Understanding Software Variation: Experiment Plan

Miles Van de Wetering and Eric Walkingshaw

1 Experiment Goals

Software variation is often difficult to comprehend - even more abstract than conditional execution, variational programming changes the very code generated at compile time. it is also very important to understand - variation is used in widespread production software. We are expanding upon previous efforts to design an editor that supports a visual representation of variation. In this study, we will focus on two different metrics: how easy it is for a user to *understand* variation in existing code, and how easy it is for a user to *correctly modify* variation. We will measure these two concepts through a variety of goals:

Goal 1: Determine whether users can *more accurately* deduce the *number of variants* represented in code presented in our prototype compared to code annotated with CPP.

Goal 2: Determine whether users can *more quickly* deduce the *number of variants* represented in code presented in our prototype compared to code annotated with CPP.

Goal 3: Determine whether users can conduct a *feature subtraction* operations more quickly and accurately.

Goal 4: Determine whether users can conduct a *feature addition* operations more quickly and accurately.

Goal 5: Determine whether users consider the prototype to be *more understandable* than code annotated with CPP.

Goal 6: Determine how users' *execution comprehension* (ability to reason about a program) is affected.

2 Participants

For our experiment we plan to recruit at least 35 Computer Science undergraduate students (the minimum of 30 required for statistical significance, plus 5 in case of no-shows). Subjects will be recruited through BeaverSource and the EECS mailing list. Participants will be compensated $20.

Potential subjects must take a brief screening test before signing up for the study. This will be used to confirm a basic understanding of C and CPP, and only students that pass the screening test will be asked to take part in the study. We anticipate recruiting problems with such strict prerequisites. If the screening procedure eliminates too many students or too few attempt to register in the first place, we will expand our population to include non-Computer Science students with programming experience (such students could be found, for example, through the Linux Users mailing list). If we still do not have enough participants, we will attempt to simplify the screening test and extend the tutorial accordingly.

Our experiment will be performed within-subjects, so all participants will undergo all treatments. The ordering of tasks will be randomized, however, along with the tasks themselves (see Section 6). All participants in the same experiment session will perform the tasks in the same order, and the tasks and ordering of tasks will be randomized per experiment session.

The return value of the C program at right depends on whether or not the C Preprocessor macros `A` and `B` are defined when the code is compiled. For each question below, write the return value if the code is compiled with the given macro settings.

- A = undefined, B = undefined:
- A = undefined, B = defined:
- A = defined, B = undefined:
- A = defined, B = defined:

Figure 1: C and CPP knowledge screening test.

3 Experiment Materials

Prior to participating in the experiment, potential subjects will submit a registration form containing a screening test. The registration form will collect the student's name, email address, phone number, major, year of study, and two questions confirming that the student has a basic understanding of C and CPP. The accompanying screening test is designed to confirm the student's responses to these questions and is shown in Figure 1.

The prototype tool will be implemented as an Atom plugin using Javascript. The experiments will be administered in a lab setting, on provided computers (20% more computers than the expected number of participants in each session, in case of technical difficulties), via the Atom editor plugin. The CPP annotated code will be presented in it's own file, and will include syntax highlighting. The prototype tool will be implemented using HTML and Javascript. Questions will be presented to the user in a separate online survey which they will progress through after completing each task..

The code that will be used in the tasks themselves is included in the Appendix. There are three examples: one implementing playing cards, one implementing an alarm clock, and one implementing the Fibonacci sequence. One of these will be used in the tutorial (currently, the alarm clock), and the other two will be used in the tasks. Because one example may be inherently more confusing than another, we will make prototype and CPP versions of both remaining examples and randomly determine which combination is used for each session.

At the beginning of the experiment session, a tutorial will be verbally administered. Users will have minor tasks to perform during this tutorial. These will be done on the same computers and in the same way as the subsequent tasks.

3.1 Logging

The local web servers will produce log files which contain the data necessary for our analysis. Table 1 describes the structure of the log file that will be generated during CPP related tasks. Because our interface for these tasks is so simple, there are only two possible events. A *QuestionStarted* event occurs when

Field	Description
TimeStamp	Time that the event occurred.
SubjectID	Unique subject identifier.
TaskID	Task number.
QuestionID	Question number within a task.
EventType	*QuestionStarted* or *AnswerSubmitted*.
Answer	The submitted answer, if applicable.

Table 1: Structure of log file for CPP-related tasks.

Field	Description
TimeStamp	Time that the event occurred.
SubjectID	Unique subject identifier.
TaskID	Task number.
QuestionID	Question number within a task.
EventType	*QuestionStarted*, *AnswerSubmitted*, or *TagSelected*.
Parameter	If *AnswerSubmitted*, the submitted answer.
	If *TagSelected*, the new set of selected tags.

Table 2: Structure of log file for prototype-related tasks.

a subject loads a page containing a new question, and an *AnswerSubmitted* event occurs when the subject submits the form containing the answer. The *AnswerSubmitted* event contains the answer the subject entered, which we will use to determine the accuracy of the subject's understanding. Both events contain timestamps, which we can take the difference of to determine the amount of time it took to answer the question. In order for this strategy to work effectively, we will need to precede each question page with a "Start Question" page that both reminds the user that the questions are timed, and gives them an un-timed place to rest briefly.

Table 2 describes the structure of the log file that will be generated during the prototype related tasks. The structure is almost identical to the log file for CPP tasks, except that we also add a *TagSelected* task that is logged whenever a user selects a new tag in the prototype interface, changing the currently visible variant. When this event occurs, we also record the new set of currently visible tags. Strictly speaking, only the just-selected tag is needed to replay the subject's actions, but we record all currently visible tags just to provide a bit of redundancy in case the user manages to get into an unexpected state (for example, by using the browser's "Back" button, although we will probably try to disable this particular feature).

3.2 Post-Task Questionnaire

Finally, a post-task questionnaire will be given so that we can assess the perceived usefulness of the prototype compared to CPP annotations. This questionnaire is provided in Figure 3.2. We begin by asking a few questions about the tasks themselves. Knowing how difficult the users found the tasks will help us to interpret our other results. We then ask the subjects to directly compare the two systems for a few specific understanding tasks. While we use a Likert scale for several questions in the questionnaire, we use a semantic differential scale for the direct comparison questions, in order to avoid as much bias as possible. We then ask some questions specific to one system or the other, and some questions which try to determine whether or not the subject believes they have a grasp on the concept of dimensions. Finally, we end with gusto, with a basic "which is better" question, in which we force the user to commit to one of the systems.

Post-Task Questionnaire
Subject ID:

Please circle your response to each question.

1. In general, the tasks I completed in this experiment were difficult.
   strongly agree | agree | do not agree or disagree | disagree | strongly disagree
2. In which system was it easier to determine *how many* different programs were represented by the code?
   much easier in CPP | easier in CPP | same difficulty in both | easier in prototype | much easier in prototype
3. In which system was it easier to understand *a particular* program generated from the code?
   much easier in CPP | easier in CPP | same difficulty in both | easier in prototype | much easier in prototype
4. In which system was it easier to see how the different programs were *related* to each other?
   much easier in CPP | easier in CPP | same difficulty in both | easier in prototype | much easier in prototype
5. When looking at the CPP annotated code, I could tell which macros were meant to be mutually exclusive (only one can be selected at a time).
   strongly agree | agree | do not agree or disagree | disagree | strongly disagree
6. I think that I could translate the CPP annotated code into an equivalent representation in the prototype.
   strongly agree | agree | do not agree or disagree | disagree | strongly disagree
7. In general, I found the CPP annotated code easy to understand.
   strongly agree | agree | do not agree or disagree | disagree | strongly disagree
8. I think that I could translate the code in the prototype into an equivalent CPP annotated program.
   strongly agree | agree | do not agree or disagree | disagree | strongly disagree
9. In general, I found the prototype easy to understand.
   strongly agree | agree | do not agree or disagree | disagree | strongly disagree
10. Overall, which tool do you think makes it easier to understand software variation (code that represents many different programs)?
    CPP | the prototype

Figure 2: Post-Task Questionnaire

4 Tasks

Each session will consist of two tasks: one CPP task, and one prototype task. Throughout the CPP task, subjects will be shown one of the pieces of code from the Appendix in a web browser window, with syntax highlighting. The task consists of answering several questions as quickly and accurately as possible. The code is displayed only while answering a question. In between questions, subjects will be shown a wait screen that reminds them (subtly) that responses are timed, and prompts them to click a link to start the next question when they are ready.

The prototype task proceeds similarly, except that the HTML/Javascript prototype replaces the static code in the question-answering window. Since we will be comparing the accuracy and speed of the subjects' answers in both tasks, the questions will of course be mostly the same. However, there are some questions which are specific to the CPP task. These do not relate directly to the primary research questions presented here, but will be helpful for our secondary research questions described elsewhere. Questions will also sometimes require minor rewording between tasks (for example, changing "CPP" to "the prototype"). Finally, questions will sometimes be phrased such that they answer a previous question in the sequence. This will hopefully prevent one incorrect answer from snowballing into several.

Below is a sequence of questions that will be provided for the playing cards example used for the CPP task. These questions can be easily adapted to the other example and task. Questions that are CPP-specific will be marked with an asterisk (*).

Q1: How many variants can be generated from this code?

Q2: Suppose that we fix the `AcesLow` macro to be defined, how many variants can be generated given this constraint?

Q3: How many different CPP macros are present within this code?*

Q4: Given that there are three different macros, and that macros can be either defined or undefined, how many different ways can these macros be set at compile time?*

Q5: Because there are six different ways to assign these macros, there are a total of six different variants that can be generated from this code. How many do you think the programmer *intended* to define? Why? (Open answer, untimed).*

Q6: What is the output of this program if the `AcesLow` and `Verbose` macros are defined, but the `AcesHigh` macro is undefined?

Q7: What is the output of this program if the `AcesLow` macro is defined, but the `Verbose` and `AcesHigh` macros are undefined?

Q8: What is the output of this program if the `AcesHigh` and `Verbose` macros are defined, but the `AcesLow` macro is undefined?

5 Hypotheses and Variables

Following are the semi-formalized and English versions of the null and alternative hypotheses that follow from our listed goals in Section 1. For each Goal i, the corresponding null hypothesis is labeled H_{i0}, and the corresponding alternative hypothesis is labeled H_{i1}. We use $a_p(\cdot)$ to represent the average accuracy of a prediction p and $t_p(\cdot)$ to represent the average question response time for a prediction p. Predictions can be either n, for the number of variants represented in some code, or b, for the predicted behavior of some particular variant. As arguments to a and t, we use C to represent questions about code represented in CPP, and P to represent questions about code represented in the prototype. Finally, we use comparison operators like $<$ as shorthand for phrases like, "is statistically significantly less than", and $u(\cdot)$ to represent the subjects post-experiment rating of the understandability of each tool.

H_{10}: $a_n(P) \ngtr a_n(C)$. Users *do not* predict the number of variants more accurately when using the prototype than when looking at CPP-annotated code.

H_{11}: $a_n(P) > a_n(C)$. Users predict the number of variants more accurately when using the prototype than when looking at CPP-annotated code.

H_{20}: $t_n(P) \nless t_n(C)$. Users *do not* predict the number of variants in less time using the prototype than when looking at CPP-annotated code.

H_{21}: $t_n(P) < t_n(C)$. Users predict the number of variants in less time using the prototype than when looking at CPP-annotated code.

H_{30}: $a_b(P) \ngtr a_b(C)$. Users *do not* describe the behavior of a particular variant more accurately when using the prototype than when looking at CPP-annotated code.

H_{31}: $a_b(P) > a_b(C)$. Users describe the behavior of a particular variant more accurately when using the prototype than when looking at CPP-annotated code.

H_{40}: $t_b(P) \nless t_b(C)$. Users *do not* describe the behavior of a particular variant in less time when using the prototype than when looking at CPP-annotated code.

H_{41}: $t_b(P) < t_b(C)$. Users describe the behavior of a particular variant in less time when using the prototype than when looking at CPP-annotated code.

H_{50}: $u(P) \ngtr u(C)$. Users *do not* rate the prototype as more understandable than code annotated with CPP.

H_{51}: $u(P) > u(C)$. Users rate the prototype as more understandable than code annotated with CPP.

The notation described and used above reveals the structure of the major variables in our experiment. The dependent variables—accuracy, response time, understandability rating—are represented as functions of the independent variables of the treatment group (CPP or prototype) and task type (variant counting or understanding). Other major independent variables not reflected in the above equations are the example used for each treatment group (Fibonacci or playing cards) and the order in which the treatment groups are presented (CPP first or prototype first). In Section 6 we will show how we use randomization to hopefully mitigate the effects of these "uninteresting" independent variables.

6 Experiment Design

Our experiment will be conducted *within subjects* to maximize statistical power with an expected small number of subjects.

All uninteresting/potentially confounding independent variables will be distributed and randomized as much as possible. Specifically, we will have four different groups representing the four possible ways of instantiating our two most significant uninteresting independent variables: the example used in each treatment group (Fibonacci or playing cards), and the order that the treatment groups are presented (CPP first or prototype first). The four possibilities are represented in Table 3.

	Fibonacci	Cards
First Task	CPP	CPP
Second Task	Prototype	Prototype

Table 3: Four experiment groups, to distribute potentially confounding variables.

For pragmatic reasons, we will assign all subjects in a particular session to the same experiment group. Therefore, we will aim to have either four or eight sessions in order to distribute subjects across the four groups evenly. We will randomly assign each groups to sessions within these constraints.

All other potentially confounding variables, such as differing experience levels and intelligence, should be mitigated by randomly assigning subjects to sessions, by the screening test, and by treating all subjects with the same introductory tutorial.

7 Experiment Procedure

Subjects will be sent an email reminder one day before their scheduled session. Subjects will also be called the evening before their scheduled session to confirm their participation. Two hours are allotted for each experiment session, scheduled as follows.

0:00 Participants arrive at the testing room and are randomly assigned to a computer. Driver/helper enters subjects' IDs into computer as they are seated. Tutorializer calls participants that have not arrived by the scheduled start time to confirm that they are still coming.

0:05 Door closed.

0:05–0:20 Informed consent form read and signed. Confirm that subjects' IDs are on consent forms.

0:20–0:40 Give tutorial (tutorializer: Eric, driver: Duc).

0:40–0:50 Questions and extra time in case of technical problems.

0:50–0:55 Introduction to Task 1.

0:55–1:10 Task 1.

1:10–1:15 Introduction to Task 2.

1:15–1:30 Task 2.

1:30–1:40 Post-task questionaire.

1:40–2:00 Pay subjects and get receipts.

2:00 Retrieve log files and get final screenshots.

More specific details of the procedure are described throughout this plan, but particularly in Section 3, which describes how and what data is captured in log files and the post-task questionnaire.

8 Analysis Procedure

The data that is required to evaluate each of our hypotheses follows from the formalized hypotheses provided in Section 5 and the discussion of experiment materials in Section 3. For each dependent variable in the hypotheses, we can trace a path back to the data that allows us to measure that variable. The results are summarized in Table 4.

Hypotheses	**Dependent variable**	**Data source**
$H_{10}, H_{11}, H_{30}, H_{31}$	Answer accuracy	Log file: answers in *AnswerSubmitted* events
$H_{20}, H_{21}, H_{40}, H_{41}$	Response time	Log file: difference in TimeStamp of corresponding *QuestionStarted* and *AnswerSubmitted* events
H_{50}, H_{51}	Understandability	Post-experiment questionnaire: answers to question 11

Table 4: Data sources to evaluate each hypothesis.

We have not yet considered (nor discussed in class!) how to evaluate these hypotheses statistically.

Appendix: Code Listings

8.1 cards.c

```
int do_interesting_math(int n) {
    int x = 0;
    int y = 1;
    while(x < n) {

      #ifdef B
        #ifdef C
        print("X: %d\nY: %d");
        #else
        print("working...");
        #endif
      #endif

      #ifdef A
      y = y * (n-x);
      #else
      y = y * n;
      #endif


      #ifndef B
        #ifdef C
        print("X: %d\nY: %d");
        #else
        print("working...");
        #endif
      #endif
```

```
                x++;
            }
    }
```

8.2 clock.c

8.3 fib.c