How Does Feature Dependency Affect Configurable System Comprehensibility?

Djan Santos
Federal Institute of Bahia
Vitória da Conquista, Bahia, Brazil
Federal University of Bahia
Salvador, Bahia, Brazil
djan.santos@ifba.edu.br

Cláudio Sant'Anna
Department of Computer Science
Federal University of Bahia
Salvador, Bahia, Brazil
santanna@dcc.ufba.br

***Abstract*—Background: Conditional compilation is often used to implement variability in configurable systems. This technique relies on #ifdefs to delimit feature code. Previous studies have shown that #ifdefs may hinder code comprehensibility. However, they did not explicitly took feature dependencies into account. Feature dependency occurs when different features refer to the same program element, such as a variable. Comprehensibility may be even more affected in the presence of feature dependency, as the developer must reason about different scenarios affecting the same variable. Aim: Our goal is to understand how feature dependency affects the comprehensibility of configurable system source code. Method: We carried out an experiment in which 30 developers debugged programs with different types of feature dependency. We recorded the time each of them had spent to find a bug. Also, we used an eye-tracking device to record developers' gaze movements while they debugged programs. Results: Debugging programs with global and interprocedural dependency required more time and higher visual effort. Conclusion: Our study showed that #ifdefs affect comprehensibility in different degrees depending on the type of feature dependency. Therefore, when possible, developers should take more care when dealing with code with global and interprocedural dependencies.**

***Index Terms*—Feature dependency, Program comprehension, Configurable Systems, Variability bugs, Eye tracking**

I. Introduction

Configurable systems address variability by means of features that can be enabled or disabled [1], [2]. There are large industrial product lines [1], [3] and open-source systems, like the Linux kernel, that are examples of configurable systems [4], [5]. One of the techniques most used to allow variability is conditional compilation. By means of preprocessor directives, like #ifdef, this technique allows developers to include or exclude code fragments that will or will not be compiled [6], [7], [2]. While programming, developers use #ifdefs to delimit code fragments related to optional or alternative features. Then, only features explicitly enabled are compiled.

Previous studies have shown that the presence of #ifdefs might hinder code comprehensibility [8], [9], [10]. For instance, Schulze et al. [9] showed that #ifdefs make debugging more difficult and time-consuming. Melo et al. [8] compared pieces of source code with and without #ifdefs and confirmed that #ifdefs increased debugging time and visual effort. In addition, many studies have found bugs that occur due to the use of preprocessor directives [2], [11], [12], [13], [4], [5]. These bugs are called as variability bugs [4], [5].

Configurable systems usually include a high number of features. Thus, two or more features are likely to share code fragments. When different features refer to the same program element, such as a variable, we have an occurrence of *feature dependency* [14]. Depending on the scope of the variable shared between features, Rodrigues et al. [14] defined three types of feature dependency: global, intraprocedural and interprocedural. Feature dependencies are common in practice [15]. However, little is known about how #ifdefs affect comprehensibility when their use causes different types of feature dependency. In fact, previous studies do not explicitly take feature dependencies into account when investigating the impact of #ifdefs on comprehensibility.

Conditional compilation forces developers to consider multiple scenarios while trying to comprehend source code. The effort for understanding source code that contains feature dependencies may be even higher, as developers must reason about different scenarios affecting the same variable. To investigate this hypothesis, we carried out a controlled experiment to study how feature dependency affects comprehensibility of configurable systems implemented with #ifdefs.

In our experiment, thirty developers performed tasks on which they tried to find different variability bugs in programs with feature dependency. Each developer analyzed programs with different types of feature dependency. They also analyzed programs with and without #ifdefs. We recorded the time each developer spent to debug each program. Also we counted the number of bugs each developer succeeded to find. Evaluating software comprehensibility is a challenge because we need to consider developer's human factors, such as visual effort [16], [17]. Considering this, we also recorded developers' eye movements using an eye-tracking device.

Our findings show that feature dependencies affected program comprehensibility in different ways: (i) global and interprocedural dependencies required more debugging time than intraprocedural dependencies, (ii) interprocedural dependencies required higher visual effort than the other two types of dependency, (iii) #ifdefs did not affect comprehensibility in programs with intraprocedural dependencies, and (iv) feature dependency did not affect the number of bugs correctly found. In a nutshell, our results show that #ifdefs may hinder code comprehensibility mainly when they cause global an interprocedural dependencies.

The rest of the paper is organized as follow. Section II explains conditional compilation and feature dependency. Section III describes our experiment settings. Section IV discusses the experiment results. Section V discusses threats to validity. Section VI discusses related work. Section VII presents our conclusions and future work.

II. Conditional Compilation and Feature Dependency

Conditional compilation allows variability. In this case, variability means that specific features can be compiled or not. For this, programmers often use C preprocessor (Cpp), which allows the use of annotation directives (#ifdef) to indicate the program elements, such as variables and functions, that implement specific features [6], [7]. Listing 1 illustrates how developers use Cpp to implement features by adding #ifdefs to source code. Between lines 2 and 5 and lines 8 and 10, we have a source code that implements the `APPLY_PENALTY` feature. It will be compiled only if, before compiling the program, the developer sets `APPLY_PENALTY` as enabled.

Listing 1. Example of Conditional Compilation and Global Dependency

```
 1 float final_grade=0;
 2 #ifdef APPLY_PENALTY
 3 const float penalty = 1.5;
 4 int exceeded_time = 4;
 5 #endif
 6 int main() {
 7 ...
 8 #ifdef APPLY_PENALTY
 9 final_grade = final_grade - (exceeded_time *
       penalty);
10 #endif
11 #ifdef RESULT
12 if (final_grade >= 0 && final_grade < 5)
13         printf ("disapproved");
14 else if(final_grade >=5 && final_grade <7)
15         printf ("approved to MS.C.");
16 else
17         printf ("approved to Ph.D.");
18 #else
19         printf ("Final Grade = %f", final_grade);
20 #endif
21 ...
```

Conditional compilation implies in feature dependency whenever a developer defines a variable in a feature and uses it in another feature or when two or more features use the same variable [14]. For instance, in Listing 1, the `final_grade` variable is defined outside any #ifdef, that is, it is defined in a mandatory feature, which, in this example, we can call as final grade calculation. Also, `final_grade` is used in features `APPLY_PENALTY` (line 9) and `RESULT` (lines 12, 14 and 19). Therefore, there is feature dependency between: (i) the mandatory feature and `APPLY_PENALTY`, (ii) the mandatory feature and `RESULT` and (iii) features `APPLY_PENALTY` and `RESULT`. In this example, the `final_grade` variable is called as *dependent variable* as it is the element that causes the dependency.

Taking Listing 1 as a motivating example, we can see that, to understand the behavior of `final_grade`, the developer should at least consider: both `APPLY_PENALTY` and `RESULT` features enabled, both `APPLY_PENALTY` and `RESULT` disabled, `APPLY_PENALTY` enabled and `RESULT` disabled, and vice-versa. Mentally simulating all these scenarios may increase the program comprehension effort.

Based on the definition and use scope of the dependent variable, Rodrigues et al. [14] define three types of feature dependency: global, intraprocedural and interprocedural.

Global dependency occurs when different features refer to the same global variable. Listing 1 presents an example of global dependency, as `final_grade` (line 1) is a global variable. As explained before, `final_grade` is the dependent variable, in this example.

Listing 2. Example of Intraprocedural Dependency

```
 1 void return_book( ) {
 2 #ifdef FINE_IN_CASH
 3 float fine_rate_day = 1.5, fine_amount = 0;
 4 #endif
 5 #ifdef PUNISHMENT_IN_DAYS
 6 int days_of_punishment=1, days_punished = 0;
 7 #endif
 8 int days_delay = 2;
 9 if(days_delay>0){
10 #ifdef FINE_IN_CASH
11         fine_amount= days_delay * fine_rate_day;
12 #ifdef PUNISHMENT_IN_DAYS
13         days_punished= days_delay *
               days_of_punishment;
14 #endif
15 #endif
16 }
17 ....
18 int main() {
19         return_book( );
20 ...
```

Listing 3. Example of Interprocedural Dependency

```
 1 float calc_points(float total_points){
 2 ...
 3 #ifdef CONSIDER_SPECIALIZATION
 4         bool isSpecialist = true;
 5 #endif
 6 #ifdef CONSIDER_SPECIALIZATION
 7         if(isSpecialist)
 8                 total_points += 1;
 9 #endif
10 ...
11 return total_points;
12 }
13 int main() {
14         float total_points;
15         total_points = calc_points(total_points);
16         printf("Total Points = %.2f", total_points
               );
17 ...
```

Intraprocedural dependency occurs when different features inside a function refer to the same variable, which is local to that function. Listing 2 illustrates an intraprocedural dependency. In this case, the dependent variable is `days_delay`, which is defined within `return_book()` function. Still within `return_book()`, `days_delay` is used by features `FINE_IN_CASH` (line 11) and `PUNISHMENT_IN_DAYS` (line 13).

Interprocedural dependency occurs when a function defines or uses a variable and passes the content of the variable as argument to another function. In the first function, the variable is manipulated by one feature, and, in the second feature, the argument is used by another feature. Listing 3 illustrates an interprocedural dependency. The `total_points` variable is defined in the function `main()` (line 14), which implements a mandatory feature. Its content is passed to `calc_points()`, where the corresponding parameter is used by the `CONSIDER_SPECIALIZATION` feature (line 8).

III. Experiment Settings

The main goal of our experiment is to evaluate the impact of different types of feature dependency on the comprehensibility of configurable systems. If a specific type of dependency makes it more difficult to understand the source code, configurable system developers should take care when either maintaining or testing code fragments with such type of dependency. In this context, the following research question guides our study:

RQ – *How do different types of feature dependency affect the comprehensibility of configurable system source code?*

A. Design

To answer our research question, we carried out a controlled experiment with 30 developers, who analyzed programs trying to find bugs. They analyzed different programs each with a different type of feature dependency (global, intraprocedural and interprocedural). We compared the comprehension effort they spent to analyze each program. We quantified comprehension effort from different perspectives: (i) time to analyze each program, (ii) number of correctly found bugs, and (iii) visual effort. We quantified visual effort by means of different metrics collected by the use of an eye-tracking device. We give more details about these metrics in Section IV.

We also aimed to confirm whether the differences on comprehension effort were due to the different feature dependency types (implemented with #ifdefs) or were only due to differences related to variable scope, such as the use of global variables or local variables, regardless of the use of #ifdefs. To achieve this, we also asked the developers to analyze programs without #ifdefs, but equivalent to the ones with #ifdefs. For each program with #ifdef, we have an equivalent one without #ifdef. They are equivalent mainly in two aspects. First, in the programs without #ifdef, we replaced the #ifdefs with regular *if* statements. Second, the program without #ifdef follow the same structure in terms of variable use of its equivalent with #ifdef. For instance, a program with intraprocedural dependency has an equivalent with the same function and local variable, but with *ifs* as replacements of #ifdefs. In Section III-C, when describing the bugs we used in our experiment, we give an example of two equivalent programs with and without #ifdefs.

In order to avoid learning effect, we selected six different variability bugs to compose each program: (i) null pointer dereference, (ii) assertion error, (iii) variable overlap, (iv) nested feature, (v) undefined variable, and (vi) uninitialized variable. Related literature has reported these bugs as recurring in configurable systems [4]. Section III-C describes each of them. Also to avoid learning effect, we had programs on six different domains, each domain for a variability bug. For example, we implemented the null pointer dereference variability bug in a program on the "sales authorization message" domain.

In summary, for each variability bug, we implemented six similar programs on the same domain, each one with the following characteristics: (i) global dependency with #ifdef (GI), (ii) equivalent to global dependency without #ifdef (GW), (iii) intraprocedural dependency with #ifdef (IAI), (iv) equivalent to intraprocedural dependency without #ifdef (IAW), (v) interprocedural dependency with #ifdef (IEI) and (vi) equivalent to interprocedural dependency without #ifdef (IEW). Therefore, we implemented 36 programs.

We designed our experiment as a standard Latin square, which is a common solution for this kind of experiment [18], [19], [8]. We can explain the 6x6 Latin square we adopted by means of Table I (Section IV). In its columns, we have the variability bugs. The lines represent the groups of developers. We divided the developers in six groups (G1 to G6). The acronyms in the cells bellow columns "Characteristic" represent the program characteristics, as listed in the previous paragraph. For instance, developers in group G1 analyzed the program with *global dependency with #ifdef (GI)* that has the *null pointer dereference bug* (first column of G1 line). Developers in group G2 also analyzed the program with *global dependency with #ifdef (GI)*, but the one with the *uninitialized variable bug (VI)* (three last columns of G2 line). The values in Table I correspond to mean of time developers spent analyzing the programs (time column) and number of bugs they correctly found (hits). We discuss them in Section IV.

The Latin square design controls one factor and its variations, ensuring that no row or column contains the same treatment twice. Our factor was the program characteristic (type of dependency + with/without #ifdef). Each line of our Latin square has six programs with different characteristics arranged in different orders. However, there might still be learning effects due to repetition of variability bugs. We avoided this by distributing the variability bugs along our Latin square columns. Each column has a different type of variability bug (without repetition). Therefore, according to his/her designated group, each developer debugged six different programs, each with a different variability bug (and its respective domain) and each with a different type of feature dependency (with and without #ifdef). The result is the same number of data points for all debugging tasks. As we have 30 participants, we obtained 180 data points, five data points for each of the 36 programs.

B. Participants

We counted on 30 participants to run our experiment: six undergraduate students, six MSc students, six PhD students, six professors, and six developers from industry. We put them in six groups with five participants each. Randomly, we formed each group with one participant of each category, i.e., one undergraduate student, one Msc student and so forth.

We selected the students from three universities located at three different cities of Brazil and the developers from two companies located at two different cities of Brazil. No compensation was provided for the participants.

Out of these 30 participants, one participant's eye tracking data were discarded due to poor quality. This was due to technical issues with the eye tracker. Eighteen participants have normal vision and 12 have vision corrected by glasses. Seven participants are females and 23 are males. All the participants have similar experience in C programming language. Sixteen participants declared themselves as expert developers and only three claimed that had not worked at the industry yet.

C. Variability Bugs

When a fault or error happens in a configurable system due to variability implementation, it is called as variability bug [4]. Previous studies related variability bugs to feature dependencies [4], [20]. We implemented the programs used in our experiment inspired in concrete variability bugs, which occurred in real large-scale configurable systems [4], [8], [5]: Linux [4], [5], BusyBox [21], [8], BestLap [20], GLib [14] and Libxml [14]. However, we could not use source code of large-scale systems in our study due to the following restrictions:
Domain. Our programs should be on domains which participants could easily understand. We avoided programs (like the ones from Linux) which could affect comprehensibility.
Native language. To facilitate the understanding and to widen the audience of potential participants, the programs should be written in the participants' native language (Portuguese).
Small programs. The programs should fit on a 39-line display window so that the eye-tracking device would record all gaze movements of participants.

Having these restrictions in mind, we took an example of each selected variability bug from a real configurable system or from previous studies. Then, we reproduced that bug in a small program, on a popular domain, written in C with variable and feature names in the participants' native language.

The 36 programs are similar in terms of number of lines of code (LOC) [22], number of features (NOFC) [7] and McCabe cyclomatic complexity (CC) [23]. In the following, we describe each variability bug we implemented.

Null pointer dereference. This bug happens when a program attempts to read a value from a null pointer. Listing 4 shows a code snippet of the program we wrote with a null pointer dereference bug. It is about "printing a sales authorization message". It has two features: `CUSTOMIZE_MESSAGE` and `SETUP_COMMUNICATION`. `CUSTOMIZE_MESSAGE` personalizes the message with the name of the store and `SETUP_COMMUNICATION` checks communication errors. An exception occurs when `CUSTOMIZE_MESSAGE` is disabled. It happens because `p` is updated within `CUSTOMIZE_MESSAGE`. The expression `if(*p =="")` (line 12) causes a null-pointer exception because `p` has null value and cannot be compared with empty. We wrote this program taking as example a bug found in BusyBox [21], [8], an open source system that provides essential Unix tools.

Listing 4. Code snippet of the null pointer dereference bug with #ifdef
```
 1 char *p = NULL;
 2 char message[25];
 3 ...
 4 int main() {
 5 char msg[25];
 6 ...
 7 #ifdef CUSTOMIZE_MESSAGE
 8     strcpy(message,"Store XYZ - ");
 9     p = message;
10 #endif
11 if(*p =="")
12     strcpy(message, msg);
13 ...
```

Listing 5 shows the version of the program with null pointer dereference bug now without #ifdefs. We rewrote the program showed in Listing 4 by replacing #ifdefs with `if` clauses. The bug remains the same in Listing 5, as line 12 executes the expression `if(*p =="")`, which causes a null-pointer exception. However, in this case, the bug is caused by a variable whose value is false, and not by disabling a feature. It is important to recall that Listing 4 shows the program with global dependency with #ifdef and Listing 5 shows its equivalent without #ifdef. Besides them, we also had in our experiment other four programs with the null pointer dereference bug, which implement intraprocedural and interprocedural dependencies and its equivalent without #ifdefs. For the bugs we describe next, we only show one of the programs with #ifdef.

In some of the programs, there are #ifdefs surrounding variable definitions (for instance, Listing 2, lines 2 to 4), or #ifdefs surrounding`else` clauses (for instance, Listing 6, lines 11 to 13). In this cases, we just deleted the #ifdefs and did not replace them with `if` clauses. We did this because, in fact, converting those #ifdefs into `ifs` does not make sense.

Listing 5. Code snippet of the null pointer dereference bug without #ifdef
```
 1 char *p = NULL;
 2 char message[25];
 3 bool customize_message = false;
 4 int main() {
 5 char msg[25];
 6 ...
 7 if (customize_message){
 8     strcpy(message,"Store XYZ - ");
 9     p = message;
10 }
11 if(*p =="")
12     strcpy(message, msg);
13 ...
```

Assertion-Error. An Assertion-Error occurs when something that should never happen happens. This bug was found in BestLap, which is a commercial highly configurable race game, investigated by previous research [15], [20], [21], [8]. The car racing game calculates score of players and adds a penalty when their cars crash. As the score should not be negative, the assertion error occurs when the score stores negative values. Based on BestLap, we wrote a program that "calculates scores in language proficiency tests". We apply penalties in case the exam time is exceeded. Listing 1 (Section II) shows a code snippet of one of the programs with this bug. It has two features: APPLY_PENALTY and RESULT. APPLY_PENALTY applies penalties in case the exam time is exceeded, and RESULT prints the final result. The bug occurs when both APPLY_PENALTY and RESULT are enabled and final_grade is negative. In this case, instead of generating an error, the program prints a wrong message (line 17).

Logic error: Logic error occurs when a program produces unintended or undesired output. Logic errors are often the most general errors and hardest to identify [24]. Listing 6 shows an excerpt of our program that "calculates the value of payments if the customer decides to purchase in 3 installments". The customer can use credit card or checks to split a sale. If he or she chooses checks, the program adds an interest of 5% to the purchase value. If the sale is paid on a credit card, the program does not add any interest. The customer also may choose not to split the purchase. The CREDIT_CARD and CHECK features calculate instalment values without and with interest, respectively. Note that both features enable the program to split the purchase in three installments, but the purchase will be split whether the value of use_card (line 5) or use_checks (line 9) is true. The logic error occurs when CREDIT_CARD and CHECK are enabled. When the two features are enabled, the clause else (line 13) is associated with the scope of clause if of use_checks only. Thus, if use_card (line 5) is true and use_checks (line 9) is false, the value of instalments is overlapped.

Listing 6. Code snippet of the logic error

```
1  int main() {
2  ...
3  #ifdef CREDIT_CARD
4  if (use_card)
5      instalments = purchase/3;
6  #endif
7  #ifdef CHECK
8  if (use_checks)
9      instalments=(purchase + (purchase * 0.05))/3;
10 #endif
11 #ifdef CREDIT_CARD || CHECK
12 else
13 #endif
14 instalments = purchase;
15 ...
```

Nested features. Previous studies [7] reported that nesting features make the source code prone to errors [7], [25]. Listing 2 (Section II) shows an excerpt of our program where nested features cause a variability bug. The program processes the "return of books to a library". If a book is returned after the due date, the system applies either a fine or a penalty in days during which the user will be unable to make new loans. FINE_IN_CASH calculates the fine (line 11) and PUNISHMENT_IN_DAYS calculates the penalty in days. Note that PUNISHMENT_IN_DAYS (line 12) is inside FINE_IN_CASH (line 10). The bug happens when FINE_IN_CASH is disabled, because doing this also disables PUNISHMENT_IN_DAYS.

Undefined variable. This bug happens when a variable is not previously declared but it is accessed later on. To write our program we took as example a bug found in Libxml[1], a configurable system for parsing XML files [14]. Listing 7 presents a code snippet of our program with this variability bug. It "calculates the registration fee of an event". The program applies discounts for students or attendees with membership association. The program has two features called STUDENT_DISCOUNT and MEMBERSHIP_ASSOCIATION. STUDENT_DISCOUNT applies a discount for students (lines 3, 7 and 10). MEMBERSHIP_ASSOCIATION applies a discount for attendees with a membership association (line 7). The bug happens because the discount variable is defined only if either STUDENT_DISCOUNT or MEMBERSHIP_ASSOCIATION is enabled (line 8). If both features are disabled, discount is undefined and the program runs into an undefined variable error, because discount is used ahead (in line 15).

Listing 7. Code snippet of the undefined variable bug

```
1  ...
2  float calculateRegistrationFee() {
3  #ifdef STUDENT_DISCOUNT
4      bool apply_discount_student = true;
5  #endif
6  ...
7  #ifdef STUDENT_DISCOUNT || MEMBERSHIP_ASSOCIATION
8      float discount=0;
9  #endif
10 #ifdef STUDENT_DISCOUNT
11     if (apply_discount_student)
12         discount += 0.1;
13 #endif
14 ...
15 registration = registration - (registration *
       discount);
16 ...
```

Uninitialized variable. This bug happens when a variable is declared but is not set before its use. Abal *et al.* show this is a frequent bug in Linux kernel [4], [5]. Our program with this bug "calculates points based on a students' curriculum" (Listing 3 (Section II)). The number of points of a student increases with the number of courses he or she accomplished. When CONSIDER_SPECIALIZATION is enabled, total_points receives one extra point (line 8), if isSpecialist is true. The variability bug occurs because total_points is not initialized before (line 14).

D. Procedure

Before executing the actual experiment, we carried out three pilot studies. In the first one, in addition to find bugs, we asked participants to fix them. However, they took so long to do that. Moreover, we had problems to record their gaze movements as our eye-tracking device does not work well on non-static screens. So, we changed our experiment procedures for the second and third pilot studies: the participants only had to find bugs analyzing a small program showed in a static image, which they could not manipulate. The second and third pilot studies were mainly useful for us to correct problems with the programs. We performed the pilot studies with four PhD students. We did not consider their results in our analysis.

[1]http://xmlsoft.org/

Before starting the experiment tasks, we briefly trained every developer on conditional compilation, variability, features and system configuration. Then, we calibrated the eye-tracking device and performed a small warm-up task. All the participants signed a consent form.

The participants debugged the programs as we planned in our Latin square design. When the participant indicated he or she finished analyzing a program, we registered the time. Finally, to check whether he or she correctly found the bug, we asked each developer: "how would you fix this bug?".

We presented each program to the participants as static images displayed on a single screen. Participants did not have access tools, IDEs or browsers. For each participant, we recorded x and y coordinates (fixations) via an eye tracker.

We performed each experiment trial individually. All participants used the same personal computer to avoid unintended effects from different software and hardware environments. The computer has the following configuration: a 64-bit windows 10 home single language with Intel core i5. The screen resolution was set to 1920 by 1080 pixels into a 15 inch LCD screen. All experiment trials were conducted in similar classrooms. We recorded all of the eye tracking data using the open-source tool OGAMA [26]. We used the Tobii EyeX Device.

IV. Experimental Results

In this section, we test our hypotheses and discuss results. We measured comprehensibility according to: (i) time to find a bug, and (ii) number of correctly found bugs. We also measured developers' visual effort: (i) number of fixations, and (ii) gaze transitions. We also analyzed attention maps generated by means of the eye tracker.

We ran ANOVA tests for hypothesis testing. We used p-value < 0.05 as the probability about rejecting null hypotheses. The only exception, was the *number of found bugs* variable. ANOVA does not apply for it as it holds binary values. Thus, we used inferential statistics to evaluate it. We ran our tests with the support of R[2]. All artifacts used in our experiment are available at our website[3]. In the following, we present the results regarding each metric.

A. Time to find bugs

We measured the time (in seconds) each participant took to analyze each program, similarly as Sharif et al. [27] did. Our null hypothesis about this metric is:

H_0t: There is no significant difference in the time to find bugs when comparing programs with different types of feature dependency.

Rows in Table I show the mean time (column time) spent by each group of participants (G1 to G6) for each characteristic of program and for each variability bug. Each group was compose by 5 participants. Consequently, we had 5 observations for each group, 30 observations for each row, which summed up as a total of 180 observations. Shapiro test confirmed that the data about time to find bugs was normally distributed.

[2]http://www.r-project.org/

[3]http://www.djansantos.com.br/projects/FeatureDependency/

Table II shows the mean time spent by all participants for each type of dependency. For example, they spent a mean time of 323.10 seconds to analyze programs of global dependency with #ifdefs (GI). For global dependency without #ifdefs (GW), the mean time was 228.70 seconds. Our data revealed that there was a significant difference in time for the developers to analyze different types of dependency (p-value = 5.613e-05). We, thus, reject our null hypothesis (H_0t).

We, then, used Tukey HSD (honestly significant difference) test to find means of time that are significantly different. Tukey HSD test compares all possible pairs of means of time. First, we compared the mean time related to programs with #ifdefs (GI, IAI and IEI) (Table II). The difference of time between GI and IEI is negligible (p-value = 1.00). In contrast, when comparing IAI with GI and IEI, the difference is significant (p-value = 0.045 and 0.049, respectively). This leads to our first result.

Result 1: Global dependencies and interprocedural dependencies required more time for finding bugs than intraprocedural dependency.

We also compared the time spent for analyzing the three types of programs without #ifdefs (GW, IAW and IEW) (Table II). According Tukey HSD test, the time to analyze each of them is not significantly different (all p-values are larger than 0.126). This, somehow, reinforces that the differences stated in Result 1 are due to the use of #ifdefs.

Curiously, our data also revealed that the mean time is not significantly different when we compare programs with intraprocedural dependency with and without #ifdefs (IAI vs. IAW) (p-value = 0.961). This leads to our second result, which, regarding a specific context, contradicts previous study that says that #ifdefs increase debugging time [8].

Result 2: The use of #ifdef did not increase bug detection time in programs with intraprocedural dependency characteristic.

We also compared programs with interprocedural characteristic with and without #ifdefs (IEI vs. IEW). Similarly to intraprocedural programs, our results showed negligible difference in time for bug detection (p-value = 0.657).

We also compared global dependency with and without #ifdefs (GI vs. GW) (Table II). In this case, the difference in bug detection time is significant (p-value = 0.001).

Result 3: The use of #ifdefs increases bug detection time for programs with global dependency characteristic.

B. Number of correctly found bugs

The number of correctly found bugs metric refers to whether a participant answered each task correctly. It is about correctness [27]. If a participant finds a bug correctly, he or she scores one for that program. For this, we registered the answers provided by participants. Our null hypothesis about this metric is:

H_0a: There is no significant difference in number of correctly found bugs when comparing programs with different types of feature dependency.

TABLE I
OVERVIEW OF EXPERIMENT DESIGN AND RESULTS

	Variability Bugs																	
	Null pointer dereference			Assertion error			Logic error			Nested feature			Undefined variable			Uninitialized variable		
Group	Characteristic	time	hits	Characteristic	time	hits	Characteristic	time	hits	Characteristic	time	hits	Characteristic	time	hits	Characteristic	time	hits
G1	GI	338	3	GW	287	3	IAI	292	5	IAW	216	4	IEI	252	3	IEW	203	3
G2	GW	418	3	IAI	366	5	IAW	258	5	IEI	267	1	IEW	212	4	GI	182	5
G3	IAI	315	1	IAW	262	5	IEI	224	3	IEW	281	5	GI	123	5	GW	101	5
G4	IAW	383	1	IEI	339	4	IEW	224	4	GI	412	2	GW	362	5	IAI	234	5
G5	IEI	414	1	IEW	225	4	GI	194	2	GW	206	5	IAI	324	3	IAW	148	4
G6	IEW	461	3	GI	353	4	GW	390	3	IAI	287	4	IAW	288	5	IEI	168	5

TABLE II
MEAN TIME AND FOUND BUGS FOR EACH TYPES OF DEPENDENCY

Dependencies	GI	GW	IAI	IAW	IEI	IEW
Mean time	323.10	228.70	255.10	235.00	322.50	287.20
#found bugs	21	24	23	24	17	23

Table I shows the number of bugs correctly found (column hits) by each group of participants (G1 to G6) for each characteristic of program and for each variability bug. Table II shows the number of correctly found bugs by all participants for each type of dependency.

Comparing programs with #ifdefs, from a total of 30 tasks, 21 participants answered correctly the tasks about global dependency (GI), 23 answered correctly for intraprocedural dependency (IAI) and 17 for interprocedural dependency (IEI). Interprocedural dependency, therefore, seems to make bug detection more difficult. However, the χ^2 test (Pearson's Chi-squared test) [28] revealed no significant difference between the three types of feature dependency. The value χ^2 is less than 5.53, and the p-values are greater than 0.35. Based on this, we cannot reject our null hypothesis H_0a.

Result 4: There was no significant difference on the number of correctly found bugs for different types of feature dependency.

C. Visual effort

Visual effort is directly linked to the cognitive effort [29], [30], [27]. A set of eye-tracking measures representing visual effort are derived from eye gaze data. A fixation is a type of eye movement in which the eye stops on some object of interest to obtain information. Saccades are very fast voluntary movements between fixings. Regression is a saccade performed in the opposite direction to the reading direction [29].

The number of fixations is thought to be negatively correlated with search efficiency [31]. Also, the proportion of time looking at a particular display element could reflect the importance of that element [32], [33]. The literature sets a duration of 60 microseconds as the minimum threshold for having a fixation. Also, it sets a space of seven to nine letters for characterizing saccades [34]. We followed these thresholds and discarded anything below them [35], [36], [32].

Number of fixations

The number of fixations increases when a text is difficult to comprehend [34]. We counted the number of fixations per program. Our null hypothesis about this metric is:

H_0f: There is no significant difference in number of fixations when developers try to find bugs in programs with different types of feature dependency.

Considering programs with #ifdef, our data revealed that the number of fixations were significantly different between programs with different types of feature dependency (p-value = 5.613e-05). Analyzing programs with global dependency (GI), required from all participants a mean of 1011.93 fixations. For intraprocedural dependency (IAI), it required a mean of 741.56 fixations, and for interprocedural dependency (IEI), the mean was 1016.67 fixations.

Tukey HSD showed that the difference between global dependency (GI) and interprocedural dependency (IEI) is negligible (p-value = 0.99). In contrast, the number of fixations in programs with global and interprocedural dependencies is significantly higher than the number of fixations in programs with intraprocedural dependency (IAI) (p-value = 0.008 and 0.006 respectively).

Result 5: Developers made more fixations to understand programs with global and interprocedural dependencies than programs with intraprocedural dependencies.

In addition, when considering programs without #ifdefs (GW vs. IAW vs. IEW), our data shows that there is no significant difference among them in terms of number of fixation (all p values are larger than 0.9127183). This somehow reinforces that the differences stated in Result 5 are due to the use of #ifdefs and not due to the characteristic of the programs in terms of use of variables and functions.

Our study also revealed that the number of fixations is not significantly different when comparing data obtained for programs with intraprocedural characteristic with and without #ifdefs (IAI vs. IAW) (p-value = 0.86). This leads to the following result, which reinforces Result 2.

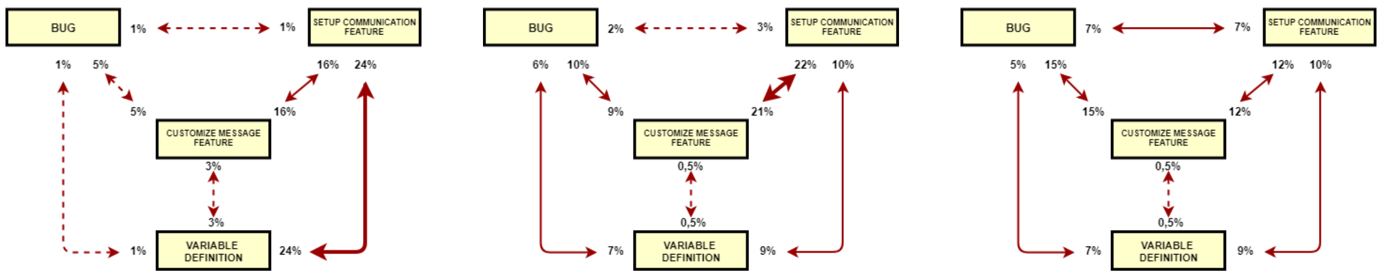
Result 6: The use of #ifdefs did not increase the number of fixation when developers try to find bugs in programs with intraprocedural dependency characteristic.

Finally, our results indicate that, regarding global and interprocedural characteristics, programs with #ifdefs required more number of fixations than the ones without #ifdefs (p-value = 2.1e-06 and 1.93e-04, respectively).

Result 7: The use of #ifdefs increased the number of fixations in programs with global and interprocedural characteristics.

Gaze transitions

Gaze transitions (a.k.a. saccades) are rapid eye movements from one place to another separated by pauses [34]. A larger



(a) intraprocedural (b) global (c) interprocedural

Fig. 1. Gaze transitions between AOIs for different types of feature dependency in programs with the null pointer dereferenced bug.

number of saccades in both directions indicates difficulty associated with understanding [36], [34], [29].

We compute gaze transitions based in areas of interest (AOI) of each program. An AOI is a region of interest in a study. We defined the AOIs with OGAMA's AOI editor. Our AOIs comprised regions of source code that showed variable definitions, feature code and bug regions. For example, in Listing 4, we define variables in lines 1 and 2. Thus, we defined this region as the AOI "variable definition". Lines 7 to 10 comprise source code of `CUSTOMIZE_MESSAGE` feature. We defined this AOI as "customize message feature". The same happens with the `SETUP_COMMUNICATION` feature (not shown in Listing 4). We defined its source code as AOI "setup communication feature". Finally, the AOI "Bug" is limited by lines 11 and 13, because it is where the bug occurs. The definitions of AOIs for each of our programs are detailed in our website.

Figure 1 shows our gaze transitions diagrams related to programs with the "null pointer dereferenced" bug. Arrows indicate the percentage of gaze transactions between different AOIs of the program in both directions. For example, Figure 1a depicts that 5% of all gaze transitions performed by participants to find the "null pointer dereferenced" bug happened from the AOI"bug" to the AOI "customize message feature". A dashed arrow indicates that the sum of gaze transactions in both directions corresponds to 10% or less of the total of gaze transactions. A bold arrow indicates that the sum of gaze transactions corresponds to 40% or more of the total of gaze transactions. A regular line indicates intermediate values.

Figure 1a shows the diagram for the program with intraprocedural dependency with #ifdef (IAI), Figure 1b shows the diagram for the program with global dependency with #ifdef (GI) and Figure 1c shows the diagram for the program with interprocedural dependency with #ifdef (IEI). The three are related to the "null point dereferenced" bug. We took this bug as an example for what similarly happens for other cases.

The gaze transitions diagram reveals that, for programs with global and intraprocedural dependencies, participants concentrated the largest number of saccades between few AOIs. Fig. 1a shows a concentration of about 48% (24% + 24%) of gaze transitions between the AOIs "variable definition" and "setup communication feature". Fig. 1b shows a concentration of about 43% (21% + 22%) of gaze transitions between AOIs "customize message feature" and "setup communication feature". On the other hand, Fig. 1c reveals that that participants, when debugging programs with interprocedural dependency, need to navigate over all source code, and, as a consequence, the gaze transitions are more distributed between all AOIs. Note in Fig. 1c that values do not exceed 30%. Although we only show here the diagrams for the "null pointer dereferenced" bug, we observed similar results for most of the other bugs.

Result 8: Interprocedural dependency seems to force the developer to perform more gaze transitions over different parts of the source code.

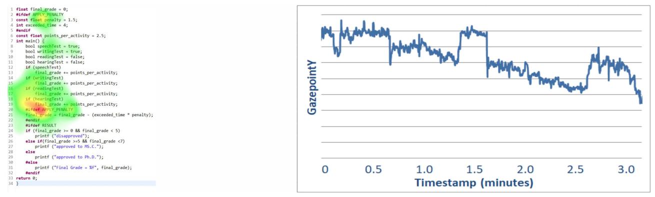
Attention map

An attention map is a histogram, also known as heat map, that displays an aggregation of fixations. An attention map shows how attention is distributed among program parts [37]. It uses colors to represent the fixation time in each location on the screen. Three examples are shown in Fig. 2. The lowest value in the attention map (short fixation time) is shown with the green color and the highest value in red (long fixation time), with a smooth transition between these extremes.

Figure 2 shows the aggregated attention maps for the "assertion error" variability bug for the three types of feature dependency. We generated the aggregated attention maps using OGAMA. We superimposed all individual attention maps from each participant. Each attention map of the "assertion error" variability bug (Fig. 2) is, thus, composed by the overlapping of five individual attention maps.

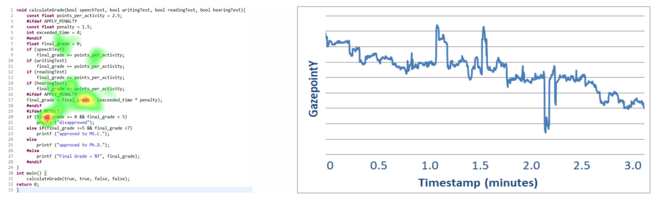
The red regions indicate where most of participants' attention was directed to. Comparing the red regions of the three attention maps in Fig 2, we observe that the attention distribution is similar for programs with global and intraprocedural dependencies. In these cases, participants focused most of their attention in the source code of the `APPLY_PENALTY` feature. The bug occurs inside `RESULTS`, when `APPLY_PENALTY` is enable. Thus, this area requires more attention from participants. In addition. in both programs, `APPLY_PENALTY` is near `RESULTS`.

On the other hand, in the attention map regarding interprocedural dependency (Fig. 2c), there are two distinct red regions. One region encapsulates the source code



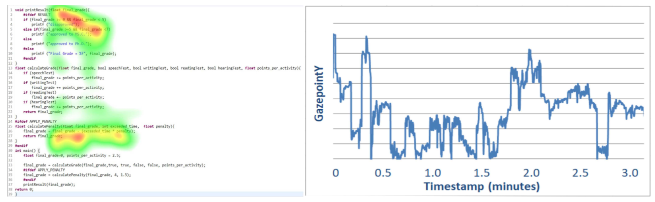
```
float final_grade = 0;
#ifdef APPLY_PENALTY
const float penalty = 1.5;
int exceeded_time = 4;
#endif
const float points_per_activity = 2.5;
int main() {
    bool speechTest = true;
    bool writingTest = true;
    bool readingTest = false;
    bool hearingTest = false;
    if (speechTest)
        final_grade += points_per_activity;
    if (writingTest)
        final_grade += points_per_activity;
    if (readingTest)
        final_grade += points_per_activity;
    if (hearingTest)
        final_grade += points_per_activity;
    #ifdef APPLY_PENALTY
    final_grade = final_grade - (exceeded_time * penalty);
    #endif
    #ifdef RESULT
    if (final_grade >= 0 && final_grade < 5)
        printf ("disapproved");
    else if(final_grade >=5 && final_grade <7)
        printf ("approved to MS.C.");
    else
        printf ("approved to Ph.D.");
    #else
        printf ("Final Grade = %f", final_grade);
    #endif
return 0;
}
```

(a) Global



```
void calculateGrade(bool speechTest, bool writingTest, bool readingTest, bool hearingTest){
    const float points_per_activity = 2.5;
    #ifdef APPLY_PENALTY
    const float penalty = 1.5;
    int exceeded_time = 4;
    #endif
    float final_grade = 0;
    if (speechTest)
        final_grade += points_per_activity;
    if (writingTest)
        final_grade += points_per_activity;
    if (readingTest)
        final_grade += points_per_activity;
    if (hearingTest)
        final_grade += points_per_activity;
    #ifdef APPLY_PENALTY
    final_grade = final_grade - (exceeded_time * penalty);
    #endif
    #ifdef RESULT
    if (final_grade >= 0 && final_grade < 5)
        printf ("disapproved");
    else if(final_grade >=5 && final_grade <7)
        printf ("approved to MS.C.");
    else
        printf ("approved to Ph.D.");
    #else
        printf ("Final Grade = %f", final_grade);
    #endif
}
int main() {
    calculateGrade(true, true, false, false);
return 0;
}
```

(b) Intraprocedural



```
void printResult(float final_grade){
    #ifdef RESULT
    if (final_grade >= 0 && final_grade < 5)
        printf ("disapproved");
    else if(final_grade >=5 && final_grade <7)
        printf ("approved to MS.C.");
    else
        printf ("approved to Ph.D.");
    #else
        printf ("Final Grade = %f", final_grade);
    #endif
}
float calculateGrade(float final_grade, bool speechTest, bool writingTest, bool readingTest, bool hearingTest, float points_per_activity){
    if (speechTest)
        final_grade += points_per_activity;
    if (writingTest)
        final_grade += points_per_activity;
    if (readingTest)
        final_grade += points_per_activity;
    if (hearingTest)
        final_grade += points_per_activity;
    return final_grade;
}
#ifdef APPLY_PENALTY
float calculatePenalty(float final_grade, int exceeded_time,  float penalty){
    final_grade = final_grade - (exceeded_time * penalty);
    return final_grade;
}
#endif
int main() {
    float final_grade=0, points_per_activity = 2.5;

    final_grade = calculateGrade(final_grade,true, true, false, false, points_per_activity);
    #ifdef APPLY_PENALTY
    final_grade = calculatePenalty(final_grade, 4, 1.5);
    #endif
    printResult(final_grade);
return 0;
}
```

(c) Interprocedural

Fig. 2. Heat map and gaze transition diagram with initial scan

of APPLY_PENALTY and the other red region is about RESULTS. In this case, however, the two regions are far from each other. Participants need, thus, to focus on each region separately. This leads to the following result.

Result 9: Interprocedural dependencies appear to increase fixation time in more distinct areas of the source code.

In the right side of Figure 2, we also show gaze transition diagrams related to each attention map. These diagrams depict the first three minutes of an "assertion error variability" debugging task performed by a participant. The diagram shows the y-coordinate that the participant was looking at as a function of time. The top of the diagram corresponds to the first line of the program and the bottom corresponds to the last line. We can observe in these diagrams that the initial scan the participant performed on the program is relatively similar for programs with global and intraprocedural dependencies. Normally, developers perform a preliminary reading of the source code in a very fast pace and then start reviewing the source code afterward. Concerning the source codes with #ifdefs, participant appears to prolong the preliminary reading in programs with global and intraprocedural dependencies. Similar behavior was reported in previous studies [38], [8].

In contrast, Fig. 2c shows that, in the program with interprocedural dependency, the participant performed a very fast preliminary reading and, then, continued with the mental simulation of the `main()` function. This reinforces our hypothesis that participants navigated through more distinct areas of the code in programs with interprocedural dependency.

D. Discussion

Here we answer our research question *How different types of feature dependency affect the comprehensibility of configurable systems?* by discussing different aspects of our findings.

#ifdefs affect comprehensibility in different degrees according with the type of feature dependency.

Our results 1, 3, 5 and 7 show that programs with global or interprocedural dependencies demand more comprehension effort for finding bugs. Developers spent more time and more number of fixations, when compared with programs with intraprocedural dependency. We hypothesize that this occurs because with both global and interprocedural dependencies, the point of definition of the dependent variable is far from its usage. Our results also show that the use of #ifdef hinders comprehensibility while debugging. Multiple researches also indicate that #ifdef might amplify maintenance problems [39], [40], [15], [2], [6]. However, our results indicate that this only happened for programs with global and interprocedural dependencies. Again, we hypothesize that this happens because the long distance between the dependent variable definition and its usages makes it difficult to simulate different configurations of enabled/disabled features.

Intraprocedural dependencies had no influence on program comprehensibility.

Our Results 2 and 6 indicate that #ifdefs may not affect the comprehensibility of programs with intraprocedural dependency. We hypothesize that this happens because the point of definition of dependent variable is closer from its usage. This result contradicts previous studies [8].

Interprocedural dependencies required more visual effort.

Results 8 and 9 show that the interprocedural dependency may increase visual effort. To find a bug, participants needed to perform more gaze transitions and focused more time on distinct parts of the source code.

Feature dependency did not affect the number of found bugs.

Result 4 revealed that the number of correctly found bugs was not affected by features dependency. This means that feature dependency may increase time and visual effort to find bugs, but do not decrease developers' ability to find bugs. This result confirms previous studies that showed that most participants correctly identify bugs in programs with #ifdef [21], [8]. However, programs with interprocedural dependency had fewer hits than other types of feature dependency.

Other findings

We also compared data in terms of variability bugs. We observed that the "null pointer dereference" bug was always difficult to find. Participants spent more time and found a lower number "null pointer dereference" bugs in both with and without #ifdef versions of the programs. Only 40% of the participants found this bug. In contrast, the "uninitialized variable" bug was always easy to find. Only 3 developer of 30 did not find this bug. These results may indicate that these bugs are not induced by variability, which contradicts previous studies [4], [5]. It is important to recall that these differences did not impact the analysis about feature dependencies as all participants analyzed programs with all types of bugs.

V. Threats to Validity

A. Internal validity

Programming language. We wrote our programs in C, because conditional compilation directives in C are native and are one of the most popular mechanisms in use. Beside, most of the studies that report variability bugs are also in C. The knowledge in C could influence our results. To minimize that, we only admitted participants with previous experience on C.

Participants' experience. We selected participants according their level of experience and distributed them into the Latin square groups. For example, each of the six PhD students went to a different group (G1 - G6) (Table I). The order of participation in the experiment determined the group. For example, the first PhD student went to group one (G1) and so on. So, we controlled confounding factors via the Latin square design and randomization.

Lab settings. All experiment trials were done in similar classrooms and with our supervision. We observed temperature and brightness conditions. In the moment of the execution of experiment, the classroom had only the participant and the authors.

B. External validity

Real bugs and features. Due to limitations we used small programs. But, our programs were inspired on concrete variability bugs found in real configurable systems. For this reason, our results may hold to other programs. However, for programs over 39 lines of code and more than two features, there may be additional effects that we have not observed.

Lab settings. Our results are also limited to the environment we adopted. A more realistic environment, with IDEs and source code with multiple files, would be ideal. However, this design would not be attractive for many participants, since it would require more time for execution. In addition, we have the limitation that the source code should fit on the screen due to the eye tracking device.

C. Construct validity

Comprehensibility measurement. Measuring comprehensibility is not trivial because it involves human factors. Therefore, it is always a threat to construct validity. To minimize this threat, we quantified comprehensibility by means of different metrics, all of them already used in previous studies.

VI. Related Work

A variety of studies focused on configurable systems based on preprocessing directives and feature dependencies. Rodrigues et al. [14] classified the types of feature dependency. They also established a set of metrics that measure the occurrence of each type of dependency. Ribeiro et al. [15] showed that feature dependency occurs in 65% of the methods of the systems they studied. Thus, feature dependency often occurs in practice. Abal et al. performed a qualitative study about 42 variability bugs collected from bug-fixing commits of the Linux kernel repository, and provided insights into the nature and occurrence of variability bugs [4].

Previous studies used biometric devices to evaluate humans' behavior while debugging source codes. Siegmund et al. [30] used images captured from a functional magnetic resonance imaging (fMRI) to identify patterns of brain activation for small comprehension tasks. Kevic et al. [41] used eye tracing device to identify the navigational behavior of the developer when performing a source code change activity.

Some studies have assessed the effect of #ifdef in maintenance tasks. Schulze et al. observed that finding and correcting errors is a time-consuming and tedious task in the presence of preprocessor annotations [9]. Melo et al. [8] used an eye-tracking device to evaluate the comprehensibility of configurable systems. However, they compared programs with and without #ifdef of only two domains. Moreover, they did not analyzed their data taking feature dependency into account. This is the main difference from our study, which explicitly analyzed in details how three types of feature dependency affect the comprehensibility of configurable systems.

VII. Conclusion and Future Work

We executed a controlled experiment with human subjects to investigate how feature dependencies affect the comprehensibility of configurable systems implemented with #ifdefs. We asked the participants to try to find different types of bugs in programs with different types of feature dependency. Then, we measured their performance in terms of spent time, number of found bugs and visual effort. Our results show that different types of feature dependency affect comprehensibility in different degrees. We observed that: (i) global and interprocedural dependencies demanded more time understand, (ii) interprocedural dependencies required more visual effort and (iii) #ifdefs did not impact the comprehensibility of programs with intraprocedural dependencies. These results lead us to hypothesize that comprehensibility is more negatively affected when a variable which is shared between features is defined in a point far from the points where it is used.

The insights obtained with our study can, in the future, support developers of configurable systems to know the parts of the source code they should take more care about. These parts would be the ones with certain characteristics (for instance, contains certain type of feature dependency) that make them more difficult to understand and, therefore, more bug prone.

As future work, we intend to further investigate the insights we obtained with the results of this study. We plan to undertake another experiment to evaluate whether the current results hold for large-scale maintenance tasks and programs with more features.

Acknowledgments

This work was supported by CNPq (grant 312153/2016-3) and FAPESB.

References

[1] P. Clements and L. Northrop, *Software product lines: practices and patterns*. Addison-Wesley Reading, 2002, vol. 3.

[2] B. J. Garvin and M. B. Cohen, "Feature interaction faults revisited: An exploratory study," in *Software Reliability Engineering (ISSRE), 2011 IEEE 22nd International Symposium on*. IEEE, 2011, pp. 90–99.

[3] T. Berger, D. Nair, R. Rublack, J. M. Atlee, K. Czarnecki, and A. Wasowski, "Three cases of feature-based variability modeling in industry," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2014, pp. 302–319.

[4] I. Abal, C. Brabrand, and A. Wasowski, "42 variability bugs in the linux kernel: a qualitative analysis," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 2014, pp. 421–432.

[5] I. Abal, J. Melo, S. Stanciulescu, C. Brabrand, M. Ribeiro, and A. Wasowski, "Variability bugs in highly configurable systems: a qualitative analysis," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 26, no. 3, p. 10, 2018.

[6] M. D. Ernst, G. J. Badros, and D. Notkin, "An empirical analysis of c preprocessor use," *IEEE Transactions on Software Engineering*, vol. 28, no. 12, pp. 1146–1170, 2002.

[7] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze, "An analysis of the variability in forty preprocessor-based software product lines," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 105–114.

[8] J. Melo, F. B. Narcizo, D. W. Hansen, C. Brabrand, and A. Wasowski, "Variability through the eyes of the programmer," in *Program Comprehension (ICPC), 2017 IEEE/ACM 25th International Conference on*. IEEE, 2017, pp. 34–44.

[9] S. Schulze, J. Liebig, J. Siegmund, and S. Apel, "Does the discipline of preprocessor annotations matter?: a controlled experiment," in *ACM SIGPLAN Notices*, vol. 49, no. 3. ACM, 2013, pp. 65–74.

[10] H. Spencer and G. Collyer, "# ifdef considered harmful, or portability experience with c news," *Usenix Summer 1992 Technical Conf.*, pp. 185–197, 1992.

[11] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger, "Variability-aware parsing in the presence of lexical macros and conditional compilation," in *ACM SIGPLAN Notices*, vol. 46, no. 10. ACM, 2011, pp. 805–824.

[12] F. Medeiros, M. Ribeiro, and R. Gheyi, "Investigating preprocessor-based syntax errors," in *ACM SIGPLAN Notices*, vol. 49, no. 3. ACM, 2013, pp. 75–84.

[13] R. Tartler, C. Dietrich, J. Sincero, W. Schröder-Preikschat, and D. Lohmann, "Static analysis of variability in system software: The 90, 000# ifdefs issue." in *USENIX Annual Technical Conference*, 2014, pp. 421–432.

[14] I. Rodrigues, M. Ribeiro, F. Medeiros, P. Borba, B. Fonseca, and R. Gheyi, "Assessing fine-grained feature dependencies," *Information and Software Technology*, vol. 78, pp. 27–52, 2016.

[15] M. Ribeiro, F. Queiroz, P. Borba, T. Tolêdo, C. Brabrand, and S. Soares, "On the impact of feature dependencies when maintaining preprocessor-based software product lines," *ACM SIGPLAN Notices*, vol. 47, no. 3, pp. 23–32, 2012.

[16] B. Sharif, M. Falcone, and J. I. Maletic, "An eye-tracking study on the role of scan time in finding source code defects," in *Proceedings of the Symposium on Eye Tracking Research and Applications*. ACM, 2012, pp. 381–384.

[17] J. Siegmund, "Program comprehension: Past, present, and future," in *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, vol. 5. IEEE, 2016, pp. 13–20.

[18] G. E. Box, J. S. Hunter, and W. G. Hunter, *Statistics for experimenters: design, innovation, and discovery*. Wiley-Interscience New York, 2005, vol. 2.

[19] R. A. Bailey, *Design of comparative experiments*. Cambridge University Press, 2008, vol. 25.

[20] M. Ribeiro, P. Borba, and C. Kästner, "Feature maintenance with emergent interfaces," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 989–1000.

[21] J. Melo, C. Brabrand, and A. Wasowski, "How does the degree of variability affect bug finding?" in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 679–690.

[22] M. Lanza and R. Marinescu, *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media, 2007.

[23] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[24] M. Hristova, A. Misra, M. Rutter, and R. Mercuri, "Identifying and correcting java programming errors for introductory computer science students," in *ACM SIGCSE Bulletin*, vol. 35, no. 1. ACM, 2003, pp. 153–156.

[25] N. Singh, C. Gibbs, and Y. Coady, "C-clr: a tool for navigating highly configurable system software," in *Proceedings of the 6th workshop on Aspects, components, and patterns for infrastructure software*. ACM, 2007, p. 9.

[26] A. Voßkühler, V. Nordmeier, L. Kuchinke, and A. M. Jacobs, "Ogama (open gaze and mouse analyzer): open-source software designed to analyze eye and mouse movements in slideshow study designs," *Behavior research methods*, vol. 40, no. 4, pp. 1150–1162, 2008.

[27] B. Sharif, G. Jetty, J. Aponte, and E. Parra, "An empirical study assessing the effect of seeit 3d on comprehension," in *Software Visualization (VISSOFT), 2013 First IEEE Working Conference on*. IEEE, 2013, pp. 1–10.

[28] G. Camilli and K. D. Hopkins, "Applicability of chi-square to 2× 2 contingency tables with small expected cell frequencies." *Psychological Bulletin*, vol. 85, no. 1, p. 163, 1978.

[29] K. Rayner, "Eye movements and attention in reading, scene perception, and visual search," *The quarterly journal of experimental psychology*, vol. 62, no. 8, pp. 1457–1506, 2009.

[30] J. Siegmund, C. Kästner, S. Apel, C. Parnin, A. Bethmann, T. Leich, G. Saake, and A. Brechmann, "Understanding understanding source code with functional magnetic resonance imaging," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 378–389.

[31] J. H. Goldberg and X. P. Kotval, "Eye movement-based evaluation of the computer interface," *Advances in occupational ergonomics and safety*, pp. 529–532, 1998.

[32] P. A. Kolers, R. L. Duchnicky, and D. C. Ferguson, "Eye movement measurement of readability of crt displays," *Human Factors*, vol. 23, no. 5, pp. 517–527, 1981.

[33] R. J. Jacob and K. S. Karn, "Eye tracking in human-computer interaction and usability research: Ready to deliver the promises," in *The mind's eye*. Elsevier, 2003, pp. 573–605.

[34] K. Rayner, K. H. Chace, T. J. Slattery, and J. Ashby, "Eye movements as reflections of comprehension processes in reading," *Scientific studies of reading*, vol. 10, no. 3, pp. 241–255, 2006.

[35] A. T. Duchowski, "Eye tracking methodology," *Theory and practice*, vol. 328, 2007.

[36] K. Rayner, "Eye movements in reading and information processing: 20 years of research." *Psychological bulletin*, vol. 124, no. 3, p. 372, 1998.

[37] O. Špakov and D. Miniotas, "Visualization of eye gaze data using heat maps," *Elektronika ir elektrotechnika*, vol. 74, no. 2, pp. 55–58, 2007.

[38] H. Uwano, M. Nakamura, A. Monden, and K.-i. Matsumoto, "Analyzing individual performance of source code review using reviewers' eye movement," in *Proceedings of the 2006 symposium on Eye tracking research & applications*. ACM, 2006, pp. 133–140.

[39] F. Medeiros, C. Kästner, M. Ribeiro, S. Nadi, and R. Gheyi, "The love/hate relationship with the c preprocessor: An interview study," in *LIPIcs-Leibniz International Proceedings in Informatics*, vol. 37. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[40] W. Fenske, S. Schulze, and G. Saake, "How preprocessor annotations (do not) affect maintainability: a case study on change-proneness," in *ACM SIGPLAN Notices*, vol. 52, no. 12. ACM, 2017, pp. 77–90.

[41] K. Kevic, B. M. Walters, T. R. Shaffer, B. Sharif, D. C. Shepherd, and T. Fritz, "Tracing software developers' eyes and interactions for change tasks," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 202–213.