#ifdef Confirmed Harmful: Promoting Understandable Software Variation

Duc Le
Oregon State University
ledu@eecs.oregonstate.edu

Eric Walkingshaw
Oregon State University
walkiner@eecs.oregonstate.edu

Martin Erwig
Oregon State University
erwig@eecs.oregonstate.edu

***Abstract*—Maintaining variation in software is a difficult problem that poses serious challenges for the understanding and editing of software artifacts. Although the C preprocessor (CPP) is often the default tool used to introduce variability to software, because of its simplicity and flexibility, it is infamous for its obtrusive syntax and has been blamed for reducing the comprehensibility and maintainability of software. In this paper, we address this problem by developing a prototype for managing software variation at the source code level. We evaluate the difference between our prototype and CPP with a user study, which indicates that the prototype helps users reason about variational code faster and more accurately than CPP. Our results also support the research of others, providing evidence for the effectiveness of related tools, such as CIDE and FeatureCommander.**

I. Introduction

Managing software variation is a fundamental problem in software engineering that manifests itself in many different ways throughout the field. In particular, there is a need to diversify software, for example, in terms of functionality or for different platforms and users. As with all software representations, it is important to determine the representation that best supports software developers in their work.

Among different types of representations, the C preprocessor (CPP) is often used to implement software variation because of its simplicity and flexibility. However, CPP has been criticized both for its obtrusive syntax and its lack of structure, which reduce comprehensibility and increase maintenance costs [10], [14]. Existing research has addressed these problems with various tool features to support the understanding of variational software. For example, both the *FeatureCommander* tool [5] and the *CIDE* tool [7] use background colors to highlight variational code, while CIDE also provides a "virtual separation of concerns" and imposes several restrictions on how a program can vary.

Our ultimate goal in this paper is to combine these existing approaches into a system with a highly structured, but more flexible, variational model that can address the limitations of CPP by supporting understanding and reasoning about software variation. The system should support at least the following features.

- Provide a structured and comprehensible model of variation in software.
- Remove the noisy syntax of CPP to create simpler code.
- Support virtual separation of concerns by hiding unrelated code when focusing on a particular program variant.
- Use colors to help users identify the code corresponding to particular CPP macros.

To this end, we have developed a GUI representation and implemented a simple prototype. This prototype is based on a formal representation of software variation, the choice calculus [3], that we have developed in previous work. This underlying model imposes restrictions on how and where a program can vary and informs several aspects of the prototype.

An example of the prototype's interface (modified here for readability) is shown in Figure 1. The prototype is divided into two columns. The left column is call the *dimension area*, where users can choose which program variant they want to see. Dimensions are a feature from the choice calculus and structure how a program can be configured. In terms of CPP, a dimension can be considered a way of grouping related macros together. The right column of the prototype is called the *code area*, which contains the source code of the currently selected program variant. The current selection in the dimension area is called a *configuration*. When users change a configuration on the left, the corresponding code on the right is updated. Code that is highlighted in the code area represents variation points that are associated with a particular dimension, as indicated by the color. Highlighted code can be expected to change if the selection in that dimension is changed in the dimension area. Notice in the figure that we have some purple code inside of the grey code. This is code that is included only if both of the corresponding options are selected in the dimension area.

Obviously, our prototype removes the noisy syntax of CPP and uses colors to mark variational code, but in addition to these syntactic aspects, it also provides a virtual separation of concerns [7] by only showing the code that is related to the currently selected configuration. This feature is intended

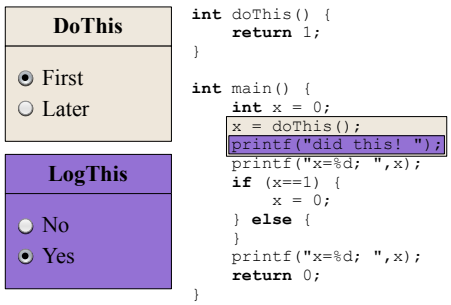

Fig. 1: The variation editor prototype.

to support the understandability of a particular variant, but it also has some associated costs: First, there is a need for the user to switch between different configurations. Second, there is a loss of *context* in which a variation point occurs; that is, we can only see the code of one variant at a time, and not how the code differs in other variants. So it is not at all obvious that the proposed representation performs better in terms of supporting understanding.

Additionally, there is evidence that the use of color alone *does not* significantly improve understanding for some kinds of tasks involving variational software [5]. Therefore, there is a need to evaluate the effectiveness of the prototype, or more specifically, the effectiveness of the combination of the above features to support users in understanding software variation. To do this we have conducted a user study with 31 computer science undergraduate students at Oregon State University. The study has shown that using the prototype can help users reason about variational code faster and more accurately than when reading syntax-highlighted CPP code. Not only does this result confirm the effectiveness of our prototype, it also supports other research, providing evidence that other tools and environments with similar characteristics, like CIDE [7], will also support the understanding of software variation.

In the rest of this paper we describe the details of the study and its results using the recommended format for reporting empirical research in [12]. We begin by clarifying the specific goals of the study in Section II. We provide information about the subject population in Section III and about experiment materials in Section IV. The tasks that study participants were asked to complete are described in Section V, and Section VI states our hypotheses and the variables involved in the study. In Sections VII and VIII we describe the experiment's design and results, respectively. We present a discussion and plans for future work in Section IX, consider threats to validity in Section X, discuss related work in Section XI, and offer conclusions in Section XII.

II. Experiment Goals

In this study, we focused primarily on comparing subjects' performance in reading and understanding code in CPP and in the prototype. Our specific goals for this experiment were:

1) Determine whether subjects can *more accurately* deduce the *number of variants* represented in code presented in our prototype compared to code annotated with CPP.
2) Determine whether subjects can *more quickly* deduce the *number of variants* represented in code presented in our prototype compared to code annotated with CPP.
3) Determine whether subjects can *more accurately* describe the *behavior* of a particular variant represented in code presented in our prototype compared to code annotated with CPP.
4) Determine whether subjects can *more quickly* determine the *behavior* of a particular variant represented in code presented in our prototype compared to code annotated with CPP.
5) Determine whether subjects consider the prototype to be *more understandable* than code annotated with CPP.

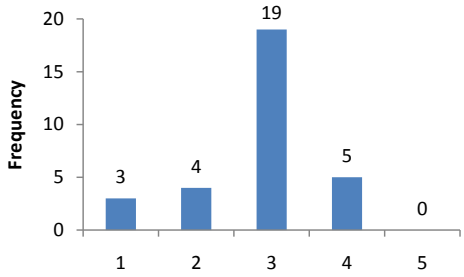

Fig. 2: Histogram of subjects' self-assessment of C/CPP programming expertise (1=beginner, 5=expert).

III. Participants

We had 31 subjects participate in the study and each subject was compensated $20. Subjects were recruited through Oregon State University's EECS mailing list. Potential subjects had to take a brief screening test before signing up for the study. This test was used to confirm a basic understanding of C and CPP, and only students that passed the screening test were asked to take part in the study. Among the 53 students who took the screening test, 45 students passed the test, and 32 of those signed up for study sessions. One subject did not show up for his study session.

The studied group consisted of 11 freshmen, 4 sophomores, 8 juniors, and 8 seniors. There were just 2 female subjects and 29 male subjects. We conducted a background questionnaire before the study that asked about the number of programming courses the students had taken and the number of courses that used C or CPP. On average, students had taken 4.9 programming courses (3.4 std. dev.) and 1.6 of those involved C or CPP (1.5 std. dev.). We also asked about their experience programming professionally, on open source projects, and whether they used C and CPP in their own personal projects. 8 of the 31 participants had professional programming experience, 7 had experience on open-source projects, and 2 had both. Of these 13 subjects, 6 used C or CPP at their job or on their open-source projects, which we qualified as "real world" C or CPP experience. 25 out of 31 subjects claimed to use C or CPP in their own personal work. Finally, subjects were asked to rate their C/CPP programming experience on a scale from 1 to 5, with 1 being a beginner and 5 being an expert. This data is summarized in Figure 2. On average, subjects rated themselves at 2.8 with a standard deviation of .81.

IV. Experiment Materials

Prior to participating in the experiment, potential subjects submitted a registration form containing a screening test through a dedicated website. This form collected the student's name, email address and answers to four questions to verify that the student had a basic understanding of C and CPP.

The experiments were administered in a lab setting, on provided computers via a web browser. The CPP annotated code was presented on a simple static web page and included syntax highlighting. The prototype tool was implemented using PHP, HTML, and Javascript. Questions were presented

to the user at the top of each page and answered via a web form that submitted the results to a secure web database.

At the beginning of the experiment session, subjects were asked to fill out a background questionnaire as described in the previous section. Next, a tutorial about how to use the CPP environment and the prototype environment was verbally administered as subjects followed along with interactive examples on the screen. At the end of the tutorial, subjects had a few minor sample tasks to perform. These were done on the same computers and in the same way as the subsequent tasks.

After finishing the main tasks, subjects were asked to fill out a post-study questionnaire to assess the perceived usefulness of the prototype compared to CPP annotations. The questionnaire included a few questions about the tasks themselves to help us interpret the other results. All questions were answered with a Likert-scale or semantic-difference scale from 1 to 5.

V. Tasks

Our experiment was performed *within-subject*, so all participants underwent both treatments (CPP and the prototype) for all tasks. All participants had to perform three tasks (1) a simple operating system selection program (the OS task), (2) a do-and-log program (the doThis task), and (3) an assembly-like language evaluator (the opcode task). These three tasks were presented to subjects in order of difficulty (OS $<$ doThis $<$ opcode). Each task was presented to each user in both CPP and the prototype, though the order of the treatments was random for each user. Also, the programs presented in each treatment for the same task were slightly different. For example, a subject who received the doThis task in the CPP representation first would receive a slightly different version of the task, with different semantics but similar overall structure, in the prototype. Both variants of each task were presented in each tool, to different subjects. All of this, combined with the randomized order of treatments, was designed to decrease the learning effect while still giving subjects tasks of equal difficulty in each treatment.

Each session was divided into two sections. The first section included the two OS and doThis tasks, and the second section included the opcode task. Subjects were allowed to rest for up to 10 minutes between sections. Otherwise, after submitting an answer to a question, subjects were immediately presented with the next. Subjects were asked to answer each question as quickly and accurately as possible.

Throughout the tasks in the CPP environment, subjects were shown the code corresponding to one task-treatment pair in a web browser window, with syntax highlighting, and the question was presented at the top of the screen. The code was displayed only while answering a question.

The prototype tasks were structured similarly, except that the prototype replaced the static code in the question-answering window. Since we were comparing the accuracy and speed of the subjects' answers in both tasks, the questions were the same, except for minor wording differences.

Below is a sequence of questions that were used for the doThis example of the CPP task.

1) How many different ways can this program be configured (by setting each macro to defined or undefined)?
2) How many program variants do you think the writer of this code *intended* to specify?
3) How many *unique* programs can be generated from this code?
4) What is the printed output of the program if the macros are defined as follows? `DoThisFirst`=undefined, `DoThisLater`=defined, `LogThis`=undefined?
5) What is the printed output of the program if the macros are defined as follows? `DoThisFirst`=defined, `DoThisLater`=defined, `LogThis`=defined?
6) What is the printed output of the program if the macros are defined as follows? `DoThisFirst`=undefined, `DoThisLater`=undefined, `LogThis`=defined?

VI. Hypotheses and Variables

The hypotheses of this study were derived from the goals described in Section II:

- H_1: Subjects predict the *number* of variants *more accurately* using our prototype than with CPP.
- H_2: Subjects predict the *number* of variants in *less time* using our prototype than with CPP.
- H_3: Subjects predict the *behavior* of a particular variant *more accurately* using our prototype than with CPP.
- H_4: Subjects predict the *behavior* of a particular variant in *less time* using our prototype than with CPP.
- H_5: Subjects consider the prototype to be *more understandable* than CPP.

These hypotheses reveal 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 hypotheses are the examples used for each treatment group and the order in which the treatment groups are presented (CPP first or prototype first). In the next section we describe how we used randomization to mitigate the effects of these uninteresting independent variables.

VII. Experiment Design

Our experiment was conducted *within subjects* to maximize statistical power with a relatively small number of subjects. All uninteresting and potentially confounding independent variables were distributed and randomized as much as possible. Specifically, we had three tasks, ordered by increasing level of difficulty. For each of the tasks, a corresponding version of the code was presented in CPP, and a slightly different version with different semantics was presented in the prototype. The order of treatments was randomized for each task to make it different from subject to subject. There are two ways to order the treatments in each task, either CPP followed by the prototype or the other way around. As a consequence, there are eight different ordering possibilities to present the tasks. We assigned subjects a random ordering and distributed subjects across the eight ordering groups evenly.

All other potentially confounding variables are mitigated by randomly assigning subjects to sessions, by the screening test, and by treating all subjects with the same introductory tutorial.

VIII. Experiment Results

Each of our hypotheses is a different view of a more fundamental claim that software variation is more understandable when represented in our prototype than when represented with CPP directives. The first four hypotheses can be addressed by analyzing the quantitative data gathered throughout our experiment, and we do this in depth in the next two subsections. Hypothesis H_5 is addressed by questions in our post-study questionnaire that asked subjects directly which tool was more understandable.

A. Counting Variants

Hypotheses H_1 and H_2 consider how accurately and quickly a subject can determine how many program variants are represented by a piece of variational software. In the experiment, we addressed these questions by having the subjects count the number of *possible* variants, the number of *unique* variants, and the number of *intended* variants represented by some code, and timing their responses. For each of the three tasks, subjects were presented an example represented in CPP and a similar example represented in the prototype (not necessarily in that order). They were asked how many possible variants, how many unique variants, and how many intended variants each representation contained. The responses to each question were scored either "correct" or "incorrect."

We assigned each subject a point for each correct response and combined the results from the three tasks. We then compared each subject's score using the CPP representation with the scores using the prototype. Four plots of this data are given in Figure 3. For each of the three types of counting questions, subjects were significantly more likely to answer correctly using the prototype than using the CPP representation.[1] If we group all of the counting questions into one category and count the number of correct answers that subjects gave using the prototype with those using the CPP representation, subjects were significantly more likely to give correct answers using the prototype than using the CPP representation (paired t-test, $t = -15.0721$, $\mathrm{df} = 30$, $p = 1.543 \times 10^{-15}$). These results provide multiple arguments in support of hypothesis H_1.

Now we consider hypothesis H_2, that subjects can count the number of variants *more quickly* using the prototype. There are two possible approaches to statistically analyzing the time data, either by basing the analysis solely on the time results, or by basing it on the relationship between the time taken and the correctness of the answers. Analyzing the data based solely on time is simpler, but it paints an inaccurate picture in some cases where subjects had no idea how to answer, so quickly gave a random answer (or no answer) and moved on to the next question. Analyzing time data based on the relationship between the correctness of the answers and the used time is a bit more complex but can address the "fast guessing" problem. Therefore, we chose this approach as our primary method for analyzing the time data (though as we show below, a straight comparison of time spent also yields significance).

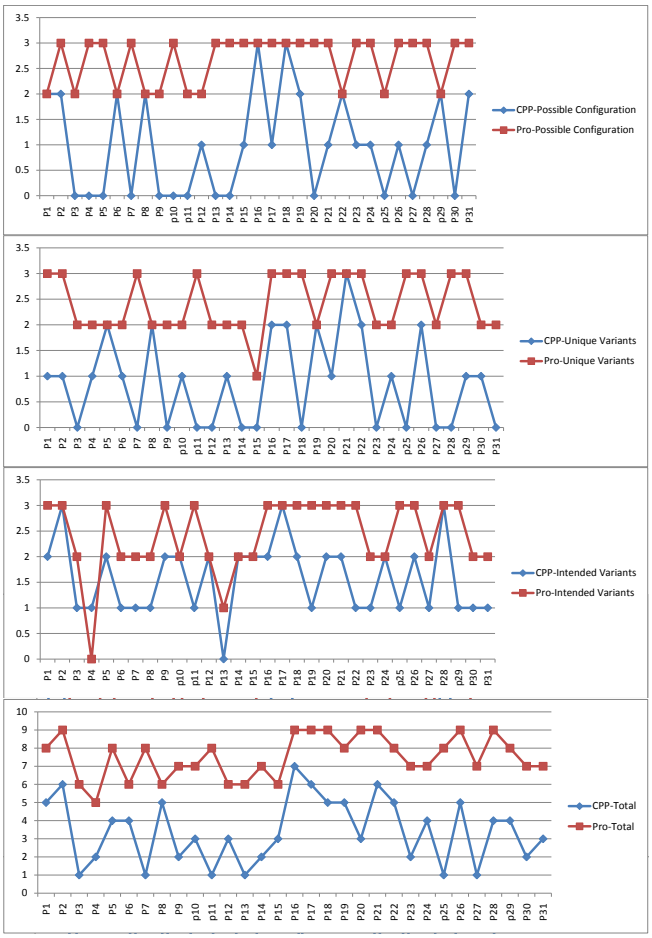

Fig. 3: Number of correctly answered counting questions for each subject, by treatment.

From the response data we computed the *efficiency* of each subject on each question, calculated as the quotient of received points over response time (in minutes). Higher efficiency scores are better, indicating a faster, more accurate response. If a subject finishes a question quickly by guessing the wrong answer, he or she will score zero for that question. The distributions of the average efficiency data are presented in Figure 4. The first two distributions represent the average efficiency by subjects for Task 1, for each environment; the next two distributions show the average efficiency for Task 2; the next two for Task 3; and the last two distributions represent overall the average efficiency for counting questions in each environment. We can confirm that the relative difficulties of the tasks were assigned correctly by observing that as the tasks get more difficult, the efficiency of the subjects decreases.

For each of the three tasks, subjects achieved significantly higher efficiency using the prototype than using the CPP

[1] Possible-configuration questions: paired t-test, $t = -8.4152$, $\mathrm{df} = 30$, $p = 2.162 \times 10^{-9}$, unique-variant questions: paired t-test, $t = -9.3185$, $\mathrm{df} = 30$, $p = 2.306 \times 10^{-10}$, intended-variant questions: paired t-test, $t = -6.356$, $\mathrm{df} = 30$, $p = 5.178 \times 10^{-7}$).

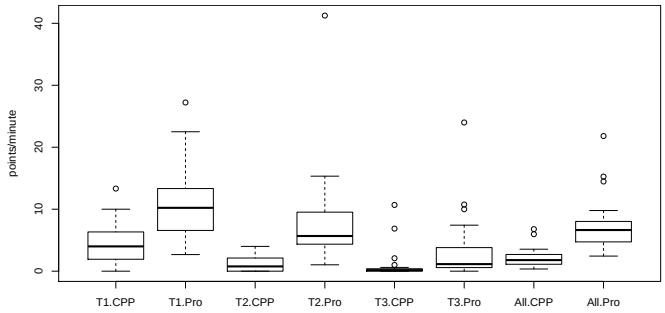

Fig. 4: Subjects' counting efficiency (points/minute).

environment for counting questions.[2] If the results of all tasks are combined, subjects also achieved significantly higher efficiency using the prototype than using the CPP environment for counting questions (paired t-test, $t = -7.1222$, df $= 30$, $p = 6.377 \times 10^{-8}$). These results support hypothesis H_2.

We tried running the same tests based solely on the time subjects used to answer counting questions and also obtained a significant difference for the all three tasks combined and for the tasks individually.[3] We can therefore note that the significant differences subjects achieved on the time scores are not dependent on the particulars of our analysis.

B. Variant Behavior

Hypotheses H_3 and H_4 consider how accurately and quickly a user can understand a particular program variant, given a piece of variational software. Within each treatment, we posed two variant comprehension questions for the first task (OS) and three variant comprehension questions for each of the remaining tasks (doThis and opcodes), for a total of eight comprehension questions per environment. These questions asked about the specific output that would be printed if a particular variant was generated and executed. As we did for the variant counting questions, we scored and timed the responses. Each answer was marked either "correct" or "incorrect." We then counted the number of correct answers for all of the comprehension questions. Four line-plots of the variant comprehension data are given in Figure 5.

Combining the results of all three tasks, statistical analysis confirms that subjects were significantly more likely to score higher on variant comprehension questions when using the prototype than when using the CPP representation (paired t-test, $t = -4.6032$, df $= 30$, $p = 7.127 \times 10^{-5}$). Likewise for the first and second tasks in isolation, subjects achieved significantly higher scores using the prototype than when using CPP.[4] Hypothesis H_4 is supported in these cases. No conclusion about the difference between using the CPP environment and the prototype can be made if we consider only the data Task 3 (paired t-test, $t = -1.5406$, df $= 30$, $p = 0.1339$).

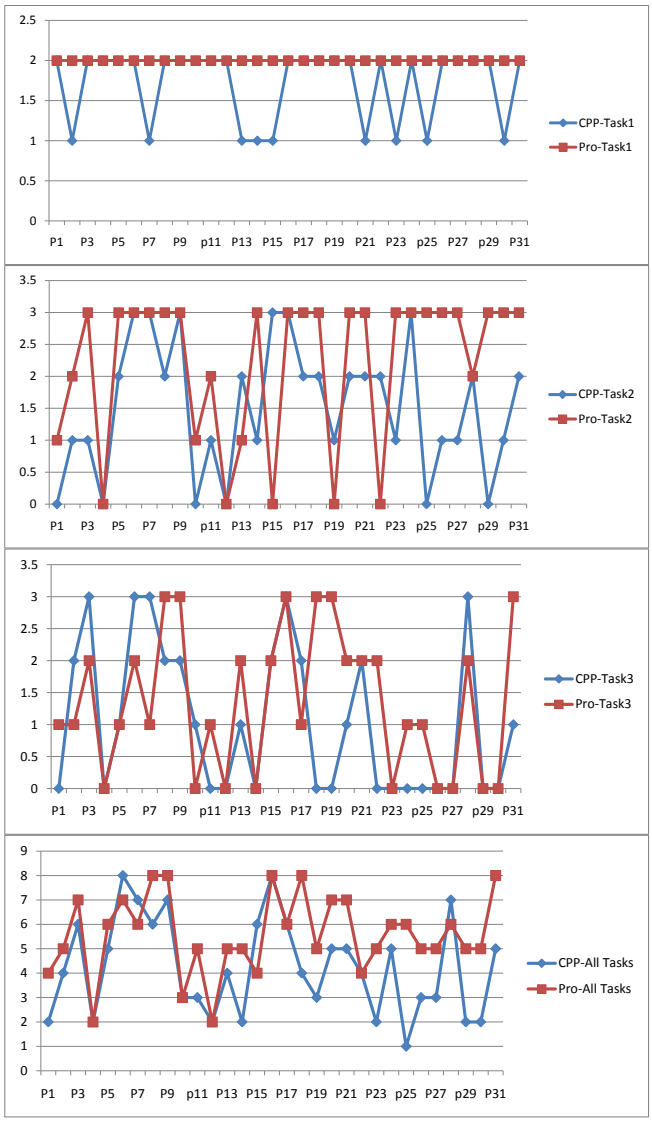

Fig. 5: Number of correctly answered variant comprehension questions for each subject, by treatment.

We suspect that the reason for non-significance of the variant comprehension questions in Task 3 is that it was simply too hard given the constraints of the study. As a result, the effects of the differences between subjects dwarfed the effects of the differences between treatments. Among the 31 subjects, 7 subjects did not get any points on the variant comprehensions questions in Task 3, and another 6 subjects got just 1 point. This means that over 40% of our subjects (13/31) received only 0 or 1 points for these tasks, making the differences in their correctness performance difficult to analyze. This theory is also supported by the data collected in the post-study questionnaire: 23 subjects agreed or strongly agreed with the statement, "the opcode (Task 3) examples were difficult to understand". In comparison, only 8 students thought Task 2 was difficult, and 3 students thought Task 1 was difficult.

[2]Task 1: paired t-test, $t = -6.6958$, df $= 30$, $p = 2.031 \times 10^{-7}$, Task 2: paired t-test, $t = -5.0559$, df $= 30$, $p = 1.990 \times 10^{-5}$, Task 3: paired t-test, $t = -2.4062$, df $= 30$, $p = 0.022$.

[3]Combined: paired t-test, $t = 6.0644$, df $= 30$, $p = 1.165 \times 10^{-6}$, Task 1: paired t-test, $t = 5.1048$, df $= 30$, $p = 1.733 \times 10^{-5}$, Task 2: paired t-test, $t = 5.2183$, df $= 30$, $p = 1.258 \times 10^{-5}$, Task 3: paired t-test, $t = 3.4906$, df $= 30$, $p = 0.0015$.

[4]Task 1: paired t-test, $t = -3.5032$, df $= 30$, $p = 0.0014$, Task 2: paired t-test, $t = -2.9901$, df $= 30$, $p = 0.0055$

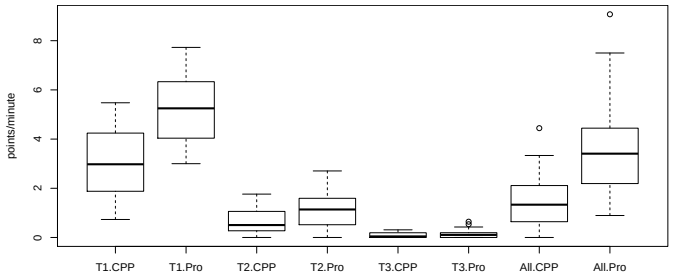

Fig. 6: Subjects' efficiency score for comprehension questions.

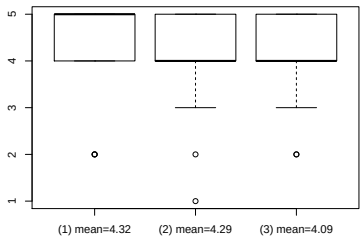

Fig. 7: Answers to questions regarding hypothesis H_5.

To confirm hypothesis H_4, that subjects understand a particular variant more quickly using the prototype, we present the efficiency scores for the comprehension questions in Figure 6. Recall that efficiency is computed by dividing the points earned for each question by the time used on that question. The first six distributions represent each of the six task-tool pairs, and the average efficiency subjects achieved on the questions in that pair. The last two distributions represent the average efficiency of subjects on all comprehension questions, for each treatment. As is obvious by looking at the distributions, subjects were significantly more likely to complete variant comprehension questions with more efficiency when using the prototype than when using the CPP representation for all of the tasks combined (paired t-test, $t = -6.6958$, $\text{df} = 30$, $p = 2.031 \times 10^{-7}$), providing support for hypothesis H_4.

In both Task 1 and Task 2, subjects were significantly more likely to complete variant comprehension questions with higher efficiency when using the prototype than when using CPP.[5] No significant difference was found for Task 3 (paired t-test, $t = -1.9054$, $\text{df} = 30$, $p = 0.066$). Again, we suspect the relative difficult of Task 3 is the reason.

C. Questionnaire Results

A post-study questionnaire consisting of 19 questions was given to subjects after they finished their main tasks. The answers to most questions are given on a Likert scale from 1 (strongly disagree) to 5 (strongly agree). There are three questions at the end of the questionnaire asking users to compare CPP and the prototype directly. These answers are captured on 5-point semantic difference scale, where 1 means "much easier in CPP" and 5 means "much easier in prototype." These three questions are used to answer hypothesis H_5 and are listed below:

(1) In which tool was it easier to count the number of configurations and program variants?
(2) In which tool was it easier to understand a particular program variant?
(3) Overall, which tool made it easier to understand software variation (code that represents many different programs)?

These questions were all answered overwhelmingly in favor of the prototype. For question (1) the average response score was 4.32 (std. dev. of 0.894). For question (2) the average response was 4.29 (std. dev. of 0.923). And for question (3) the average response was 4.10 (std. dev. of 0.893).

To triangulate and confirm the responses to the above direct comparison questions, other questions on the post-task questionnaire asked subjects to rank the difficulty of various types of tasks separately in both CPP and the prototype. We compared the responses of users on matching pairs of questions with a Wilcoxen signed-rank test. The results are given in Figure 8 and provide strong evidence for H_5.

IX. Discussion and Future Work

Here we present some other interesting observations and insights gathered during this user study and its analysis.

First, the subject's rating of the prototype seemed to coincide with their behavior during the study. P18's case is an especially strong example. During Task 3 (the hardest task), P18 began with the CPP environment. He spent 105 seconds on the first comprehension and eventually clicked next without giving any answer. For the next comprehension question in the CPP environment, he spent 12 seconds, again giving no answer and moving on. And for the third question, he spent two seconds before moving on. This sequence of events clearly indicates that P18 gave up on Task 3 for the CPP environment. One explanation for this is that the task was simply too overwhelming. When he moved on to the prototype environment for Task 3, we expected him to behave similarly by giving up on the comprehension questions, but he did not. Instead he spent 719 seconds on the first comprehension question, 246 on the second, and 184 on the third, giving correct answers to all three questions. Separation of concerns is widely believed to help program understanding by reducing the effort and attention paid to irrelevant parts of the code. In this case, however, it seemed to also increase the subject's confidence that he could even solve the problem at all. This suggests a possible line of future work.

When subjects used the prototype, we counted the number of times subjects changed the configuration before giving answers. For comprehension questions, most subjects had the same number of configuration changes. However, for the counting questions, some subjects changed the configuration many more times than others. This is due to the fact the subjects wanted to compare the differences between code variants. Informally, one subject stated the need for seeing two variants on the screen at the same time in the prototype. In addition, in the post-study questionnaire, 13 students agreed or strongly agreed that it was helpful to see multiple program

[5]Task 1: paired t-test, $t = -7.6542$, $\text{df} = 30$, $p = 1.546 \times 10^{-8}$, Task 2: paired t-test, $t = -3.0931$, $\text{df} = 30$, $p = 0.00425$.

Question Pairs	Wilcoxon signed-rank test results
In CPP annotated code/the prototype, it was easy to determine the total number of configurations.	$W = 41.5, Z = -3.7483, p = 5.595 \times 10^{-5}$ mean(CPP)=2.8, mean(Pro)=4.19
In CPP annotated code/the prototype, it was easy to determine how many unique program variants could be generated/selected.	$W = 46, Z = -3.7019, p = 9.31 \times 10^{-5}$ mean(CPP)=2.48, mean(Pro)=3.96
In CPP annotated code/the prototype , it was easy to determine how many program variants the programmer intended to specify.	$W = 25.5, Z = -3.843, p = 4.345 \times 10^{-5}$ mean(CPP)=2.58, mean(Pro)=3.93
In CPP annotated code/the prototype, it was easy to understand how a particular generated/selected program would function.	$W = 26, Z = -3.5018, p = 0.00017$ mean(CPP)=3.12, mean(Pro)=4.16
In general, the CPP annotated code/code in the prototype was easy to understand.	$W = 26.5, Z = -4.2636, p = 2.883 \times 10^{-6}$ mean(CPP)=2.58, mean(Pro)=4.19

Fig. 8: Wilcoxon signed-rank test results for questionnaire data.

variants at the same time in the CPP environment. Although the feature was not implemented in the version of the prototype used in the study, we plan to include tooltips in future versions of the prototype that show how highlighted code in the code section is different in other configurations. We believe that this feature could be especially useful in helping users determine the number of unique variants a piece of software represents. Task 3 contained 8 total configurations but only 6 unique variants. This proved difficult for users to determine (only 17 out of 31 got the question correct), since it required clicking through all 8 variants and remembering which had been seen. While users still performed better on this task than when using CPP, we believe that further gains could be made by providing additional variational context via tooltips.

At the end of their study sessions, many subjects expressed their enthusiasm about the prototype. One subject said, "This is really cool! I love this tool." Another subject even wanted to use the tool for his work, "If you guys (the researchers) develop this tool, I would love to try it for my work. This would definitely make a difference." This feedback from subjects suggests the possibility of a field study. There are several issues that must be addressed before the tool is ready for real-world tasks, however. These are addressed in the discussion of external validity in the next section.

X. Threats to Validity

Here we describe some threats to the validity of our study. We focus on threats not discussed elsewhere in the paper. Many other internal threats to validity are addressed with randomization, as discussed in Sections V and VII.

A. Construct Validity

In the pre-study questionnaire we asked subjects to judge their C/CPP programming experience. This is subjective and may not accurately represent a subject's ability to read and understand CPP-annotated programs. For example, participant P29 gave himself 4 points on a 5-point Likert scale for his C/CPP experience but did not get any answers correct for the third task. At the same time, participant P16 gave himself 3 points and got correct answers for all the questions in the third task. This information was used for demographic purposes only and our study results do not rely on these data.

B. Internal Validity

For each task in the study, we created a version to be presented in the CPP environment and a slightly different version to be presented in the prototype. Implementing the tasks this way ensured that the code shown in each environment is at the same difficulty level. However, this also introduce a significant learning effect. That is, if subjects begin with the CPP environment for a task, they will very likely do better with the prototype for the same task and vice versa. We controlled for this learning effect by randomizing the order of treatments for each task, within each subject.

C. External Validity

All of the subjects recruited in this study were students, so the results of the study might not generalize to all software developers. Moreover, the tasks in the study were designed to be appropriate for an undergraduate level of programming expertise and to fit within the constraints of the study's format, and thus do not represent typical C/CPP programs in the real world. These are typical limitations of controlled studies. Had we given students real-world C programs, it would have taken much longer for them to finish the tasks, with a higher likelihood of differing levels of experience introducing noise into the study's outcome.

The prototype itself is also quite limited since code can only be read and not edited. Our focus in this study is on promoting understanding, but in order to support the larger problem of maintaining variational software, it would also need to provide ways to edit and refactor code containing variation. How to best support these operations remains an important open research question.

Finally, the tasks concerned with counting configurations probably do not correspond directly to tasks performed by programmers in the real world. These tasks are intended as a simple way to assess the user's understanding of the overall variational structure of the code. While this information might sometimes be more readily attainable in CPP-based projects by looking at external documentation or a feature model [6], this information is very often not available, out-dated, or inconsistent with the implementation. Therefore, it is important for programmers to be able to easily determine the variational structure of software from the code and environment alone.

XI. Related Work

Several authors have argued that the use of CPP can lead to highly complex code, prevent comprehension, and increase the cost of maintenance. Spencer [14] states that the "C programmer's impulse to use #ifdef in an attempt at portability is usually a mistake." Lohmann [11] uses the term "ifdef-hell" to describe the implementation of cross-cutting concerns [9] in the source code of different contemporary operating systems. Favre identifies many specific difficulties when trying to understand CPP, including lack of abstraction, lack of available techniques, and unclear tool semantics [4]. All mentioned papers point out the need for a tool to support understanding CPP, which is one goal of this paper.

Several tools have been proposed to manage the complexity of CPP, for example, the VSF editor [2] and C-CLR [13]. One of the most popular and actively researched tools is CIDE [8], a graphical integrated development environment for creating and managing features in feature-oriented software development [1]. The model used by CIDE is basically equivalent to CPP with only #ifdef statements, with many changes made in the interest of usability. CIDE annotates optional code using background colors, similar to our prototype. Unlike our prototype, code can only be marked as *optional* in CIDE—there is no equivalent to the notion of a "dimension" in which alternatives can be selected. Our approach is more expressive, as shown in [3], and provides additional structure to the space of potential variants. While CPP allows arbitrary lines of code to be marked as optional, CIDE limits variation points to nodes in the AST. This leads to more structured and regular variation than in CPP, and ensures that, at the very least, all program variants will be syntactically correct. This feature is also present in our prototype, inherited from our underlying choice calculus model [3]. Finally, like our prototype, CIDE provides a "virtual separation of concerns" (a term coined by the CIDE researchers) by allowing users to select which features to show and which to omit [7].

Another related tool is FeatureCommander (FC) [5]. FC relies on CPP directly as its underlying model, simply adding background color to existing CPP annotated documents. The primary focus of FC is on scaling the idea of background color to code bases with hundreds of CPP macros. FC copes with this problem by retaining the underlying CPP annotations and allowing users to mark only those macros which they are most interested in, and only those will be marked with background colors. FC was recently evaluated in a user study with 14 graduate students at University of Magdeburg, Germany. The examples used in this study were closer to real-world examples than those used in our study, containing over 160,000 lines of code and 340 features. The study tasks revolved around finding the code associated with a smaller set of 12 features and solving simple maintenance tasks. While they did find a significant difference in the response times to these questions, they did not find a significant difference in correctness. We believe this provides evidence that background color alone is not enough, and that separation of concerns is also important for program understanding.

XII. Conclusion

We have demonstrated that a representation of software variation based on the principles of (1) an unobtrusive syntax based on background coloring, (2) virtual separation of concerns, and (3) a dimension-based model of variation can improve understandability over software variation implemented with the C Preprocessor.

In future work we will investigate the impact of the loss of variational context that results from virtual separation of concerns, whether this poses problems for specific kinds of tasks, and whether tooltips that show alternative code simultaneously can mitigate any such issues. We will also consider the extension of the prototype to support the editing and refactoring of variational software, and whether these extensions are better than manipulating CPP-annotated code directly.

Acknowledgement

This work is supported by the Air Force Office of Scientific Research under the grant FA9550-09-1-0229 and by the National Science Foundation under the grant CCF-0917092. We would also like to thank Margaret Burnett, whose thoughtful advice was critical to the success of this study.

References

[1] S. Apel and C. Kästner. An Overview of Feature-Oriented Software Development. *Journal of Object Technology*, 8(5):49–84, 2009.

[2] M. C. Chu-Carroll, J. Wright, and A. T. T. Ying. Visual Separation of Concerns through Multidimensional Program Storage. In *Int. Conf. on Aspect-Oriented Software Development*, pages 188–197, 2003.

[3] M. Erwig and E. Walkingshaw. The Choice Calculus: A Representation for Software Variation. *ACM Trans. on Software Engineering and Methodology*, 2011. To appear.

[4] J. M. Favre. Understanding-in-the-Large. In *Work. on Program Comprehension*, pages 29–38, 1997.

[5] J. Feigenspan, M. Schulze, M. Papendieck, C. Kästner, R. Dachselt, V. Köppen, and M. Frisch. Using Background Colors to Support Program Comprehension in Software Product Lines. In *Int. Conf. on Evaluation and Assessment in Software Engineering*, pages 66–75, 2011.

[6] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Nov. 1990.

[7] C. Kästner and S. Apel. Virtual Separation of Concerns—A Second Chance for Preprocessors. *Journal of Object Technology*, 8(6):59–78, 2009.

[8] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *IEEE Int. Conf. on Software Engineering*, pages 311–320, 2008.

[9] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *European Conf. on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer-Verlang, 1997.

[10] M. Krone and G. Snelting. On the Inference of Configuration Structures from Source Code. In *Int. Conf. on Software Engineering*, pages 49–57, 1994.

[11] D. Lohmann, F. Scheler, R. Tartler, O. Spinczyk, and W. Schröder-Preikschat. A Quantitative Analysis of Aspects in the eCos Kernel. *ACM SIGOPS Operating Systems Review*, 40(4):191–204, 2006.

[12] F. Shull, J. Singer, and D. I. K. Sjöberg. *Guide to Advanced Empirical Software Engineering*. Springer-Verlag, New York, 2007.

[13] N. Singh, C. Gibbs, and Y. Coady. C-CLR: A Tool for Navigating Highly Configurable System Software. In *Work. on Aspects, Components, and Patterns for Infrastructure Software*, 2007.

[14] H. Spencer and G. Collyer. #ifdef Considered Harmful, or Portability Experience With C News. In *Proc. of the USENIX Summer Conf.*, pages 185–198, 1992.