AN ABSTRACT OF THE PROJECT REPORT OF

Duc Le for the degree of Master of Science in Computer Science presented on January 27, 2014.

Title: #ifdef Confirmed Harmful - Promoting Understandable Software Variation

Abstract approved: ____________________________________________________________

Martin Erwig

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 project, we address this problem by developing a prototype for managing software variation at the source code level. We evaluated the difference between our prototype and CPP with a user study, which indicated that the prototype helped 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.
#ifdef Confirmed Harmful - Promoting Understandable Software Variation

by

Duc Le

A PROJECT REPORT

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented January 27, 2014
Commencement June 2014
Master of Science project report of Duc Le presented on January 27, 2014.

APPROVED:

__________________________
Major Professor, representing Computer Science

__________________________
Director of the School of Electrical Engineering and Computer Science

__________________________
Dean of the Graduate School

I understand that my project report will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my project report to any reader upon request.

__________________________
Duc Le, Author
ACKNOWLEDGEMENTS

First and foremost, I would like to thank my wife for her support from across the Pacific Ocean. Thank you for always being there when I need courage and encouragement.

I would like to express my deepest appreciation to my advisor, Professor Martin Erwig. Your guidance helped me grow over the years, and your insightful feedback made me a better and more critical thinker.

I would like to thank Professor Margaret Burnett for her thoughtful advice during the course of this project, which was critical to the success of our user study. I would also like to thank Professor Chris Scaffidi for his feedback during VL/HCC 2011’s graduate consortium and Professor Alex Groce for his support during my graduate study at Oregon State University.

A very special thank to my fellow graduate students, Sheng Chen, Chris Chambers, Eric Walkingshaw, and others. Your friendship made Corvallis less of a rainy town.
CONTRIBUTION OF AUTHORS

This project report draws materials from the paper “#ifdef Confirmed Harmful - Promoting Understandable Software Variation [19],” of which I am the corresponding author. My co-authors are Eric Walkingshaw and Martin Erwig.

The following new materials have been added to the content of [19]:

- The user study’s screening test (page 6).
- Subjects’ study results based solely on timing data (page 13 and page 17).
- Pearson’s correlation between subjects’ ratings of the prototype and their comprehension scores (page 19).
- A more thorough discussion of related work (page 23).
- User study materials, which include pre-study tutorials (page 37) and pre/post-study questionnaires (page 47).
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th></th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
</tr>
<tr>
<td>2</td>
<td>Experiment Design</td>
</tr>
<tr>
<td>2.1</td>
<td>Experiment Goals</td>
</tr>
<tr>
<td>2.2</td>
<td>Participants</td>
</tr>
<tr>
<td>2.3</td>
<td>Experiment Materials</td>
</tr>
<tr>
<td>2.4</td>
<td>Tasks</td>
</tr>
<tr>
<td>2.5</td>
<td>Hypotheses and Variables</td>
</tr>
<tr>
<td>2.6</td>
<td>Experiment Design</td>
</tr>
<tr>
<td>3</td>
<td>Experiment Results</td>
</tr>
<tr>
<td>3.1</td>
<td>Counting Variants</td>
</tr>
<tr>
<td>3.2</td>
<td>Variant Behavior</td>
</tr>
<tr>
<td>3.3</td>
<td>Questionnaire Results</td>
</tr>
<tr>
<td>4</td>
<td>Discussion and Future Work</td>
</tr>
<tr>
<td>5</td>
<td>Threats to Validity</td>
</tr>
<tr>
<td>5.1</td>
<td>Construct Validity</td>
</tr>
<tr>
<td>5.2</td>
<td>Internal Validity</td>
</tr>
<tr>
<td>5.3</td>
<td>External Validity</td>
</tr>
<tr>
<td>6</td>
<td>Related Work</td>
</tr>
<tr>
<td>6.1</td>
<td>Configuration-Supporting Tools</td>
</tr>
<tr>
<td>6.1.1</td>
<td>K-Config</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Feature Models</td>
</tr>
<tr>
<td>6.1.3</td>
<td>TypeChef</td>
</tr>
<tr>
<td>6.2</td>
<td>Representation-Supporting Tools</td>
</tr>
<tr>
<td>6.2.1</td>
<td>C-CLR</td>
</tr>
<tr>
<td>6.2.2</td>
<td>CIDE</td>
</tr>
<tr>
<td>6.2.3</td>
<td>FeatureCommander</td>
</tr>
<tr>
<td>7</td>
<td>Conclusion</td>
</tr>
</tbody>
</table>
## TABLE OF CONTENTS (Continued)

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bibliography</td>
<td>32</td>
</tr>
<tr>
<td>Appendices</td>
<td>36</td>
</tr>
<tr>
<td>A Study Materials</td>
<td>37</td>
</tr>
</tbody>
</table>
**LIST OF FIGURES**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>The variation editor prototype.</td>
<td>2</td>
</tr>
<tr>
<td>2.1</td>
<td>Histogram of subjects’ self-assessment of C/CPP programming expertise (1=beginner, 5=expert).</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>C and CPP knowledge screening test.</td>
<td>6</td>
</tr>
<tr>
<td>3.1</td>
<td>Number of correctly answered counting questions for each subject, by treatment.</td>
<td>11</td>
</tr>
<tr>
<td>3.2</td>
<td>Subjects’ counting efficiency (points/minute).</td>
<td>13</td>
</tr>
<tr>
<td>3.3</td>
<td>Subjects’ timing results for counting questions.</td>
<td>13</td>
</tr>
<tr>
<td>3.4</td>
<td>Number of correctly answered variant comprehension questions for each subject, by treatment.</td>
<td>15</td>
</tr>
<tr>
<td>3.5</td>
<td>Subjects’ efficiency score for comprehension questions.</td>
<td>16</td>
</tr>
<tr>
<td>3.6</td>
<td>Subjects’ timing results for comprehension questions.</td>
<td>17</td>
</tr>
<tr>
<td>3.7</td>
<td>Answers to questions regarding hypothesis $H_5$.</td>
<td>18</td>
</tr>
<tr>
<td>3.8</td>
<td>Wilcoxon signed-rank test results for questionnaire data.</td>
<td>18</td>
</tr>
<tr>
<td>6.1</td>
<td>An Excerpt of KConfig Configurations [21]</td>
<td>24</td>
</tr>
<tr>
<td>6.2</td>
<td>An Example of a Feature Model [5]</td>
<td>25</td>
</tr>
<tr>
<td>6.3</td>
<td>Variational Reference Analysis Example [15]</td>
<td>26</td>
</tr>
<tr>
<td>6.4</td>
<td>Macro Selection in C-CLR [24]</td>
<td>28</td>
</tr>
<tr>
<td>6.5</td>
<td>Variational Code Mining in C-CLR [24]</td>
<td>28</td>
</tr>
<tr>
<td>6.6</td>
<td>CIDE - Virtual Separation of Concerns</td>
<td>29</td>
</tr>
<tr>
<td>6.7</td>
<td>Feature Commander - Using Background Colors to Support Program Comprehension in SPL</td>
<td>30</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>----------</td>
<td>--------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>A.1</td>
<td>Clock Example</td>
<td>38</td>
</tr>
<tr>
<td>A.2</td>
<td>Adding 12-hour mode</td>
<td>40</td>
</tr>
<tr>
<td>A.3</td>
<td>Introduction to Prototype</td>
<td>43</td>
</tr>
<tr>
<td>A.4</td>
<td>Adding a Second Dimension</td>
<td>44</td>
</tr>
</tbody>
</table>
Chapter 1: 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 [17, 25]. Existing research has addressed these problems with various tool features to support the understanding of variational software. For example, both the FeatureCommander tool [8] and the CIDE tool [10] use background colors to highlight variational code, while CIDE also provides virtual separation of concerns, the idea that irrelevant and distracting code should be hidden from developers. Another feature that CIDE supports is the restriction on how a program can vary. In CIDE, code can only be marked as optional, and variation can only happen on abstract syntax trees’ nodes.

Our ultimate goal in this project 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 [6], that Erwig and Walkingshaw 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 1.1. The prototype’s interface is divided into two columns. The left column is called 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. A program variant is a program that contains no variation and is obtained by resolving the variational structures of the code base. The interface’s right column 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 have been resolved to a single version and 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 [10] by only showing the code that is related to the currently selected configuration. This feature

```c
int doThis() {
    return 1;
}
int main() {
    int x = 0;
    x = doThis();
    printf("did this! ");
    if (x==1) {
        x = 0;
    } else {
    }
    printf("x=%d; ",x);
    return 0;
}
```

![Figure 1.1: The variation editor prototype.](image-url)
is intended 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 to find code that is relevant to their current tasks. 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 [8]. 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 [10], will also support the understanding of software variation.

In the rest of this report we describe the details of the study and its results using the recommended format for reporting empirical research in [23]. In Chapter 2, we begin by clarifying the specific goals of the study in Section 2.1. We provide information about the subject population in Section 2.2 and about experiment materials in Section 2.3. The tasks that study participants were asked to complete are described in Section 2.4, and Section 2.5 states our hypotheses and the variables involved in the study. In Chapter 3 we describe the experiment’s results. We present a discussion and plans for future work in Chapter 4, consider threats to validity in Chapter 5, discuss related work in Chapter 6, and offer conclusions in Chapter 7.
Chapter 2: Experiment Design

2.1 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.

2.2 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. The data are summarized in Figure 2.1. On average, subjects rated themselves at 2.8 with a standard deviation of .81.

2.3 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 accompanying screening test was designed to confirm the student’s responses to these
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:

```c
int main() {
  #ifdef A
    int x = 0;
  #else
    int x = 1;
  #endif
  if (x) {
    #ifdef B
      return 3;
    #else
      return 4;
    #endif
  }
  return 5;
}
```

Figure 2.2: C and CPP knowledge screening test.

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.

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.
2.4 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). The source code of these tasks can be found at [26]. These three tasks were presented to subjects in increasing 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?
   \[ \text{DoThisFirst=undefined, DoThisLater=defined, LogThis=undefined} \]

5. What is the printed output of the program if the macros are defined as follows?
   \[ \text{DoThisFirst=defined, DoThisLater=defined, LogThis=defined} \]

6. What is the printed output of the program if the macros are defined as follows?
   \[ \text{DoThisFirst=undefined, DoThisLater=undefined, LogThis=defined} \]

2.5 Hypotheses and Variables

The hypotheses of this study were derived from the goals described in Section 2.1:

\[ H_1: \] 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.
2.6 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.
Chapter 3: 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.

3.1 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 these data are given in Figure 3.1. 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, $t = -15.0721$, $df = 30$, $p = 1.543 \times 10^{-15}$). These results provide multiple arguments in support of hypothesis $H_1$.

Figure 3.1: Number of correctly answered counting questions for each subject, by treatment.

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).

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 3.2. 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 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\(^3\) (box plots in Figure 3.3). We can therefore note that the significant differences subjects achieved on the time scores are not dependent on the particulars of our analysis.

\(^2\)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, \(t = 3.4906, df = 30, p = 0.0015\).
Figure 3.2: Subjects' counting efficiency (points/minute).

Figure 3.3: Subjects’ timing results for counting questions.
3.2 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 opcode), 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 3.4.

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. 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, $p = 0.1339$).

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 comprehension 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.

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 3.5. Recall

\[\text{Task 1: paired } t\text{-test, } t = -3.5032, \text{ df } = 30, \ p = 0.0014, \text{ Task 2: paired } t\text{-test, } t = -2.9901, \text{ df } = 30, \ p = 0.0055\]
Figure 3.4: Number of correctly answered variant comprehension questions for each subject, by treatment.
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, $t = -6.6958$, 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.\footnote{Task 1: paired $t$-test, $t = -7.6542$, df = 30, $p = 1.546 \times 10^{-8}$, Task 2: paired $t$-test, $t = -3.0931$, df = 30, $p = 0.00425$.} No significant difference was found for Task 3 (paired $t$-test, $t = -1.9054$, df = 30, $p = 0.066$). We obtained similar results by running statistical tests based solely on timing data\footnote{Task 1: paired $t$-test, $t = 3.8182$, df = 30, $p = 0.000628$, Task 2: paired $t$-test, $t = 3.2407$, df = 30, $p = 0.00292$, Task 3: paired $t$-test, $t = 0.9047$, df = 30, $p = 0.3729$.} (box plots in Figure 3.6). Again, we suspect the relative difficult of Task 3 is the reason.
3.3 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 $H_3$ 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) (Figure 3.7).
Figure 3.7: Answers to questions regarding hypothesis $H_5$.

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

Figure 3.8: Wilcoxon signed-rank test results for questionnaire data.

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 Wilcoxon signed-rank test. The results are given in Figure 3.8 and provide strong evidence for $H_5$. 
Chapter 4: Discussion and Future Work

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

First, subjects’ ratings of the prototype seemed to coincide with their behavior during the study. We found that subjects’ ratings of the prototype’s comprehensibility and their comprehension scores were moderately correlated (Pearson’s $r(29) = 0.398, p = 0.026$). In addition, 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 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.
Chapter 5: Threats to Validity

Here we describe some threats to the validity of our study. We focus on threats not discussed elsewhere in the report. Many other internal threats to validity are addressed with randomization, as discussed in Chapter 2.

5.1 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.

5.2 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 introduces 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.
5.3 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 [9], this information is very often not available, outdated, 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.
Chapter 6: 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 [25] states that the “C programmer’s impulse to use #ifdef in an attempt at portability is usually a mistake.” Lohmann [20] uses the term “ifdef-hell” to describe the implementation of cross-cutting concerns [16] 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 [7]. All mentioned papers point out the need for a tool to support understanding CPP, which is one goal of our project.

Existing tool support for CPP understanding can be classified into two categories: configuration support and representation support. Configuration-supporting tools provide solutions to inspect, configure, and reason about CPP macros. Representation-supporting tools, on the other hand, aid the representation of #ifdef code to developers by using visual notations (often colors) or by adding an abstraction layer on top of CPP macros. Our prototype has both configuration and representation support—the dimension area manages and restricts variational configurations while the code area hides underlying CPP directives. In the following sections, we discuss several configuration and representation-supporting tools and compare them with our approach.

6.1 Configuration-Supporting Tools

6.1.1 K-Config

K-Config [21] is a language for modeling the Linux Kernel’s variability. The language provides a set of constructs for specifying CPP configuration options, or simply configs, and dependencies among them. There exist several types of configs. Boolean configs can be turned on and off, tristate configs have two “on” states to indicate whether corresponding code should be linked statistically or dynamically, and value configs define numerical and strings values.
menu "Power management and ACPI options"
depends on !X86_VOYAGER
config PM
  bool "Power Management support"
  depends on !IA64_HP_SIM
  ---help---
  "Power Management" means that . . .
config PM_DEBUG
  bool "Power Management Debug Support"
  depends on PM
config CPU_IDLE
  bool "CPU idle PM support"
  default ACPI
config PM_SLEEP
  bool
  depends on SUSPEND || HIBERNATION ||
      XEN_SAVE_RESTORE
  default y
...
endmenu

Figure 6.1: An Excerpt of KConfig Configurations [21]

Figure 6.1 shows a KConfig menu declaration that groups related configs into a manageable module. Menus have the same goal as our prototype’s notion of dimensions, yet included configs do not have to be alternative. To support alternative relationships, KConfig provides a choice construct, which is similar to the notion of choices in the Choice Calculus.

One can indicate dependencies among configs by using the depends on construct. A select config (not shown in Figure 6.1) represents the reverse relationship of depends on—whenever a config is selected, all of its select configs will also have to be selected. These features are more expressive than what our prototype provides, and more work is needed to decide what and how we can integrate into our existing system.

6.1.2 Feature Models

KConfig and its relative languages belong to a family of languages called feature models [9]. In its most general definition, a feature model is a program representing all software products
in a software product line (SPL) [18, 22]. Different languages can be used to represent features models such as propositional formulas, tree grammars, and feature diagrams [2].

Figure 6.2 shows a feature diagram from [5]. Each node in the diagram is a feature, which is a user-visible functionality. Several feature relationships are presented in Figure 6.2. The parent-child relationship requires the parent feature to be selected in software products where at least one of its children is selected. The child-mandatory relationship (solid circle such as on top of body) requires the child to be selected when its parent is, while the child-optional relationship (hollow circle such as on top of power lock) does not. An OR relationship (solid cone net such as underneath engine) requires at least one child to be selected when the parent is. An XOR relationship (hollow cone net such as underneath gear) indicates that only one child can be selected.

Feature models are used to configure software product lines, which are often implemented using CPP. This approach is considered to be advantageous as the problem space (represented by feature models) is separated from the solution space (implemented by CPP) [3]. There is, however, no one-to-one mapping between features and CPP macros in industrial SPL projects, leaving an unfortunate gap between the problem and solution spaces. The design of the Choice Calculus and our prototype spans across these domains and bridges this gap. The dimension panel provides a restricted representation of feature models, the code panel disciplines the use of variational #ifdefs, and each configuration in the dimension area is linked directly to specific implementation in the code area.
```c
#include <stdio.h>

#ifdef WORLD
char *msg = "Hello World\n";
#endif
#ifdef BYE
char *msg = "Bye bye!\n";
#endif

main() {
    printf(msg);
}
```

Figure 6.3: Variational Reference Analysis Example [15]

### 6.1.3 TypeChef

TypeChef [13, 15] addresses the problem of type checking #ifdef annotated C code. Given \( n \) different macros that can be defined/undefined, there are up to \( 2^n \) different program variants to be checked. TypeChef avoids the expensiveness of this problem by tackling the smaller problem of variational reference analysis. Figure 6.3 provides an example from [15].

In this example, there are four different program variants (as \texttt{WORLD} and \texttt{BYE} can be defined or undefined), yet only two variants are type correct. When neither of \texttt{WORLD} and \texttt{BYE} is defined, \texttt{msg} is not defined. When both macros are defined, we have a variable redefinition error.

Given no domain knowledge, the implementation in Figure 6.3 is always ill-typed as a whole. However, if this implementation is supplemented with the feature model \( \neg (\texttt{WORLD} \land \texttt{BYE}) \land (\texttt{WORLD} \lor \texttt{BYE}) \), the entire software product line will be type correct. Thus, TypeChef’s input requires a feature model and its corresponding implementation.

One major limitation of TypeChef is that it is not a type checker but rather a variational-aware reference analyzer. A constraint-satisfaction-problem solver is used underlyingly to prevent undefined variables and variable redefinition. In a more recent work [14], TypeChef developers provide a type system for variability-aware modules \( \mathcal{M}^{vl} \) as part of the TypeChef project. \( \mathcal{M}^{vl} \) considers each .c file to be a module and modularly checks all of its CPP configurations. After that, \( \mathcal{M}^{vl} \) composes module configurations to perform global type checking. This work is said to be built on “prior work on variability-aware analysis,” which includes Chen et al.’s work.
on variational type checking the Choice Calculus [4], specifically the notion of *choice type*.

### 6.2 Representation-Supporting Tools

#### 6.2.1 C-CLR

C-CLR [24] enables understanding of CPP-annotated code by listing CPP macros linearly and letting developers configure macro values. Developers can only view the variational code relevant to the current configuration. Figure 6.4 shows the macro selection window of C-CLR. If a CPP macro is selected, it is considered defined and corresponding code will be shown. This is similar to our prototype with the exception that macros and code are not highlighted with corresponding colors.

C-CLR also supports mining aspects in CPP-annotated code. Developers create patterns to be searched, and C-CLR will output all *Levenshtein-close* variational code blocks. As an example, figure 6.5 displays the results of matching the pattern

*name = "RID"; *value = &info->sigContent->rdi;

This pattern-matching feature works with source code's textual representation. Our prototype can represent source code at both textual and variational abstract syntax tree levels, which provides opportunities for using tree-matching algorithms to improve search results.

#### 6.2.2 CIDE

One of the most popular and actively researched tools is CIDE [12], a graphical integrated development environment for creating and managing features in feature-oriented software development [1]. Figure 6.6 shows CIDE’s interface[^2]. *Feature code* is variational code associating to certain features by matching colors. *Overlapping feature code* varies with regard to more than one features and has the features’ blended colors. Variational code that is currently unselected is


Figure 6.4: Macro Selection in C-CLR [24]

Figure 6.5: Variational Code Mining in C-CLR [24]
Figure 6.6: CIDE - Virtual Separation of Concerns

marked as hidden code.

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 [6], 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 [6]. 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 [10].

CIDE was empirically evaluated in [11]. The experiment’s main goal was deciding whether colors can improve program comprehension over textual preprocessors. 43 students at University of Passau were selected in a between-subject experiment with 21 working on textual preprocessors and 22 working on CIDE. Subjects had to perform two search tasks and four maintenance tasks, which involved fixing seeded bugs in the code. The study result indicates that CIDE subjects
were significantly faster in search tasks than textual subjects. However, there was no significant improvement in using colors for maintenance tasks.

6.2.3 FeatureCommander

Another related tool is FeatureCommander (FC) [8]. 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. Figure 6.7 shows FC’s interface.

FC was evaluated in a user study with 14 graduate students at University of Magdeburg. 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 FC’s developers did find a significant difference in the response times to these questions, they did not find a significant difference in correctness. We believe this, along with the results from the study on CIDE, provides evidence that background color alone is not enough, and that separation of concerns is also important for program understanding.
Chapter 7: 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.
Bibliography


APPENDICES
Appendix A: Study Materials

A.1 Tutorials

Managing Software Variation Tutorial Duc Le (Driver), Eric Walkingshaw (Tutorializer)

Introduction

Alright, let’s get started. My name is Eric, and that’s Duc over there, and we’ll be helping you throughout this study. If your cell phone is on, please set it to silent mode now. [Wait]

Ok, before we begin the study, we are going to spend 15 minutes or so introducing you to the tools that you’re going to be using and the types of questions that you will be asked. We will make it very clear when the actual study starts, so don’t worry about messing anything up yet. This is all just for practice.

I’ll be reading from this script to be consistent in the information I provide you and the other people taking part in this study. Please don’t discuss this study with anyone, since we don’t want other participants to receive any advance information.

Let’s get started. Let me know if you have any questions as we go along. [Wait]

Click on the “Start Tutorial” shortcut on your desktop now, and then click on the “Start Tutorial” link in the browser. The study will only work in Firefox, so if some other browser loaded or if your screen doesn’t look like the one on the projector, let us know now. [Wait]

It’s important that we all stay together, so please don’t click the “Next” link at the bottom until I’ve instructed you to do so. Duc will be pointing out the things that we’re talking about on the projector as we go along.
Ok, we’re going to begin by looking at this simple C program that implements a clock. At the top, we declare a new struct type called `time`.

```c
typedef struct {
    int hours;
    int minutes;
} time;

time current = {0, 0};

void printTime() {
    printf("%d:%d", current.hours, current.minutes);
}

// called once every minute
void tick() {
    if (++current.minutes == 60) {
        current.minutes = 0;
        if (++current.hours == 24) {
            current.hours = 0;
        }
    }
}
```

Next

Figure A.1: Clock Example

Clock Example

Ok, we’re going to begin by looking at this simple C program that implements a clock. At the top, we declare a new struct type called `time`.

[Driver circles “time” struct.]

Then we create a global variable to represent the current time, initialized to hour zero and minute zero, or midnight.

[Driver circles “current” variable.]  

Then we have a function `printTime` that is used to print the current time.

[Driver circles “printTime()”]

This function uses the `printf` statement, which will also be used in the study. The first argument to `printf` is a format string, and the following arguments are inserted in place of each of the “%d”
substrings in the format string. Are there any questions about this function? [Wait.]

Ok, finally, we have a function called “tick”, which we assume is called by some external code once every minute.

[Driver circles “tick.”]

When tick is called, we increment the current minutes value.

[Driver point to “++current.minutes”]

If it now equals 60, an hour has passed, so we reset the minutes value and increment the current hour.

[Driver point to “++current.hours”]

And if that now equals 24, a whole day has passed so the current hours are reset to 0 and we’re back at midnight.

This is a lot of code quickly. Does anyone have any questions? [Wait.]

Ok, let’s move on. Please click the “Next” link now. [Wait.]

**Intro to CPP: Adding 12-hour mode**

Ok, before our clock stored time in a 24-hour format, but now we’re going to extend it using C Preprocessor ifdef statements so that we can choose between a 24-hour format or a 12 hour format.

We’re using two C Preprocessor macros here. The first, _24H, is used to mark code that corresponds to 24 hour clocks.

[Driver point to “_24H.”]

And the second, _12H, is used to mark code that corresponds to 12 hour clocks.

[Driver point to “_12H.”]
typedef struct {
    int hours;
    int minutes;
#ifdef _12H
    int PM; // set to 0 if AM, 1 if PM
#endif
} time;
#endif

time current = {0, 0};
#endif
time current = {0, 0, 0};
#endif

void printTime() {
    printf("%d:%d", current.hours, current.minutes);
#ifdef _12H
    if (current.PM) printf(" PM");
    else printf(" AM");
#endif
}

// called once every minute
void tick() {
    if (++current.minutes == 60) {
        current.minutes = 0;
#ifdef _12H
    if (++current.hours == 24) {
        #elif defined _12H
            if (++current.hours == 12) {
                current.PM = ++current.PM % 2;
        #endif
            current.hours = 0;
    }
#endif
    }
}
When we compile our program, we have to set each of these macros to either defined or undefined. The code contained in an ifdef will be included only if the corresponding macro is defined. For example, the PM variable in the time struct, which keeps track of whether the current time is AM or PM, is included only if the _12H macro is defined.

[Driver points to “PM.”]

If an ifdef has an else-branch, this code is included only if the macro is undefined. For example, if _24H is undefined, then the second initialization of the current variable will be used, which also initializes the PM variable.

[Driver points to else branch of initialization block.]

Finally, an elif branch like the one in the tick function...

[Driver points to elif in tick function.]

... is included only if the first macro is undefined and the second macro is defined. In this case, the elif branch will be included if _24H is undefined, and _12H is defined.

Are there any questions about how ifdefs work, before we move on? [Wait.]

Estimating the number of variants with CPP

Ok, by now you’ve probably noticed that we don’t need both of the macros used in this file. We could use just the _12H macro, for example, and include all of the code specific to 24-hour clocks in the else branches of the ifdefs.

In fact, having two macros for this situation is risky, because somebody could compile this code with either both of these macros defined, or with both of them undefined, and we would get a nonsensical program.

We call each possible setting of macros a “configuration.” Can somebody tell me how many configurations we have for this program? [Wait.]

Right, we have four configurations, since we have two macros and each can be independently set to either defined or undefined. During the study we will be asking questions about the number
of configurations a program contains.

We will also be asking questions about whether or not each configuration produces a unique program. In this case, every configuration produces a different program, so we also have four unique programs. Does everyone agree with this? [Wait.]

Finally, we will ask questions about how many programs you think the programmer intended to specify, which might be fewer than the number they actually specified. Can somebody tell me how many programs we intended to specify here? [Wait.]

Right, we intended to specify two programs, one for 12-hour clocks and one for 24-hour clocks.

Ok, so to summarize, we have three different ways of counting the number of variants in a piece of CPP-annotated code: we can count the total number of configurations, the total number of unique programs, and the total number of intended programs. Are there any questions about these concepts? [Wait.]

Segue

Ok, when one piece of software represents many different compiled programs, we call it “variational software.” Our research is all about trying to make variational software easier to understand.

In this study, we will be comparing your ability to understand variational software managed with ifdef statement and variational software managed with a prototype of an alternative tool.

Click on the “Next” link now to see this tool. [Wait.]

Introduction to Prototype

This tool is divided into two columns. In the left hand column, we can choose which variant we want to see.

[Driver circle dimension panel.]
Try clicking back and forth on the radio button on the left side of the screen to see how it affects the code displayed in the tool. [Wait.]

When you change your selection on the left side, the code on the right side also changes. Notice that when you choose “12H” you see the code corresponding to a 12-hour clock, and when you choose “24H” you see the code corresponding to the 24-hour clock.

Code that is highlighted in the right-hand side represents code that is only included for this particular selection.

Does anyone have any questions about this tool so far? [Wait.]

Ok, click on the “Next” link now. [Wait.]

Adding a Second Dimension

Try playing around with all of the different selections in the left hand column now. [Wait.]

What we’ve done here is added a second dimension of variation. A dimension is basically just a way to structure how a program varies. You can think of it as a way of grouping related

typedef struct {
    int hours;
    int minutes;
} time;

time current = {0, 0};

// called once every minute
void tick() {
    if (++current.minutes == 60) {
        current.minutes = 0;
        if (current.hours == 24) {
            current.hours = 0;
        }
    }
}

Figure A.3: Introduction to Prototype
```c
typedef struct {
    int hours;
    int minutes;
    int PM;
} time;

time alarm = {0, 0, 0};
time current = {0, 0, 0};

void setAlarm(int h, int m, int p) {
    alarm.hours = h;
    alarm.minutes = m;
    alarm.PM = p;
}

void checkAlarm() {
    if (current.hours == alarm.hours
        && current.minutes == alarm.minutes
        && current.PM == alarm.PM) {
        printf("bzzzzz!
");
    }
}

// called once every minute
void tick() {
    if (++current.minutes == 60) {
        current.minutes = 0;
        if (++current.hours == 12) {
            current.hours = 0;
            current.PM = ++current.PM % 2;
        }
    }
    checkAlarm();
}
```

Figure A.4: Adding a Second Dimension
Now we have two dimensions. The “Time System” dimension allows us to choose whether we have a 12-hour clock or a 24-hour clock, and the “Has Alarm” dimension lets us choose whether or not our clock has an alarm functionality. Optional code in the right hand side is color-coded with the selection in the left hand side that caused it to be included.

We won’t explain the alarm code in depth. Basically it just adds a new variable storing an alarm time, and provides a way to check to see if the current time equals the alarm time.

We do want to point one thing out, however. Select the 12H option in the Time System dimension and the Yes option in the Has Alarm dimension. Your screen should look like the one on the projector. [Wait.]

Notice that here we have some pink code inside of the blue code. This is code that will only be included if both of the corresponding options are selected in the left hand column.

Does anyone have any questions about dimensions or the prototype tool? [Wait.]

Ok, the final part of this tutorial will be an example of a simple sample task, just so you can see how those will work. Please click the next link now.

**Sample Task**

You should also see a question at the top of your screen with a box below it to enter your answer. During the actual test, both your answers and your response time will be recorded for each question. Try to answer the questions as quickly and accurately as possible, but accuracy is more important than speed.

Can somebody tell me the correct answer to this question? [Wait.]

Right, it’s 4, so enter 4 and click the “Next Question” button.
Once you submit an answer, you can never go back to a previous question. So make sure you really want to click that button before you click it!

Ok, now, the next question is presented. Try to answer this question and the next one. Remember, your results aren’t being recorded yet, this is just for practice.

Once you’ve answered both questions, you should see a dialog pop up. Has everyone made it to the end? [Wait.]

Ok, you can click away the dialog and close the tutorial windows now. Congratulations, you finished the tutorial!

Are there any questions about the structure of tasks? [Wait.]

Alright, we’re just about ready to start the study. The study will be divided into two sections. For each section, you will log in and answer a series of questions. You will be logged out automatically after the first section. If you need to rest, you should do so in between the sections, before logging back in again.

To start each section, click on the “Start Task” shortcut on the desktop, then click on the link in the browser, log in using the username and password you were provided.

Are there any final questions before we get started? [Wait.]

Ok, you may now log in and begin the study! Thanks for your attention and good luck!
A.2 Questionnaires

A.2.1 Pre-Study Questionnaire

Please circle or write answers below that most accurately describe your personal background and experience.

1. What is your gender?
   
   Female | Male | Other: 

2. What is your current academic year?
   
   Freshman | Sophomore | Junior | Senior | Other: 

3. How many college-level programming courses have you taken?
   
   Number of courses: 

4. Have you used the C programming language and/or C Preprocessor (CPP) macros like #ifdef in your course work? If so, in how many courses?
   
   Number of courses using C and/or CPP:

5. Do you have experience programming professionally?
   
   no | yes

6. Do you contribute to any open source projects?
   
   no | yes

7. Have you used C and/or CPP professionally or on an open source project?
   
   no | yes

8. Do you use C and/or CPP in your own personal projects?
   
   no | yes
9. On a scale from 1 to 5, do you consider yourself a beginning C and CPP programmer, an expert C and CPP programmer, or somewhere in the middle?

(Beginner) 1 | 2 | 3 | 4 | 5 (Expert)

A.2.2 Post-Study Questionnaire

Example Difficulty

The study was divided into three sections, each one based on a different family of examples.

- OS: used examples where each variant corresponded to different operating systems
- doThis: used examples where a function named doThis or doThat was called at different times and the action was optionally logged.
- opcode: used examples that were interpreters for a very simple machine language, with instructions to add, jump, and skip.

In the following table, please circle the answer that most closely represents how you feel about the difficulty of the examples and questions used in the study.

<table>
<thead>
<tr>
<th>Question</th>
<th>strongly disagree</th>
<th>disagree</th>
<th>neither</th>
<th>agree</th>
<th>strongly agree</th>
</tr>
</thead>
<tbody>
<tr>
<td>The OS examples were difficult to understand.</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>The doThis examples were difficult to understand.</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>The opcode examples were difficult to understand.</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

Understanding CPP Annotated Code

In the following table, please circle the answer that most closely represents how you feel about the understandability of CPP annotated code (code that contains #ifdef statements).
In CPP annotated code, it was easy to determine the total number of configurations (possible settings of macros).

In CPP annotated code, it was easy to determine how many unique program variants could be generated.

In CPP annotated code, it was easy to determine how many program variants the programmer intended to specify.

In CPP annotated code, it was easy to determine which macros were supposed to be mutually exclusive (only one can be defined at a time).

In CPP annotated code, it was easy to understand how a particular generated program would function.

In CPP annotated code, it was helpful to be able to see multiple program variants at the same time.

In general, the CPP annotated code was easy to understand.

<table>
<thead>
<tr>
<th>Question</th>
<th>strongly disagree</th>
<th>disagree</th>
<th>neither</th>
<th>agree</th>
<th>strongly agree</th>
</tr>
</thead>
<tbody>
<tr>
<td>In CPP annotated code, it was easy to determine the total number of configurations (possible settings of macros).</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>In CPP annotated code, it was easy to determine how many unique program variants could be generated.</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>In CPP annotated code, it was easy to determine how many program variants the programmer intended to specify.</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>In CPP annotated code, it was easy to determine which macros were supposed to be mutually exclusive (only one can be defined at a time).</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>In CPP annotated code, it was easy to understand how a particular generated program would function.</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>In CPP annotated code, it was helpful to be able to see multiple program variants at the same time.</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>In general, the CPP annotated code was easy to understand.</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

**Understanding Code in the Prototype**

In the following table, please circle the answer that most closely represents how you feel about the understandability of code presented in the prototype.
<table>
<thead>
<tr>
<th>Question</th>
<th>strongly disagree</th>
<th>disagree</th>
<th>neither</th>
<th>agree</th>
<th>strongly agree</th>
</tr>
</thead>
<tbody>
<tr>
<td>In the prototype, it was easy to determine the total number of configurations (possible option selections).</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>In the prototype, it was easy to determine how many <em>unique</em> program variants could be selected.</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>In the prototype, it was easy to determine how many program variants the programmer <em>intended</em> to specify.</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>In the prototype, it was easy to understand how a particular selected program would function.</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>I understood the concept of grouping options into a “dimension” of variation.</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>In the prototype, it was helpful to be able to see a single program variant by itself.</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>In general, code in the prototype was easy to understand.</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

**Comparing CPP and the Prototype**

In the following table, please circle the answer that most accurately describes which tool you think is better suited for each type of task.
<table>
<thead>
<tr>
<th>Question</th>
<th>much easier in CPP</th>
<th>easier in CPP</th>
<th>neither</th>
<th>easier in prototype</th>
<th>much easier in prototype</th>
</tr>
</thead>
<tbody>
<tr>
<td>In which tool was it easier to count the number of configurations and program variants?</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>In which tool was it easier to understand a particular program variant?</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Overall, which tool made it easier to understand software variation (code that represents many different programs)?</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

Thank you for your participation in this research study! Please turn this questionnaire in to one of the study helpers to collect your $20 and your receipt.