two feedback conditions. In addition, we find four additional differences that trend toward statistical significance—creating and initializing the total variable (SC #3 and #4), visiting each cell (SC #7), and terminating the loop correctly (SC #8).

4.4.2 Programming Process Analysis

Are the differences we observed in semantic accuracy the result of discernable programming process differences? Is there evidence that semantic feedback actually helped participants in the two feedback conditions? To explore these questions, we used

Table 4. Semantic Accuracy Results

Measure	Treatment	*M*	*SD*	KW *p*-value
Total	Automatic	9.3	2.3	0.102
	On Request	7.9	3.4	
	No Feedback	5.5	4.0	
SC #1 (Create Array)	Automatic	1.00	—	—
	On Request	1.00	—	
	No Feedback	1.00	—	
SC #2 (Populate Array)	Automatic	1.0	—	0.34
	On Request	0.91	0.30	
	No Feedback	1.0	—	
SC #3 (Create Role of Total)	Automatic	0.92	0.27	0.083
	On Request	0.73	0.45	
	No Feedback	0.5	0.52	
SC #4 (Initialize Role of Total)	Automatic	0.92	0.29	0.083
	On Request	0.73	0.47	
	No Feedback	0.50	0.52	
SC #5 (Create Role of Iterator)	Automatic	0.92	0.29	**0.019**
	On Request	0.82	0.40	
	No Feedback	0.42	0.51	
SC #6 (Intialize Role of Iterator)	Automatic	0.92	0.29	**0.019**
	On Request	0.82	0.40	
	None	0.42	0.51	
SC #7 (Visit Each Cell)	Automatic	0.75	0.45	0.053
	On Request	0.55	0.52	
	No Feedback	0.25	0.45	
SC #8 (Terminate Loop Correctly)	Automatic	0.75	0.45	0.053
	On Request	0.54	0.52	
	No Feedback	0.25	0.45	
SC #9 (Add Each Cell to Total)	Automatic	0.83	0.39	**0.032**
	On Request	0.72	0.46	
	No Feedback	0.33	0.49	
SC #10 (Handle Variable Length Array)	Automatic	0.67	0.49	0.207
	On Request	0.64	0.50	
	No Feedback	0.33	0.49	
SC #11 (Print Total)	Automatic	0.58	0.51	0.825
	On Request	0.45	0.52	
	No Feedback	0.50	0.52	

our Excel scripts to generate a summary table of the statistics described in Table 2—one row for each participant. We then used the Analyse-It® Software [1] to test for statistical differences.

Table 5 presents the results of this analysis. Because they are not of interval scale, these data violate the assumptions of parametric statistics. We thus employed non-parametric Kruskall-Wallis ANOVAs to test for significant differences (see the fifth column for *p*-values). In inspecting these results, one immediately notices the large discrepancy between the two feedback conditions (Automatic, On Request) and the No Feedback condition with respect to time on task. This difference, which approaches statistical significance, can be explained by noting the obvious: participants in the two feedback conditions had the benefit of semantic feedback to guide their coding efforts, whereas

participants in the No Feedback condition did not. As a result, participants in the feedback conditions had more incentive to explore alternative solutions. In contrast, once participants in the No Feedback condition came up with a solution, they had no means to try to improve it, so they often quit.

With respect to how participants in the three conditions spent their time, we find it remarkable that the three conditions were within 15 percent of each other with respect to dead time, valid component editing time, and invalid component editing time. In contrast, owing to the fact that the No Feedback condition could not actually validate code, we do see significant differences between the two feedback conditions and the No Feedback condition with respect to the three validation measures; however, the two feedback conditions do not vary significantly. That the Automatic condition validated twice as many components per session, but waited twice as long to actually validate components, can be explained by the observation that most validation, in the Automatic treatment, occurred *implicitly* (by editing code and seeing the results immediately), rather than *explicitly* (by actually hitting the execute button).

Finally, we note that the role of semantic feedback did not appear to have the impact that we hypothesized it would. While one might speculate that feedback played a role in the notable differences we observed with respect to the percentage of invalid components either deleted or fixed (91.7 percent in the Automatic condition versus 61.5 percent in the No Feedback condition), we were unable to find evidence that this difference was due to feedback. Indeed, just 9.7 to 11.2 percent of the invalid components that were created in the feedback conditions were actually deleted with the help of feedback.

4.5 Qualitatively Analyzing the Data

The quantitative analysis presented in the previous section paints a discouraging picture: the two feedback conditions (Automatic and On Request) achieved higher accuracy with respect to certain semantic components; however, we found no evidence that the higher accuracy was attributable to feedback. In fact, it appears that such accuracy gains were due to *persistence*. Indeed, the feedback conditions spent substantially more time on task, and were able to correct or delete more of the invalid components that they did generate. In this final step of the methodology, we explore timeline visualizations of participants' coding sessions in an attempt to look for overall patterns of interaction that might shed further light on our quantitative results.

Let us first consider the timeline visualization of P909, a participant from the "No Feedback" condition (Figure 3). This participant's behavior typifies that of a group of nine participants who were able to program a correct solution both quickly (in under 20 minutes) and accurately (90 percent correct or better). Notice that P909 engaged in only four sessions focused on invalid semantic components. The rest of the time was spent efficiently coding the solution in short, 10 to 30 second bursts.

In stark contrast to the efficient behavior exemplified by P909, consider P207, whose behavior exemplifies that of a group of eight participants in the two feedback groups (see Figure 4). Like the group of participants described in the previous paragraph, the group of participants exemplified by P207 (an "Automatic" participant) achieved 90 percent or higher accuracy in their final solutions; however, the way in which they arrived at their solutions was far from efficient, requiring anywhere from 30 minutes to nearly two hours. Their general approach, as illustrated by Figure 4, was marked by numerous missteps (the red in Figure 4), along with frequent requests for semantic feedback (the blue in Figure 4). For this group of participants, the task did not appear to come easily; however, their persistence paid off, as participants were ultimately able to correct their missteps and converge on a correct solution.

The previous two examples illustrated successful coding patterns. Let us now examine coding two patterns that were ultimately unsuccessful. Figure 5 presents the timeline visualization of P507, an "On Request" participant. P507 exemplifies a group of five participants who, despite putting in an honest effort (from 30 minutes to over two hours), constructed solutions with low

Table 5. Programming Process Results

Measure	Treatment	*M*	*SD*	KW *p*-value
Time On Task (min.)	Automatic	45.6	40.1	0.057
	On Request	39.5	39.8	
	No Feedback	16.1	9.8	
% Dead Time	Automatic	42.5	11.1	0.163
	On Request	47.1	13.9	
	No Feedback	55.6	17.8	
% Valid Component Editing Time	Automatic	11.4	7.4	0.277
	On Request	11.8	11.3	
	No Feedback	16.5	9.5	
% Invalid Component Editing Time	Automatic	34.4	13.7	0.250
	On Request	21.7	18.8	
	No Feedback	28.0	21.1	
% Invalid Components Deleted/Fixed	Automatic	90.7	12.3	0.312
	On Request	78.5	30.5	
	No Feedback	61.5	42.3	
% Validation Time	Automatic	13.0	10.0	**<0.0001**
	On Request	20.5	6.8	
	No Feedback	0.0	—	
Avg. # Comp. Validated per Session	Automatic	2.3	2.1	**<0.0001**
	On Request	1.3	1.5	
	None	0.0	—	
Avg. Validation Lag Time (min.)	Automatic	5.1	4.3	**<0.0001**
	On Request	1.9	1.6	
	No Feedback	0	—	
% Invalid Comp. Deleted/Fixed via Feedback	Automatic	9.7	19.4	0.312
	On Request	11.2	9.0	
	No Feedback	0	—	
% Valid Comp. Generated via Feedback	Automatic	0.0	—	—
	On Request	0.0	—	
	No Feedback	0.0	—	
% Invalid Comp. Generated via Feedback	Automatic	0.0	—	—
	On Request	0.0	—	
	No Feedback	0.0	—	

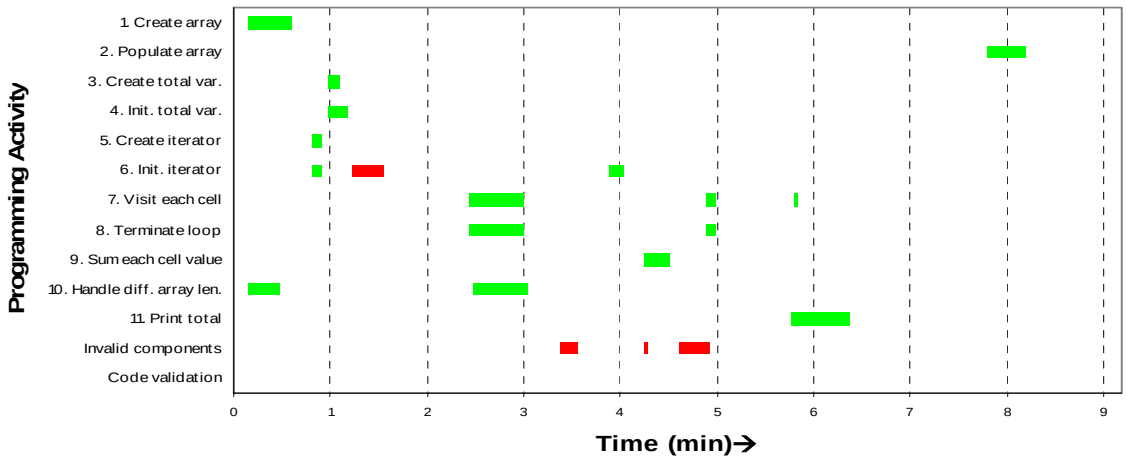

Figure 3. P909, a "No Feedback" Participant. Completes Solution with Few Missteps

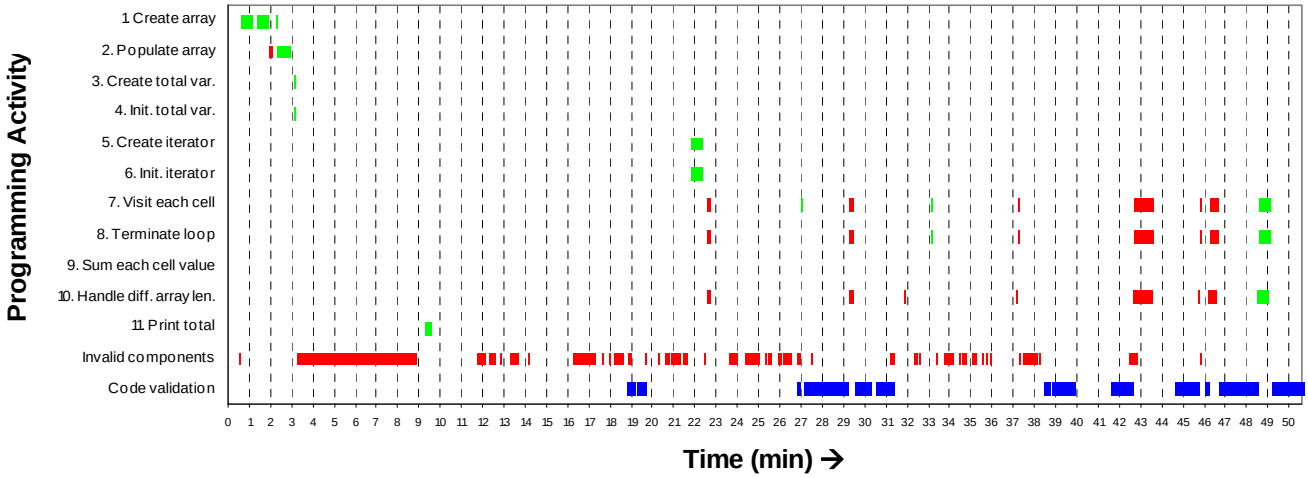

Figure 4. P207, an "Automatic" participant, Succeeds Through Persistence

accuracy (from 18 to 63 percent correct). Like the behavior of the participants who succeeded through persistence, the behavior of this group of participants was marked by frequent missteps and frequent requests for semantic feedback. In contrast to the participants who succeeded through persistence, however, this group of participants could do little more than construct a few of the necessary program objects. They spent the rest of their time locked in a pattern of invalid coding interleaved with validation (see alternating red and blue bars at the bottom of Figure 5).

Finally, Figure 6 presents a second example of unsuccessful behavior. The behavior of P907, a "No Feedback" participant, is typical of that of seven participants who coded a few components at the beginning of their sessions, and then quickly gave up (in 12 minutes or less). Not surprisingly, five of these seven participants used the "No Feedback" interface.

In sum, through our qualitative analysis of timeline visualizations, we were able to identify four distinct behavior patterns that characterize the behavior of 29 of our 35 participants (83 percent). This characterization, which paints a high level picture of how various groups of participants proceeded with the programming task, complements the quantitative results we obtained in the previous step.

5. SUMMARY AND FUTURE WORK

Drawing on both protocol analysis [8] and sequential analysis [2], this paper has proposed a new methodology for analyzing novice programming processes. The novelty of our methodology lies in the way in which it frames programming activity: as a time-ordered sequence of editing episodes focused on the semantic components of an "ideal" solution. As we have illustrated through a detailed case study, semantic components provide a potentially valuable lens through which to view novice programming behavior, because they are able to characterize the impact of each individual editing episode on the final solution.

In addition, our methodology provides a means for capturing

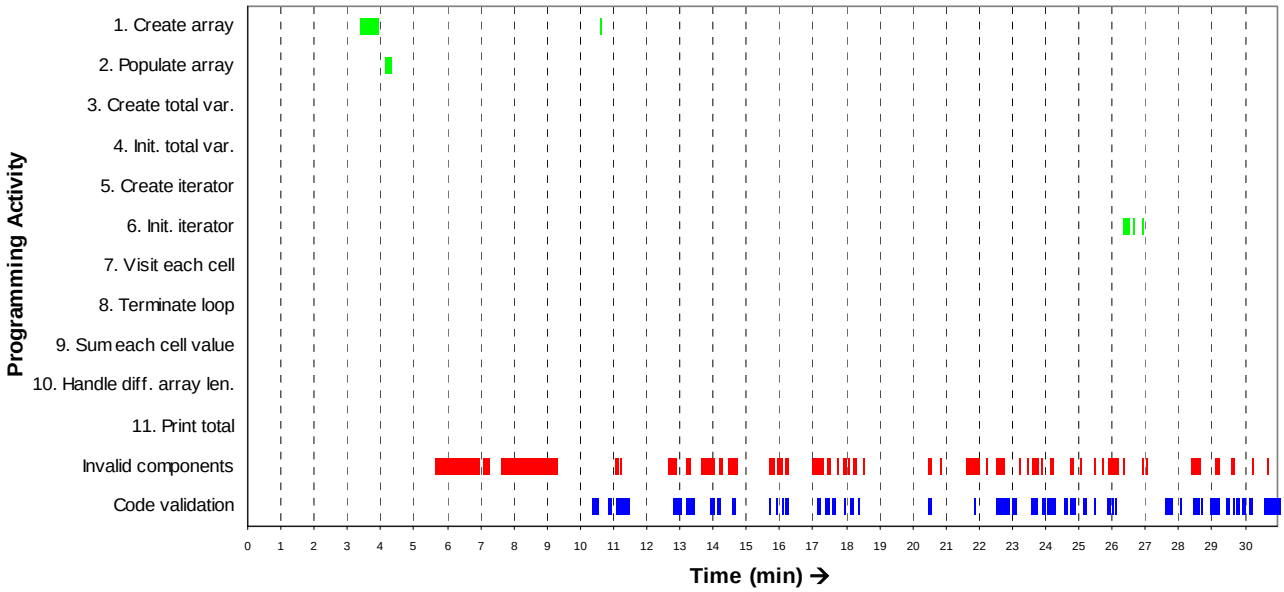

Figure 5. P507, An "On Request" Participant, Cannot Get On Track, Despite Honest Effort

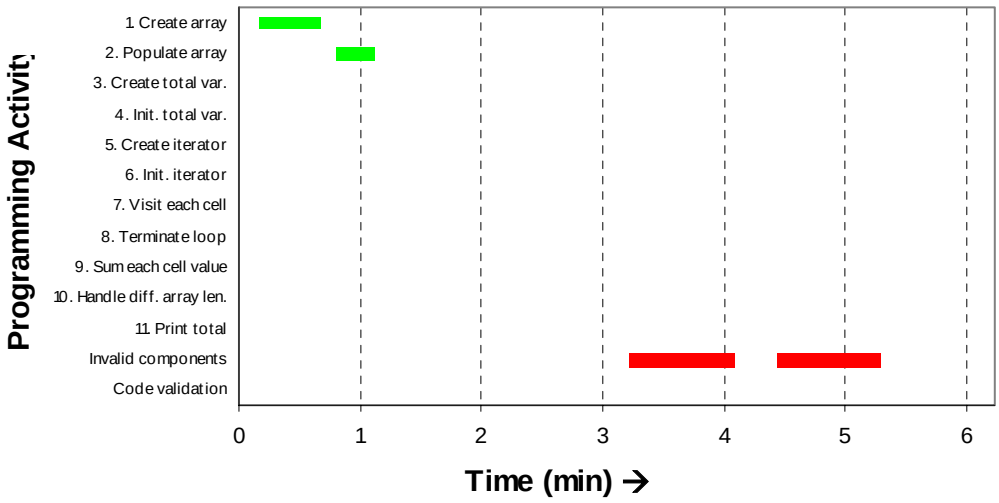

Figure 6. P907, a "No Feedback" Participant, Gives Up Quickly

the extent to which semantic feedback provided by the programming environment helps and hinders the programming process. Thus, our methodology can prove useful for the study of novice programming effectiveness, providing a standard set of measures for comparing environments across studies.

As suggested by our case study, the methodology has several limitations. Most significantly, it requires a substantial investment of time and labor. Indeed, in order to code and analyze the data presented in Section 4, four people had to work full-time for over a month. Given the documentation, coding system, and software that we have developed to support this methodology, we anticipate that we might be able to reduce our effort by one third in future studies; however, for those new to the coding system, there will always be at least a two week startup period required to learn the coding system, and to develop the common understanding necessary to achieve sufficiently high inter-rater agreement.

Given that our behavioral coding system is so labor intensive, one might wonder whether it is amenable to automation. It seems plausible to us that, with software logs, one could algorithmically determine when editing episodes start and stop. However, the actual classification of events is, by its very nature, subjective. Indeed, we spent a substantial amount time coming up with reliable definitions,. Moreover, it took a considerable amount of time and effort for us to achieve sufficiently high reliability. Even with software logs, we are skeptical that an algorithm could classify events reliability.

Besides exploring the potential for a higher degree of automation, we plan, in future research, to develop a more comprehensive classification scheme for *invalid* components. Our present coding system says a lot about the nature of valid semantic components, but next to nothing about the nature of invalid components, which actually proved to be more prevalent than valid semantic components in our case study. In our timeline visualizations, what is actually going on during those episodes marked by red bars? One could imagine a system that classified invalid components according to an established taxonomy of novice programming errors (e.g., [26]). Indeed, in future work, a valuable contribution would be to develop a reliable system for characterizing the temporal evolution of semantic errors—one analogous to the system presented here for classifying valid semantic components.

6. ACKNOWLEDGMENTS

We are grateful to all the CptS 121 students at Washington state University who enthusiastically participated in our studies of the ALVIS Live! programming environment, along with their instructor, Andy O'Fallon, who agreed to allow us to conduct these studies as part of regular CptS 121 labs. This research is funded by the National Science Foundation under grant nos. 0406485 and 0530708.

7. REFERENCES

[1] Analyse-It Software, Ltd. Analyse-It® Statistical Add-In for Excel®, 2006, http://www.analyse-it.com.

[2] Bakeman, R. and Gottman, J.M. *Observing Iteraction: An Introduction to Sequential Analysis*. Cambridge University Press, Cambridge, England, 1996.

[3] Brooks, R.E. Studying programmer behavior experimentally: the problems of proper methodology. *Communications of the ACM, 23*, *4*. 208-213.

[4] Brusilovsky, P., Calabrese, E., Hvorecky, J., Kouchnirenko, A. and Miller, P. Mini-languages: A way to learn programming principles. *Education and Information Technologies*, *2*. 65-83.

[5] Douglas, S.A., Hundhausen, C.D. and McKeown, D. Toward empirically-based software visualization languages. In *Proceedings of the 11th IEEE Symposium on Visual Languages*, IEEE Computer Society Press, Los Alamitos, CA, 1995, 342-349.

[6] Dumas, J.S. and Redish, J.C. *A Practical Guide to Usability Testing*. Intellect Books, Portland, OR, 1993.

[7] Ericsson, K.A. and Simon, H.A. *Protocol Analysis: Verbal Reports as Data*. MIT Press, Cambridge, MA, 1984.

[8] Ericsson, K.A. and Simon, H.A. Verbal reports as data. *Psychological Review, 87*. 215-251.

[9] Fisher, C. Advancing the study of programming with computer-aided protocol analysis. In *Empirical studies of programmers: Second Workshop*, Ablex, Norwood, NJ, 1987, 198-216.

[10] Gilmore, D.J. Methodological issues in the study of programming. In Hoc, J.-M., Green, T.R.G., Samurcay, R. and Gilmore, D.J. (eds.), *Psychology of Programming*, Academic Press, San Diego, 1990, 83-98.

[11] Gray, W.D. and Anderson, J.R. Change-Episodes in Coding: When and How Do Programmers Change Their Code? In *Empirical Studies of Programmers: Second Workshop*, 1987, 185-197.

[12] Green, T.R.G. and Petre, M. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Langauges and Computing, 7*, *2*. 131-174.

[13] Hoc, J.-M. and Nguyen-Xuan, A. Language semantics, mental models and analogy. In Hoc, J.-M., Green, T.R.G., Samurcay, R. and Gilmore, D.J. (eds.), *Psychology of Programming*, Academic Press, San Diego, 1990, 139-156.

[14] Hundhausen, C.D. and Brown, J.L. What You See Is What You Code: A 'Live' Algorithm Development and Visualization Environment for Novice Learners. *Journal of Visual Languages and Computing*.

[15] Kuittinen, M. and Sajaniemi, J. Teaching roles of variables in elementary programming courses. In *Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education*, ACM Press, New York, 2004, 57-61.

[16] Norman, D.A. *The Design of Everyday Things*. Doubleday, New York, 1990.

[17] Oregon Social Learning Center. Coder, 2006, http://www.oslc.org/Training/download_coder.html.

[18] Ormerod, T.C. and Ball, L.J. Does Programming Knowledge or Design Strategy Determine Shifts of Focus in Prolog Programming? In *Empirical Studies of Programmers: Fifth Workshop*, 1993, 162-186.

[19] Romney, A.K., Weller, S.C. and Batchelder, W.H. Culture as consensus: A theory of culture and informant accuracy. *American Anthropologist, 88*, *2*. 313-338.

[20] Sanderson, P.M. and Fisher, C. Exploratory sequential data analysis: Foundations. *Human-Computer Interaction, 9*, *3-4*. 251-318.

[21] Sheil, B.A. The psychological study of programming. *ACM Computing Surveys, 13*, *1*. 101-120.

[22] Shneiderman, B. Direct manipulation: a step beyond programming languages. *IEEE Computer, 16*, *8*. 57-69.

[23] Shneiderman, B. Empirical Studies of Programmers: The Territory, Paths, and Destinations. In *Empirical Studies of Programmers*, 1986, 1-12.

[24] Shrout, P.E. and Fleiss, J.L. Intraclass correlations: Uses in assessing rater reliability. *Psychological Bulletin, 36*, *2*. 420-428.

[25] Soloway, E. and Ehrlich, K. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering, SE-10*, *5*. 595-609.

[26] Spohrer, J.C. and Soloway, E. Analyzing the High Frequency Bugs in Novice Programs. In *Empirical Studies of Programmers*, 1986, 230-251.

[27] TechSmith. An overview of Morae: A breakthrough in usability testing and user experience research, 2006, http://www.techsmith.com/morae/whitepaper/overview.asp.