Feature Maintenance with Emergent Interfaces

Márcio Ribeiro
Federal University of Alagoas
marcio@ic.ufal.br

Paulo Borba
Federal University of Pernambuco
phmb@cin.ufpe.br

Christian Kästner
Carnegie Mellon University

Abstract

Hidden code dependencies are responsible for many complications in maintenance tasks. With the introduction of variable features in product lines, dependencies may even cross feature boundaries and related problems are prone to be detected late. Many current implementation techniques for product lines lack proper interfaces, which could make such dependencies explicit. As alternative to changing the implementation approach, we introduce a tool-based solution to support developers in recognizing and dealing with feature dependencies: emergent interfaces. Emergent interfaces are computed on demand, based on feature-sensitive *interprocedural* dataflow analysis. They emerge in the IDE and emulate benefits of modularity not available in the host language. In a replicated controlled experiment, we found that emergent interfaces can improve performance of maintenance tasks by up to 3 times while also reducing the number of errors.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms term1, term2

Keywords keyword1, keyword2

1. Introduction

Developers often introduce errors to software systems when they fail to recognize module and feature dependencies [9]. This problem is particularly critical for variable systems, in which features can be enabled and disabled at compile time or run time, and market and technical needs constrain how features can be combined. In this context, features often crosscut each other [22] and share program elements like variables and methods [26], without proper modular support from a notion of interface between features. So developers could easily miss code level feature dependencies, such as when a feature assigns a value to a variable read by another feature. As there is no mutual agreement between separate feature developers, changing the assigned value might be the correct action for maintaining one feature, but might bring undesirable consequences to the behavior of the other feature. Similar issues could also appear when developers assume invalid dependencies, as would be the case if the just discussed features were mutually exclusive. In a prior study collecting metrics of 43 preprocessor-based product-line implementations, we found that feature dependencies are frequent in practice [26].

To reduce the feature dependency problem, we propose a technique called *emergent interfaces* that establishes, on demand and according to a given maintenance task, interfaces to feature code (introduced previously in a vision paper [25]). We call our technique emergent interfaces because, instead of writing interfaces manually, developers can request interfaces on demand; that is, interfaces emerge to give support for specific maintenance tasks. To do so, emergent interfaces capture dependencies between the feature we are maintaining and the others we might impact by performing the corresponding maintenance task. This way, developers become aware of the dependencies, and may have better chance of not introducing errors [35]. Emergent interfaces may also help to reduce maintenance effort. In fact, instead of searching for feature dependencies throughout the code, and reasoning about requirements level feature constraints, developers can rely on proper tool support that computes interfaces using feature-sensitive analyses [7].

To evaluate the potential of emergent interfaces to reduce errors and development effort, and better understand to what extent they support feature modularity, we conducted and replicated a controlled experiment. We focus on the study of feature maintenance tasks in software product lines implemented with preprocessor directives, which are widely used to implement compile time variability in industrial practice. Also, to aid comprehensibility, we consider tasks performed with support for virtual separation of concerns (VSoC) [16], which allows developers to hide code fragments not related to features associated to a given task. This way we allow developers to focus on a feature and its (emergent) interface without the distraction brought by other features.

[Copyright notice will appear here once 'preprint' option is removed.]

In particular, we evaluate emergent interfaces by answering two research questions: *Do emergent interfaces reduce maintenance effort for tasks involving feature code dependencies in preprocessor-based systems?*; and *Do emergent interfaces reduce the number of errors introduced by maintenance tasks involving feature code dependencies in preprocessor-based systems?* We consider tasks that involve both *intraprocedural* and *interprocedural* feature dependencies from two product lines. We first conduct the experiment in one institution, using graduate students as subjects, and then replicate it in another institution, this time using undergraduate students. In both cases, around half of the students have professional experience.

Our experiment reveals that, considering the selected kinds of system and developers, emergent interfaces help to reduce maintenance effort for tasks involving *interprocedural* dependencies, which cross method boundaries. Both experiment rounds reveal that developers were, on average, 3 times faster when using emergent interfaces. As for tasks involving only *intraprocedural* dependencies, we confirm statistical significance in only one round, in which we on average observe a 1.6 fold improvement in favor of emergent interfaces. In line with recent research [35], in both rounds we observe that presenting feature dependencies help developers to detect and avoid errors, regardless of the kinds of dependencies.

In summary, we make the following contributions: (1) We revise the emergent feature modularization concept from the previous vision paper [25] and extend and evaluate tool support from *intraprocedural* analysis to the much more powerful *interprocedural* analysis; (2) We evaluate effort and error reduction when using emergent interfaces in a controlled (and replicated) experiment with in total 24 participants in two product lines and demonstrate significant potential. We complement our previous evaluation results [26], which focused only on *intraprocedural* dependencies, measured only effort, and used proxy metrics without proper tool support for emergent interfaces.

REVIEWER: You really need to make the difference compared to the older papers clearer in the introduction. You talk about intra versus inter procedural, and about previous evaluation results, but when looking at the previous papers, the best way to describe the difference is that this paper presents a comprehensive evaluation. That's what it does and does quite well.

Main contribution: evaluation.

We structure the remainder of the paper as follows. Section 2 presents motivating examples for the problem caused by feature dependencies. To reduce this problem, we show emergent interfaces in Section 3. Then, we present the experimental design of our evaluation in Section 4 and the results in Section 5. Finally, we present related work and concluding remarks.

2. Maintaining preprocessor-based product lines

To better explain the mentioned issues due to the lack of feature modularity, we outline two concrete scenarios. Although we could illustrate these issues in a number of significantly different contexts, we choose a critical one: variable systems, especially in the form of software product lines, where code fragments are configurable and may not be included in all product configurations. As a result, by selecting valid feature combinations, we can generate many different products. Industrial product lines can easily have hundreds of features with a large number of possible derivable products. When maintaining a product line, the developer must be sure not to break any of the possible products, but due to the sheer number of them, rapid feedback by testing is not possible for all products. Furthermore, code dependencies may cross feature boundaries, so that a variable changed in one feature is read by another feature. A developer must make sure that all of those dependencies are considered as well.

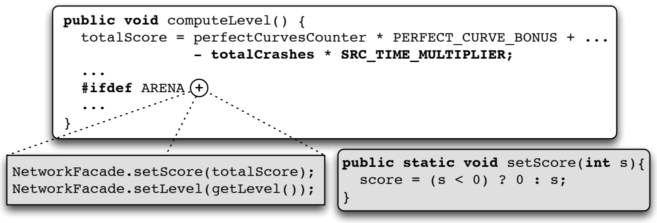
The problem we outline is shared by many implementation approaches for product lines that support some form of crosscutting. For example, if we use aspects (or similar techniques) to implement crosscutting features [12], and configure the product line by selecting which aspects to weave, we need to consider potential dependencies between (optional) aspects and classes. In practice, a more common scenario, and the one we focus here, is to use conditional-compilation (`#ifdef` constructs) with a preprocessor, where optional code fragments are merely annotated in a common code base [22]. Also in this context, dependencies cross method and feature boundaries and may only occur in specific configurations.

To deal with the scattering of feature code in preprocessor-based implementations, researchers have investigated virtual forms of separating concerns by helping developers to focus on relevant features. For example, in CIDE [16], developers can create views on a specific feature selection, hiding irrelevant files and irrelevant code fragments inside files, with standard code-folding techniques at the IDE level. Code fragments are hidden if they do not belong to the selected feature set the developer has selected as relevant for a task. While virtual separation with hiding can emulate some form of modularity, it also potentially hides code fragments that might be relevant for maintenance tasks. Similar to separating features with aspects, a view shows only the code of one feature (or a set of features) while dependencies may point to code in other aspects or hidden code fragments (visible only in other views). Developers are then prone to make errors and need to invest effort to trace dependencies, as we illustrate next with two scenarios.

2.1 Scenario 1: Implementing a new requirement

The first scenario comes from the *Best Lap* commercial car racing game[1] that motivate players to achieve the best circuit lap time and therefore qualify for the pole position. Due to portability constraints, the game is developed as product line and is deployed on 65 different devices [2].

To compute the score, developers implemented the method illustrated in Figure 1: variable `totalScore` stores the player's total score. Next to the common code base, the method contains optional code that belongs to feature *ARENA*. This feature publishes high scores on a network server and, due to resource constraints, is not available in all product configurations.



```
public void computeLevel() {
  totalScore = perfectCurvesCounter * PERFECT_CURVE_BONUS + ...
               - totalCrashes * SRC_TIME_MULTIPLIER;
  ...
  #ifdef ARENA
  ...
}
```

```
NetworkFacade.setScore(totalScore);
NetworkFacade.setLevel(getLevel());
```

```
public static void setScore(int s){
  score = (s < 0) ? 0 : s;
}
```

Figure 1. Maintenance only works for some products.

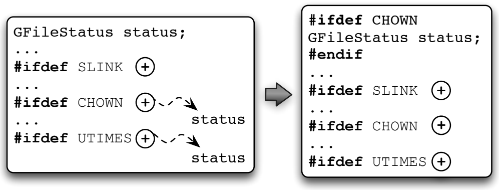
To add penalties in case the player often crashes the car, suppose now that a developer has to implement the following new requirement: let the game score be not only positive, but also negative. To accomplish the task, she localizes the *maintenance points*, in this case just the `totalScore` assignment, and changes its value (see the bold line in Figure 1). When executing products that do not have the *ARENA* feature, we can then observe the desired behavior. But, due to the *ARENA* associated `setScore` method, users using a configuration that includes that feature may report that the game does not correctly submit negative scores to the network server.

The cause of the problem is that the *ARENA* implementation extends the behavior and is therefore affected by the change as well. This was not, however, noticed by the developer, who did not realize that she had to change code associated to other features. In this case, she would have to change part of the *ARENA* code to not check the invariant that all scores are positive. In the actual implementation, feature *ARENA* is partially implemented inside method `computeLevel` and guarded with `#ifdef` directives, so it might not be so difficult to notice the dependence if the method is not so big. However, we could alternatively implement it in a separate aspect, or keep it hidden through the use of virtual separation. In both cases, for maintenance tasks that do not obviously relate to feature *ARENA*, the developer could easily miss the dependency.

In this context, searching for code level feature dependencies might increase developers effort since they have to make sure that the modification does not impact other features. Further, if they miss a dependency, they can easily introduce errors by not properly completing the maintenance task, for example.

2.2 Scenario 2: Fixing an unused variable

Our second scenario is based on a bug report from the *glibc*[2] project. This project is structured with several preprocessor macros and conditional-compilation constructs. Developers report that a variable `status` in core functionality is unused. Investigating the problem, we find that the variable is declared in all configurations, but only used when features *CHOWN* or *UTIMES* are selected, as illustrated in Figure 2 (left-hand side). When we compile the product line without either feature, the compiler issues an unused-variable warning.



```
GFileStatus status;
...
#ifdef SLINK
...
#ifdef CHOWN
...                  status
#ifdef UTIMES
                     status
```

```
#ifdef CHOWN
GFileStatus status;
#endif
...
#ifdef SLINK
...
#ifdef CHOWN
...
#ifdef UTIMES
```

Figure 2. Wrong fixing of an unused variable.

To fix the bug report, a developer would typically look for uses of the variable. If she does not carefully look beyond cross feature boundaries (potentially regarding physically separated aspects or code fragments hidden as part of virtual separation of concerns), considering also requirements level feature constraints, she can easily introduce an error.

She could, for example, only detect the variable usage in feature *CHOWN*, and then guard the declaration correspondingly as shown in the right-hand side of Figure 2. This would actually lead to a worse problem: an undeclared variable compilation error for configurations with *UTIMES* but without *CHOWN*. The correct fix would require to guard the declaration with `#ifdef (CHOWN || UTIMES)`.

Again, the initial problem and the incorrect fix are caused by the difficulty to follow dependencies across feature boundaries. This applies not only to preprocessor-based systems and virtual separation environments, but also to systems and product lines structured with other mechanisms. So, in the presence of crosscutting features, identifying feature dependencies may require substantial effort, and missing such dependencies can easily lead to programming errors.

3. Emergent Interfaces

The problems discussed so far occur essentially because features *share* elements such as variables and methods without clear interfaces. Whenever we have such sharing, we say that there is a code level *feature dependency* between the involved

[1] Developed by Meantime Mobile Creations. `http://www.meantime.com.br/`

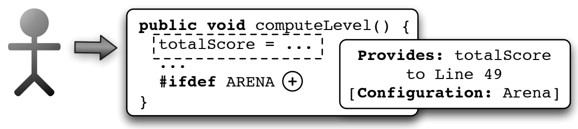
[2] `http://www.gnu.org/s/libc/`

features [26]. For instance, a mandatory feature might declare a variable subsequently used by optional features (see `totalScore` and `status` in Figures 1 and 2, respectively).

To help developers avoid problems related to feature dependencies, we propose a technique called emergent interfaces [25] that establishes, on demand and according to a given maintenance task, interfaces to feature code. When developers are interested in dependencies from a specific code block, they can ask the tool that implements the technique to compute specific interfaces, pointing out feature dependencies. The interfaces emerge on demand, giving support for developers to maintain one feature without breaking others.

REVIEWER: What are "specific interfaces"?

To illustrate how emergent interfaces work, we focus on preprocessor-based implementations combined with views of virtual separation. It allows us to hide irrelevant code fragments but, due to our interfaces, we are still warned about their dependencies. Now, consider *Scenario 1* of Section 2.1, where the developer is supposed to change how the total score is computed. The first step when using our approach consists of selecting the maintenance points. She selects the `totalScore` assignment (see the dashed rectangle in Figure 3) and then our tool analyzes the code to capture dependencies between the feature she is maintaining and the others. Finally, the interface emerges as shown in Figure 3 (right-hand side).



```
public void computeLevel() {
  totalScore = ...
  ...
  #ifdef ARENA (+)
}
```

```
Provides: totalScore
  to Line 49
[Configuration: Arena]
```

Figure 3. Emergent interface for Scenario 1.

The emerged interface states that the maintenance task may impact the behavior of products containing the *ARENA* feature. So, the core functionality provides `totalScore` current's value whereas *ARENA* requires it. The developer is now aware of the dependency. When investigating it, she is likely to discover that she also needs to modify *ARENA* code to avoid introducing an error.

Notice we do *not* provide any support to find the maintenance points. This task is up to the developer according to her knowledge about the source code. After finding these points, she takes them as input to our approach. Then, we provide information about the potential features she might impact.

We implemented the concept of emergent interfaces in an Eclipse-based prototype tool named Emergo. Emergo computes emergent interfaces based on feature dependencies between methods or within a single method, by using *interprocedural* or *intraprocedural* feature-sensitive dataflow analysis [5, 7]. This means we can take only valid feature combinations into account, preventing developers from reasoning about feature constraints and even from assuming invalid dependencies in case of mutually exclusive features (which may cause potential errors). The feature-sensitive approach is capable of analyzing all configurations of a product line without having to generate all of them explicitly. This increases performance [5, 7], which is important for interactive tools like ours that need to provide quick responses to developers requests. Emergo implements a sort of forward static slicing [33] based on data using *interprocedural* and *intraprocedural* feature-sensitive reaching definition analysis.

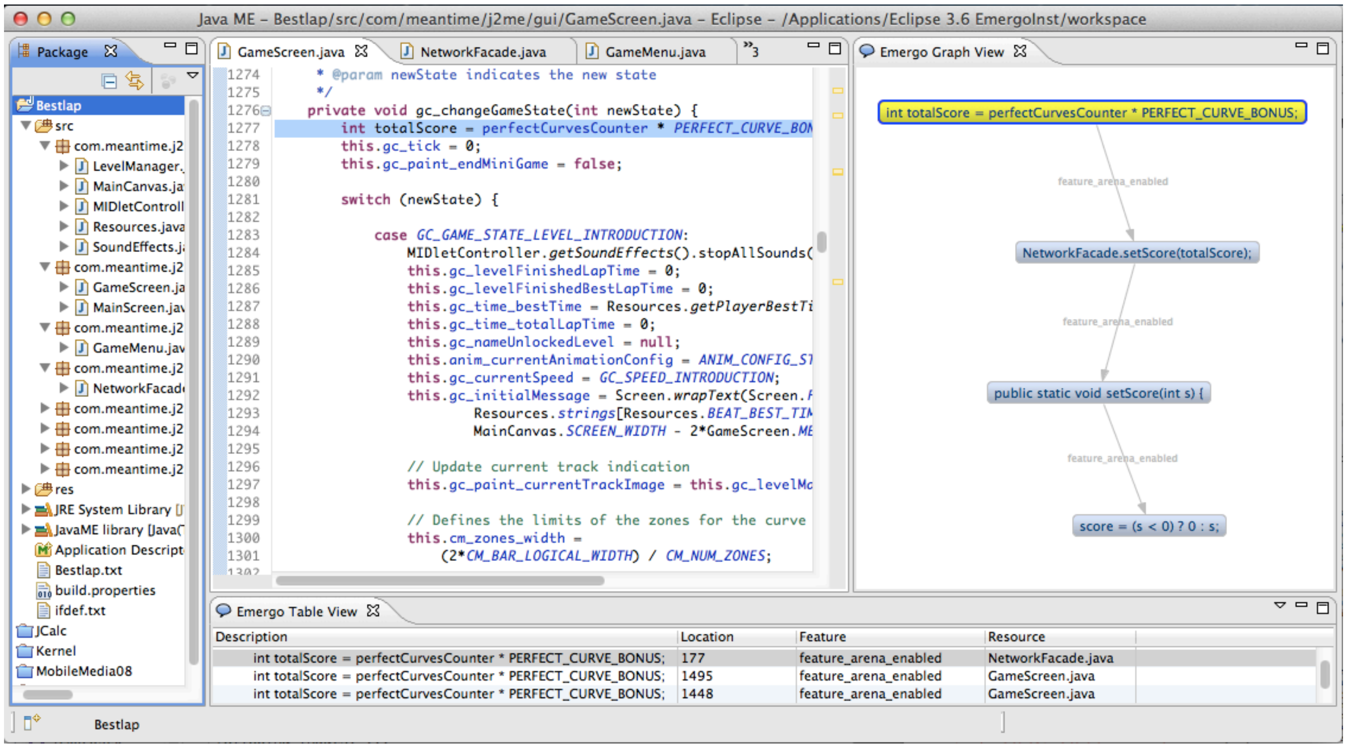
Figure 4 illustrates a screen shot of Emergo. As mentioned, we do not provide support for the developer to find the maintenance points. So, after finding and selecting the maintenance point—see Figure 4, line 1277, where she selects the `totalScore` assignment—she invokes the tool for the interfaces. Emergo shows emergent interfaces using a table view and a graph view. To better understand the results pointed by Emergo, we now focus on the table. The "Description" column illustrates the maintenance point. Since there is no `#ifdef` statement encompassing the maintenance point, we associate it with the mandatory feature. We show the potentially impacted features in the "Feature" column, here the *ARENA* feature. The table view also shows the exact lines of code that contain uses of `totalScore` as well as their respective java files (see the "Resource" column). In summary, we read the first line of the table as follows: if she changes the `totalScore` assignment belonging to the mandatory feature, she can potentially impact the *ARENA* feature in line 177 of the `NetworkFacade` class.

Initially, Emergo shows all dependencies in both views. This way, depending on the project, it points lots of dependencies, which might be difficult to read and understand them. Thus, to focus on a particular dependency, she clicks on the corresponding table line and Emergo automatically removes unrelated dependencies of the graph. This means we show only the path associated with the dependency of that table line. For example, according to Figure 4, she selected the first line of the table. The graph now contains only the path from the maintenance point to the line 177 of the `NetworkFacade` class.

Developers can quickly reach the impacted features by clicking either on table lines or on the graph nodes. For example, if she clicks on the node "`score = (s < 0) ? 0 : s;`" of the graph, she reaches the code that checks the invariant that all scores are positive. Now, she is aware of the dependency. So, to accomplish the task of allowing negative scores she knows she also needs to remove the check.

The current version of Emergo can compute emergent interfaces based on the following inputs: ...

Emergo can also help on preventing developers from analyzing unnecessary features and their associated code, which is important to decrease maintenance effort. We believe that Emergo can help on making the idea of virtual separation of concerns realistic. Thus, we can hide features and rely on Emergo to only open the ones we really need to. For instance, consider *Scenario 2* (Section 2.2). Emergo would focus on



```
         * @param newState indicates the new state
         */
        private void gc_changeGameState(int newState) {
            int totalScore = perfectCurvesCounter * PERFECT_CURVE_BON
            this.gc_tick = 0;
            this.gc_paint_endMiniGame = false;

            switch (newState) {

                case GC_GAME_STATE_LEVEL_INTRODUCTION:
                    MIDletController.getSoundEffects().stopAllSounds(
                    this.gc_levelFinishedLapTime = 0;
                    this.gc_levelFinishedBestLapTime = 0;
                    this.gc_time_bestTime = Resources.getPlayerBestTi
                    this.gc_time_totalLapTime = 0;
                    this.gc_nameUnlockedLevel = null;
                    this.anim_currentAnimationConfig = ANIM_CONFIG_ST
                    this.gc_currentSpeed = GC_SPEED_INTRODUCTION;
                    this.gc_initialMessage = Screen.wrapText(Screen.F
                            Resources.strings[Resources.BEAT_BEST_TIM
                            MainCanvas.SCREEN_WIDTH - 2*GameScreen.ME

                    // Update current track indication
                    this.gc_paint_currentTrackImage = this.gc_levelMo

                    // Defines the limits of the zones for the curve
                    this.cm_zones_width =
                        (2*CM_BAR_LOGICAL_WIDTH) / CM_NUM_ZONES;
```

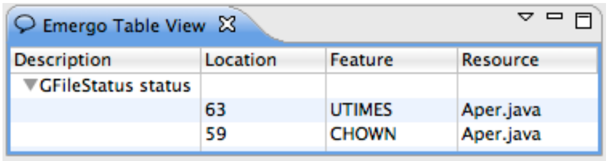
```
int totalScore = perfectCurvesCounter * PERFECT_CURVE_BONUS;
NetworkFacade.setScore(totalScore);
public static void setScore(int s) {
score = (s < 0) ? 0 : s;
```

Description	Location	Feature	Resource
int totalScore = perfectCurvesCounter * PERFECT_CURVE_BONUS;	177	feature_arena_enabled	NetworkFacade.java
int totalScore = perfectCurvesCounter * PERFECT_CURVE_BONUS;	1495	feature_arena_enabled	GameScreen.java
int totalScore = perfectCurvesCounter * PERFECT_CURVE_BONUS;	1448	feature_arena_enabled	GameScreen.java

Figure 4. Emergent interface for Scenario 1 using Emergo.

CHOWN and *UTIMES* (see Figure 5). So, we could keep *SLINK* hidden, since it is not related to the current task.

REVIEWER: Figure 4 also made me question what the inputs and outputs are. How is the impacted feature shown? What is the feature in this example?



Description	Location	Feature	Resource
GFileStatus status			
	63	UTIMES	Aper.java
	59	CHOWN	Aper.java

Figure 5. Emergo view illustrating the impacted features. Lines 63 and 59 contain uses of `status`.

REVIEWER: Rev1: That said, section III felt too short, especially if you have extended the tool. Some words on that and how it works would help.

REVIEWER: Rev2: With regards to the technique of expressing and computing emergent interfaces, it is unclear what the input to the technique is. Is it a specific variable? A variable at a particular point in the code? An entire code block? A code block that spans more than one method? I would like to see much more clarity on the inputs to the mehod than provided by:

Improve Emergo explanation. Better explain the analyses and provide a sort of “BNF” of the statements that Emergo takes into account.

4. Experimental design

In the previous section we suggest that emergent interfaces can make feature maintenance tasks (such as *Scenario 1* and *Scenario 2*) faster and less error prone. To evaluate these hypotheses and to get a better understanding of the benefits and drawbacks of emergent interfaces, we conducted and replicated a controlled experiment. We specifically investigate and compare maintenance effort and introduced errors when maintaining preprocessor-based product lines with and without emergent interfaces in a setting that supports virtual separation, allowing developers to hide feature code fragments.

4.1 Goal, Questions, and Metrics

Our evaluation aims to compare maintenance of preprocessor-based product lines with and without emergent interfaces (these are our treatments). Specifically, we are interested in the interaction with the feature hiding facilities of virtual separation of concerns, which we enable in both cases to aid comprehensibility. We evaluate emergent interfaces from the developer's point of view and observe effort and number of errors they commit. We investigate the following questions:

- Question 1: *Do emergent interfaces reduce effort during maintenance tasks involving feature code dependencies in preprocessor-based systems?*

- Question 2: *Do emergent interfaces reduce the number of errors during maintenance tasks involving feature code dependencies in preprocessor-based systems?*

To answer Question 1, we measure the time required to complete a maintenance task. To better explain this metric, consider *Scenario 1* presented in Figure 1, where the developer should also take negative scores into account: time counts the amount of time needed to find feature dependencies (in our example, the use of `totalScore` in *ARENA*) and to change the impacted features to accomplish the task. To avoid bias, we do not count the time to find the maintenance points, as we shall see.

REVIEWER: First, regarding the "to avoid bias, we do not count the time to find the maintenance points...". This sounds very odd, given that the purpose of the tool is to do exactly that? And how do you subtract the time to find this starting point in the regular Eclipse case? This should all be better explained in terms of impact on the results.

Unfortunately, the reviewer did not understand how Emergent Interfaces works. Emergo does not compute maintenance points: there is no support to do it. We should improve Section 3 to make sure that we support developers *after* finding the maintenance point.

To answer Question 2, we measure how many incorrect solutions the developer committed during a maintenance task (metric number of errors, or short NE). We consider error as a human action that introduces one or more defects to the code. As we describe below, we automatically check some submitted solutions during the experiment, so the participant can try again on an incorrect solution. On the other hand, there are also submissions that are manually checked.

4.2 Participants

We performed the experiment in three rounds. In a first pilot study, we tested the experimental design with a small group of six graduate students at the University of Marburg, Germany. Next, we performed the actual experiment with 10 graduate students at Federal University of Pernambuco, Brazil (*Round 1*). Finally, we replicated the experiment with 14 undergraduate students at Federal University of Alagoas, Brazil (*Round 2*). In both cases, around half of the participants had professional experience—varying from few months to many years of experience—and were actually part-time students. The 10 graduate students are attendants of a course on experimental software engineering lead by an independent lecturer. The 14 are UROP[3] students and volunteered to participate. All participants took part voluntarily and were informed they could stop participating at any time, but nobody did.

[3] Undergraduate Research Opportunity Program

4.3 Experimental Material and Tasks

We use two preprocessor-based product lines as experimental material: *Best Lap* and *MobileMedia*. The former is a highly variant commercial product line that has approximately 15 KLOC. The latter is a product line for applications with about 3 KLOC that manipulate photo, music, and video on mobile devices [12]. It contains feature restrictions and has been used in previous research studies [12, 24].

We ask participants to perform a number of maintenance tasks in each of the product lines. Therefore, we provide the product line's source code and corresponding tasks that the participants should perform by modifying the source code.

To cover different use cases, we prepare two kinds of tasks. In line with our motivating scenarios in Section 2, we distinguish between tasks where participants should implement a new requirement (requiring *interprocedural* analysis of the existing source code) and tasks where participants should fix an unused variable (requiring only *intraprocedural* analysis). We provide a task of each kind for each product line, for a total of four distinct tasks, as discussed in the following.

Task 1 - New requirement for Best Lap The new requirement for *Best Lap* is similar to our motivating *Scenario 1*, but differently from it, there are two methods of feature *ARENA* that contain conditional statements forbidding negative scores. So, to accomplish the task, besides changing the `totalScore` assignment, participants should remove or rewrite these conditional statements (see one of them in method `setScore` of Figure 1). To reach them, participants need to consider *interprocedural* dependencies. That is, there are dependencies from the maintenance point to two conditional statements, each one in a different method.

In case Emergo is available, the participant should use it to identify the feature dependencies between the variable `totalScore` and the rest of the code. Otherwise, the participant is free to use standard tools such as find/replace and highlighting. This also holds for the subsequent tasks.

Task 2 - New requirement for MobileMedia The task for *MobileMedia* is conceptually similar in the sense that participants should change a variable assignment, follow feature dependencies, and update conditional statements (here, only one `if` statement). However, differently from Task 1, where the method call depth to reach the two conditional statements is 1, here the call depth is 2. That is, from the maintenance point, we need to follow two method calls to reach the `if` statement.

Task 3 - Unused variable in Best Lap In *Best Lap*, we asked participants to fix the unused-variable warnings for two variables: `tires` and `xP`. Such warnings are commonly found in many bug reports of preprocessor-based systems.[4]

[4] See `https://bugzilla.gnome.org/show_bug.cgi?id=461011`, `https://bugzilla.gnome.org/show_bug.cgi?id=167715`, `https://bugzilla.gnome.org/show_bug.cgi?id=401580`, `https://bugzilla.kernel.org/show_bug.cgi?id=1664`

We introduced the bugs ourself by removing correct `#ifdef` directives around the variable declarations. We can solve all unused-variable tasks by following *intraprocedural* dependencies only, but they typically require investigating code of different features. To accomplish the tasks, we ask participants to put a correct `#ifdef` around the variable declarations. The variables `tires` and `xP` are inside methods with 147 and 216 source lines of code, respectively.

Task 4 - Unused variable in MobileMedia Again the *MobileMedia* task is conceptually similar to the *Best Lap* task. Participants should fix the unused-variable warning of `numberOfViews` and `albumMusic`. The two variables are placed in shorter methods when compared to Task 3: they have 49 and 71 lines of code. The longest method here has less than half of the lines of the shortest one in Task 3.

The Emergo version we consider to compute dependencies for these tasks uses *def-use* chains based on the reaching definitions analysis. Thus, it points *intraprocedural* and *interprocedural* dependencies when maintaining a definition used by another feature.

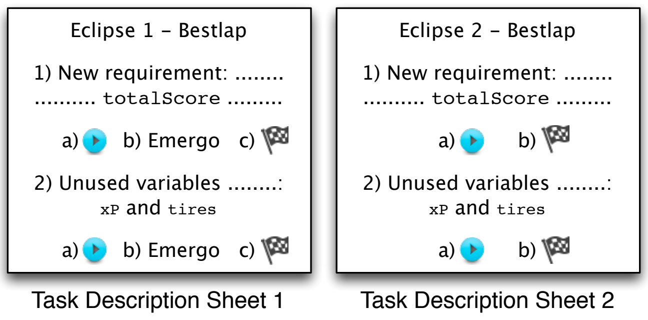
Overall, the tasks for both product lines have a roughly comparable level of difficulty, but they are not equivalent. Actually, these differences between task for both product lines help us to better analyze the effects of our two treatments and is properly controlled by our experiment design, as we shall see in Section 4.5. To illustrate the tasks from the participant's perspective, we summarize in Figure 6 two task description sheets we distributed ("a", "b", and "c" represent the steps that participants should follow). There are two more sheets related to *MobileMedia*.

REVIEWER: On what basis do you believe they are comparable? Were they of comparable difficulty to the participants in the study?

REVIEWER: Second, I do not mind that the tasks used were simple in nature, however, I do think the type of tasks (the types of changes that were required) were extremely skewed to show the benefits of the tool. It is not realistic to evaluate a tool that highlights hidden dependencies using a scenario where the variable that has the hidden dependencies has been pre-selected for the developers. I would contend that these tasks are by no means representative of real tasks, even small ones.

There is no support for finding the maintenance point, so we focus on the implementation part of a maintenance task (code change task). Given the maintenance point, what is the impact to change the code? Where should I change?

Finally, we designed warmup tasks on a toy product line so that participants could learn how to generate emergent interfaces using Emergo at the start of the experiment. We perform warmup tasks together in a class context and are not evaluated.


Eclipse 1 – Bestlap

1) New requirement: ....
.... `totalScore` ....

a) ▶ b) Emergo c) 🏁

2) Unused variables ....:
`xP` and `tires`

a) ▶ b) Emergo c) 🏁

Task Description Sheet 1

Eclipse 2 – Bestlap

1) New requirement: ....
.... `totalScore` ....

a) ▶ b) 🏁

2) Unused variables ....:
`xP` and `tires`

a) ▶ b) 🏁

Task Description Sheet 2

Figure 6. Task description sheets related to the *Best Lap* product line.

4.4 Hypotheses

Based on our goals and tasks, we evaluate the following hypotheses:

- *H1 - Effort:* With emergent interfaces, developers spend less time to complete the maintenance tasks in both product lines.
- *H2 - Error introduction:* With emergent interfaces, developers commit less errors in both product lines when performing a maintenance task involving feature dependencies.

4.5 Design

To evaluate our research questions, we distinguish between participants using our treatments (independent variable with two levels: with and without emergent interfaces). Additionally, we distinguish between the tasks of both product lines, as we cannot assume equivalence (independent variable with two levels: *Best Lap* and *MobileMedia* tasks). We measure time and the number of errors (dependent variables) for new-requirement tasks and unused-variable tasks.

Since we have two independent variables with two levels each, we use a standard *Latin Square design* [6]. We randomly distribute participants in rows and product lines tasks in columns. The treatments come inside each cell. Each treatment appears only once in every row and every column. As a result, each participant performs both kinds of tasks on each product line and also uses and not uses emergent interfaces at some part of the experiment, but no participant will perform the same task twice, which avoids corresponding carry-over effects, such as learning (if we let one to use both treatments in the same task, we would favor the second treatment, since she already knows how to accomplish the task). The design does not favor any treatment and blocks two factors: participant and maintenance tasks. We illustrate the design in Figure 7.

As analysis procedure for this design, we perform an analysis of variance (ANOVA). The test compares the effect of the treatments on the dependent variables. To give relevance to the ANOVA [6], we use the Bartlett, Box Cox, and Tukey

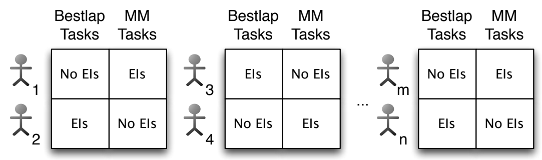


Figure 7. Layout of our experiment design: Latin squares.

tests to verify variance homogeneity, normal distribution, and model additivity, respectively. We follow the convention of considering a factor as being significant to the response variable when *p-value* < 0.05 [6].

4.6 Procedure

After randomly assigning each participant into our Latin Square design, we distribute task description sheets accordingly. Each participant performs two tasks in two individually prepared installations of Eclipse (with Emergo installed or not, with *Best Lap* or *MobileMedia* prepared readily as a project); each installation corresponds to a cell of our Latin Square design. By preparing the Eclipse installation, we prevent participants from using Emergo when they are not supposed to (it is simply not installed in that case). All Eclipse installations support hiding projections, where we leave the first line with the `#ifdef` statement to inform the user of hidden code). Also for the warmup tasks, we prepared a distinct Eclipse installation.

All tasks focus around a specific variable. Since we are not interested in the time needed to locate the variable, we prepare the installations in such a way that the correct files are opened and the cursor is positioned exactly at the maintenance point.

REVIEWER: Third, the authors explicitly ignore in their experiments the time required to locate the variable. The time and effort required to locate a bug is by no means negligible, and even if this time is of no interest to your experiment, reducing a maintenance task to modifying a variable whose location is known is, in my opinion, not a valid simplification of any real maintenance task.

There is no support to find maintenance points; we want to evaluate how efficiently developers change the code given the maintenance point. we focus on the implementation part of a maintenance task. This way, instead of saying "maintenance task", we should consider "code change task", to be more specific and not too general.

From my PhD thesis: "Typically, a maintenance task consists of reading the task, understanding it and the related source code, changing the source code, committing the changes, and reporting it has been done. In our experiments we measure only the implementation part of a maintenance task. Further, our tasks are simple (they take only seconds to be done). However, we claim that bigger tasks are formed by composing smaller ones. So, if we provide benefits for small tasks, it is plausible to consider that we can sum their times up and observe benefits for the whole task. Therefore, we might carefully extrapolate our results to bigger tasks as well."

We prepare all Eclipse installations with an additional plug-in to measure the times automatically. The plug-in adds two buttons: a *Play/Pause* button for participants to start/stop the chronometer; and a *Finish* button to submit a solution. We instruct the participants to press *Play* when starting with the task after reading its description (see Figure 6) and *Finish* when done, and to use *Pause* for breaks (for asking questions during the experiment, for example). In the pilot study, we also recorded the screen to collect qualitative data.

We also partially automated measuring the number of errors. For new-requirements tasks (Tasks 1 and 2), the plug-in automatically checks the submitted solution by compiling the code and running test cases (require about 1 second), as soon as the participant presses *Finish*. If the test passes, we stop and record the time, otherwise we increase the error counter and let the participant continue until the test eventually passes. The test cases are not accessible to the participants. For unused-variable tasks (Tasks 3 and 4) we do not provide immediate feedback but evaluate after the experiment whether one or both variables are correctly fixed. This is because we learned (when watching the pilot screen videos) that participants spend time dealing with compilation errors regarding missing `#endif` statements and tokens like "//" and "#". Because we do not want to measure this extra time, we ask participants to write the `#ifdef` statements in the task description sheets.

REVIEWER: As you point out the time penalty approach you use is rather arbitrary. While I can mostly accept your argument, I am less clear on why you chose to count number of errors for the unused variable task when participants were unable to test their results. This difference between using tests with the new requirements task and not having tests with the unused variables means the participants have a very different experience with each kind of task in using the emergent interfaces. I think the lack of tests for the unused variable tests moves far enough away from actual development that I do not think your results are very meaningful. The combination of the time penalty approach and the counting of errors without tests makes it unclear to me that the overall results of your experiments have sufficient validity.

Remove number of errors for unused variable tasks? Better explain that compilation problems like the ones we observed (#, #endif, "//") would introduce bias into our numbers?

All times using emergent interfaces *include* the time required by Emergo to compute these interfaces. Emergo takes, on the used systems, around 13 seconds and 6 seconds to generate emergent interfaces for Tasks 1 and 2, respectively. To compute interfaces for Tasks 3 and 4, we only need *intraprocedural* analyses, but, to simplify execution, instead of asking the developers to select the analysis to use, we let Emergo automatically apply *interprocedural* ones. So, instead of 1 second or less (*intraprocedural* analyses), it takes more time than needed, around 11, 16, 2, and 3 seconds for the variables `tires`, `xP`, `numberOfViews`, and `albumMusic`, respectively.

To avoid the effect of software installed in different machines and related confounding parameters, we conduct the experiment in a virtual machine (executed on comparable hardware) that provides the same environment to all participants. All participants worked on the same time in the same room under the supervision of two experimenters.

4.7 Executions and Derivations

At the start of the experiment session, we introduce preprocessors, the hiding facilities of virtual separation of concerns, and emergent interfaces. Together with the participants, we perform a warmup task that uses Emergo. We introduce how to use the *Play/Pause* and *Finish* buttons. For the entire experiment, we scheduled 2.5 hours (training, warmup, and execution). No deviations occurred.

5. Results and Discussion

Next we describe the results and test the hypotheses before discussing their implications.[5] We proceed separately with the two tasks (new requirement and unused variable), reporting results from both rounds.

5.1 New-requirement tasks

We plot the times for both new-requirement tasks (1 and 2) with in Figure 8. Here we use beanplot batches, where each batch shows individual observations as small horizontal lines—the longest represents the average of that batch—and the density trace forms the batch shape. In the ~~initial experiment~~ (see Round 1 legend in the figure), the slowest time when using emergent interfaces is still faster than the fastest time without. On average, participants accomplished the task *3 times* faster with emergent interfaces. The key results were confirmed in the replication (emergent interfaces is, on average, 3.1 times faster), despite the different student levels. According to an ANOVA ~~test,~~ we obtain statistically significant evidence that emergent interfaces reduce effort in both new-requirement tasks.

In Figure 9, we plot the ~~NE~~ results for both new-requirement tasks. In Round 1, only one participant committed more errors when using emergent interfaces than without, and all of them committed errors when not using emergent interfaces (they thought they had finished the task but had not, because they potentially missed a dependency). The replication roughly confirms the results: 8 (57%) participants committed errors when not using emergent interfaces, but only 4 (28%) participants committed errors with emergent interfaces. Here we do not perform an ANOVA test on

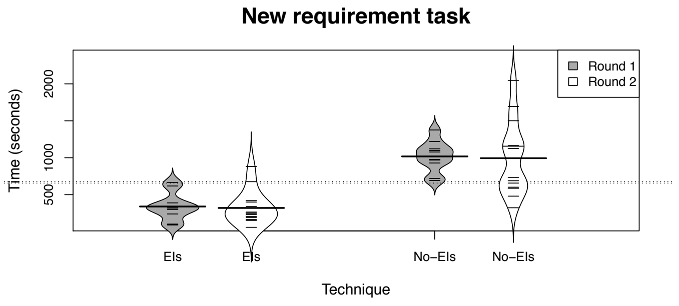


Figure 8. Time results for the new-requirement task in both rounds.

number of errors because we have many zero samples, being hard to observe a tendency and draw significant conclusions.

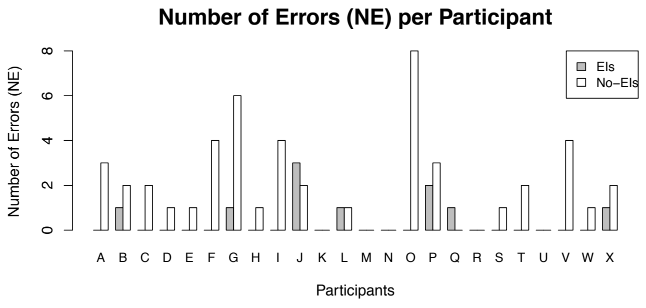


Figure 9. Number of Errors for the new-requirement task in both rounds. A-J: Round 1; K-X: Round 2.

5.2 Unused-variable tasks

Differently from the new-requirement task, here we do not have a test case, so we do not force participants to finish the task correctly. Thus, all participants finished the unused-variable task, but some of them committed errors when writing the feature expressions (so, NE $\neq 0$), which means we could have data of participants that, for example, did not try hard enough. In this manner, we assume that, to accomplish the task correctly, participants would need more time and a test case. So, to compensate for the potential lack of effort on performing this task, we test for statistical significance not only the original data, but also data obtained by adding a time penalty (the standard deviation of all participants times) to the participants that perform the task wrongly, adapting the time data.

As the unused-variable task consists of fixing two unused variables, we consider it formed by two subtasks. The collected time data corresponds to a participant accomplishing both subtasks. So, we assume that participants spend the same time to accomplish both subtasks and add the time penalty as follows: if NE $= 2$, we add the standard deviation; if NE $= 1$, we add the standard deviation divided by two; if NE $= 0$, we add no penalty.

[5] All results are available at `http://www.cin.ufpe.br/~mmr3/icse2013/`

We plot the adjusted times for both unused-variable tasks (Tasks 3 and 4) in Figure 10. Differently from the new-requirement task, here the use of emergent interfaces adds little: the difference between the treatments is smaller. In fact, we obtain statistically significant evidence that emergent interfaces reduce effort only in the second round (this result occurs in both original and adjusted data). Regarding the adjusted time, the participants were 1.6 times faster, on average.

When considering the product lines peculiarities, the *MobileMedia* methods are simpler when compared to the *Best Lap* ones. The time spent to accomplish the unused-variable task for the *MobileMedia* variables is, on average, fairly similar when using and not using emergent interfaces. However, the difference is much greater for the *Best Lap* variables: participants using emergent interfaces are *2* and *2.2 times* faster in the first and second rounds. Again, notice that the results are similar on both rounds.

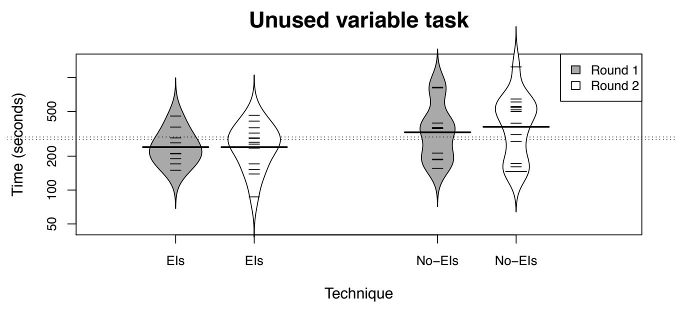


Figure 10. Time results for the unused-variable task in both rounds.

We plot the NE metric in Figure 11. The left-hand side represents Round 1; the right-hand side, Round 2. The errors consist of wrongly submitted `#ifdef` statements. In general, it turns out that participants commit less errors when using emergent interfaces. The *MobileMedia* methods are simpler, which might explain why participants commit less errors when performing the task in such product line.

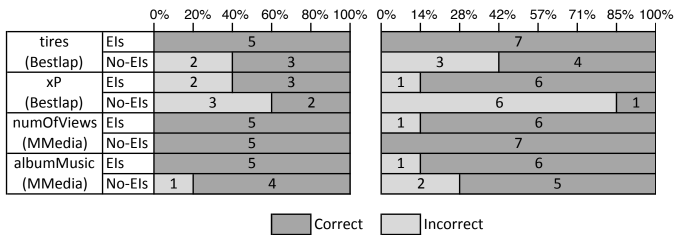


Figure 11. Number of Errors for the unused-variable task in both rounds.

To identify other tendencies, we also performed a meta-analysis, jointly analyzing both rounds by by combining their results. The differences are statistically significant for both tasks.

5.3 Interpretation

Regarding Question 1 (*Do emergent interfaces reduce effort during maintenance tasks involving feature code dependencies in preprocessor-based systems?*), we found that emergent interfaces reduce the time spent to accomplish the new-requirement tasks: the difference is large with a three-fold improvement and statistically significant. Despite different student levels (graduate *versus* undergraduate), we confirm our results in both rounds.

Emergent interfaces make feature dependencies explicit and help our participants to concentrate on the task, instead of navigating throughout the code to find and reason about dependencies. Without emergent interfaces, developers need to check for dependencies in every feature. We can see a qualitative difference between new-requirement tasks that require *interprocedural* analysis across several methods and unused-variable tasks that require to analyze code local to a single method. We argue that the effect is not particularly related to the new-requirement task, but rather to tasks containing *interprocedural* dependencies, since they are much more difficult to follow without tool support; finding and reasoning about these dependencies is not straightforward, since they are in different methods and even classes. In contrast, emergent interfaces contribute comparably little over existing search tools when applied in the local context of a function, especially small ones. In fact, we obtain statistically significant evidence in favor of emergent interfaces in only one round. Still on *intraprocedural* context, our results suggest that, the effort gains might depend on the method complexity and size. When fixing the unused variables, developers spend fairly the same time (with and without emergent interfaces) on average for *MobileMedia* variables. However, they are, on average, 2 times faster with emergent interfaces for the *Best Lap* variables, which are placed at longer methods.

Emergent interfaces can also help when reasoning about an external feature model Emergo can easily incorporate that information. Also in situations in which Emergo reports an empty interface, developers know they can stop searching.

In all cases, the performance gained from emergent interfaces outperforms the extra overhead required to compute them. Overall, we conclude that emergent interfaces may help to reduce effort, but the strength of the effect depends strongly on the task (*interprocedural* or *intraprocedural*, method size and complexity, etc).

Regarding Question 2 (*Do emergent interfaces reduce the number of errors during maintenance tasks involving feature code dependencies in preprocessor-based systems?*), the data we collect suggest that emergent interfaces can reduce errors.

The *new-requirement* task fits into an incomplete fix that has been pointed as a type of mistake in bug fixing [35]. It is "introduced by the fact that fixers may forget to fix all the buggy regions with the same root cause." Here the developer performs the maintenance in one feature but, due to feature dependencies, she needs to change some other feature as well. If she does not change it, she introduces an error into the product line. To discover this kind of incomplete fix, developers should compile and execute one product with the problematic feature combination. Because there are so many potential product combinations, they might discover the error too late.

Given that emergent interfaces make developers aware of feature dependencies, the chances of changing the impacted features increases, leading them to not press the *Finish* button too rashly. This is consistent with recent research [35]: "If all such potential 'influenced code' (either through control- or data-dependency) is clearly presented to developers, they may have better chances to detect the errors." Our results suggest that participants tend to introduce more errors without emergent interfaces. For unused-variable tasks, we can again observe that the longer methods of *Best Lap* are more prone to errors than the shorter methods of *MobileMedia*.

5.4 Threats to validity

As in every experiment, there are several threats to validity. First, there is the obvious threat to external validity that we use students instead of professional developers. Though our students have some professional experience (60 % of our graduate students and 50 % of our undergraduate students reported industrial experience) and researchers have shown that graduate students can perform similar to professional developers [8], we cannot generalize the results to other populations. The results are nevertheless relevant to emerging technology clusters, especially the ones in developing countries like Brazil, which are based on a young workforce with a significant percentage of part time students and recently graduated professionals.

We analyze relatively simple tasks on unfamiliar source code particularly sensitive to knowledge about feature dependencies. We argue that feature dependencies are important for many maintenance tasks in product lines, but cannot generalize to all maintenance tasks. Moreover, in practice developers might remember feature dependencies from knowledge gained in prior tasks. Our results are specific to preprocessor-based implementations, and may potentially generalize to feature modules and aspects, though that requires further experiments.

MobileMedia is a comparably small product line. We minimize this threat by also considering a real and commercial product line with *Best Lap*. The results for both tasks are consistent. So, we need to consider more product lines.

Emergent interfaces depend on dataflow analysis, which is potentially expensive to perform. In our experiments, we have included analysis time, but analysis time may not scale sufficiently with larger projects. In that case developers have to decide between imprecise results or advanced incremental computation strategies. When using imprecise analysis, the use of Emergo could even lead developers to a dangerous sense of security. In our experiment, analysis could be performed precisely in the reported moderate times.

Regarding construct validity, the time penalty that we add to wrong answers for the unused-variable task is potentially controversial. That way, the measured correctness influences the measured time. We argue that to provide the correct answer, participants would need more time and a test case, so the adjustment is realistic. ~~Also, we obtain similar statistical conclusions in both original and adjusted data.~~

Finally, regarding internal validity, we control many confounding parameters by keeping them constant (environment, tasks, domain knowledge) and by randomization. We reduce the influence of reading time by forcing participants to read the task before pressing the *Play* button and the influence of writing time by making the actual changes small and simple.

6. Related work

We complement our previous work [26] by pointing that, to measure effort within a single method, we should combine the proxy metrics we use in that study—number of features and preprocessor directives—with the method size and complexity. Regarding the unused-variable task, one of the *MobileMedia* methods is short but contains high values for these metrics. In contrast, besides these high values, the *Best Lap* methods are larger and more complex. The effort in the former is, on average, almost the same for both treatments. The difference is much greater for the latter (2 times). However, these metrics can also be useful in isolation to evaluate effort, even for short methods. For example, even a few number of features might require extra effort to reason about feature constraints.

REVIEWER: beginning of Section VI is odd; this is related work, not the increment on your own work

Remove it?!

Virtual separation was explored with the tool CIDE, which can hide files and code fragments based on a given feature selection [16]. The version editor [3], C-CLR [27], and the Leviathan file system [14] show only projections of variable source code along similar lines. Similar ideas have also been explored outside the product-line context most prominently in Mylyn [19], which learns from user behavior and creates tasked-based views (usually at the file level). Also in this context, emergent interfaces can help to make dependencies to hidden code fragments visible.

In our evaluation, we investigated only the influence of emergent interfaces, but not of other aspects of preprocessor usage or virtual separation. Recently, Feigenspan et al. have shown in a series of controlled experiments that different representations of conditional-compilation annotations can improve program comprehension [11], and Le et al. have shown in a controlled experiment that hiding irrelevant code fragments can improve understanding of product lines [21]. These results complement each other and help building a tool environment for efficient maintenance of preprocessor-based implementations.

Hidden dependencies are known to be problematic. This can be traced back to avoiding global variables [34], where developers have no information over who uses their variables, since there is “no mutual agreement between the creator and the accessor.” In this context, developers are prone to introduce new errors during fixing activities [35], since information about the agreement is not available. Emergent interfaces support developers maintaining (variable) systems written in languages that do not provide strong interface mechanisms (between features). Indeed, the languages do not have such mechanisms for fine-grained crosscutting features such as the ones we often find in product lines.

We look specifically at preprocessor-based implementations and extensions with views for separation [16], where no interfaces at all exist between features. Although their problems are well known [28], preprocessors are still the most common implementation form for product lines, and we should support developers forced to still maintain them. Nevertheless, aspect-oriented programming [20] and feature-oriented programming [23] have been criticized for lacking interfaces [29, 31]. There are several suggestions to improve modularity of aspects with explicit interfaces [1, 13, 15, 30] or to encode product lines in languages with a stronger module system [10], but those might not work for fine-grained crosscutting features. Also, they are not used in practical product-line development.

In this context, implicit and inferred interfaces, as computed by Emergo, might provide an interesting new point to explore feature modularity. Similar to the idea of virtual separation of concerns where we have no real separation but only emulate some form of modularity at tool level with views, emergent interfaces emulate the benefits of real interfaces at a tool level. It cannot and does not want to replace a proper module system with explicit machine-checked interfaces [1, 10, 13, 30], but it can provide an interesting compromise between specification effort and usability [17].

Conceptual Modules [4] support analyzing the interface of a specific module—also using *def-use* chains internally. Emergent interfaces extend conceptual modules by considering features relationships. Where conceptual modules were evaluated regarding correctness with case studies, we contribute a controlled experiment to evaluate correctness and reduced effort.

Emergo internally uses a variability-aware dataflow analysis that detects dependencies, even between code fragments that belong to different features. The product-line community has also investigated many analysis approaches for whole product lines, most prominently safe composition [32], which performs type checking and detects corresponding dependencies. A product-line type system in CIDE can already raise type errors in specific configurations even if it relates to hidden code [18]. Dataflow analysis in Emergo is more expensive to perform than type analysis, but gives also more precise interfaces.

7. Concluding remarks

We introduce emergent interfaces that emulate missing interfaces in many product line implementations by computing interfaces from dataflow analysis on demand. Emergent interfaces raise awareness of feature dependencies that are critical for maintaining (variable) software systems. With a replicated controlled experiment, we evaluate to what extent such tool support can help on achieving better feature modularization. Our study focuses on feature maintenance tasks in product lines implemented with preprocessors, since they are the prevalent way to implement variable software in industrial practice. We observe a significant decrease in maintenance effort by emergent interfaces, when faced with *interprocedural* dependencies. Similarly, our study suggests a reduction in errors made during those maintenance tasks. In future work, we will focus on scaling the underlying dataflow analysis by trading off performance and precision, and investigate emergent interfaces for other implementation techniques for product lines.

REVIEWER: The conclusion says this paper introduces emergent interfaces. It really does not; previous papers did this.

Focus on the evaluation.

Acknowledgments

Acknowledgments, if needed.

References

[1] J. Aldrich. Open modules: Modular reasoning about advice. In *Proc. of the 19th European Conference on Object-Oriented Programming (ECOOP)*, pages 144–168. Springer, July 2005.

[2] V. Alves. *Implementing Software Product Line Adoption Strategies*. PhD thesis, Federal University of Pernambuco, Brazil, March 2007.

[3] D. L. Atkins, T. Ball, T. L. Graves, and A. Mockus. Using version control data to evaluate the impact of software tools: A case study of the Version Editor. *IEEE Transactions on Software Engineering (TSE)*, 28(7):625–63, 2002.

[4] E. L. A. Baniassad and G. C. Murphy. Conceptual module querying for software reengineering. In *Proc. of the 20th International Conference on Software Engineering (ICSE)*, pages 64–73, Washington, DC, USA, 1998. IEEE Computer Society.

[5] E. Bodden, T. Tolêdo, M. Ribeiro, C. Brabrand, P. Borba, and M. Mezini. Spllift - transparent and efficient reuse of ifds-based static program analyses for software product lines. In *Proc. of the 34th annual ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, Seattle, USA, 2013. To appear.

[6] G. E. P. Box, J. S. Hunter, and W. G. Hunter. *Statistics for Experimenters: design, innovation, and discovery*. Wiley-Interscience, 2005.

[7] C. Brabrand, M. Ribeiro, T. Tolêdo, and P. Borba. Intraprocedural dataflow analysis for software product lines.

In *Proc. of the 11th International Conference on Aspect-Oriented Software Development (AOSD)*, pages 13–24, Potsdam, Germany, 2012. ACM. ISBN 978-1-4503-1092-5. doi: 10.1145/2162049.2162052.

[8] R. P. Buse, C. Sadowski, and W. Weimer. Benefits and barriers of user evaluation in software engineering research. *ACM SIGPLAN Notices*, 46(10):643–656, October 2011. ISSN 0362-1340. doi: 10.1145/2076021.2048117.

[9] M. Cataldo and J. D. Herbsleb. Factors leading to integration failures in global feature-oriented development: an empirical analysis. In *Proc. of the 33rd International Conference on Software Engineering (ICSE)*, pages 161–170, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0445-0. doi: 10.1145/1985793.1985816.

[10] W. Chae and M. Blume. Building a family of compilers. In *Proc. of 12th International Software Product Line Conference (SPLC)*, pages 307–316, Los Alamitos, CA, 2008. IEEE Computer Society.

[11] J. Feigenspan, C. Kästner, S. Apel, J. Liebig, M. Schulze, R. Dachselt, M. Papendieck, T. Leich, and G. Saake. Do background colors improve program comprehension in the #ifdef hell? *Empirical Software Engineering*, 2012. doi: http://dx.doi.org/10.1007/s10664-012-9208-x.

[12] E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. Filho, and F. Dantas. Evolving software product lines with aspects: an empirical study on design stability. In *Proc. of the 30th International Conference on Software Engineering (ICSE)*, pages 261–270, New York, NY, USA, 2008. ACM.

[13] W. G. Griswold, K. Sullivan, Y. Song, M. Shonle, N. Tewari, Y. Cai, and H. Rajan. Modular software design with crosscutting interfaces. *IEEE Software*, 23(1):51–60, 2006. ISSN 0740-7459. doi: http://dx.doi.org/10.1109/MS.2006.24.

[14] W. Hofer, C. Elsner, F. Blendinger, W. Schrder-Preikschat, and D. Lohmann. Toolchain-independent variant management with the leviathan filesystem. In *Proc. of GPCE Workshop on Feature-Oriented Software Development (FOSD)*, pages 18–24, 2011.

[15] M. Horie and S. Chiba. Aspectscope: An outline viewer for AspectJ programs. *Journal of Object Technology*, 6(9):341–361, 2007.

[16] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *Proc. of the 30th International Conference on Software Engineering (ICSE)*, pages 311–320, New York, NY, USA, 2008. ACM.

[17] C. Kästner, S. Apel, and K. Ostermann. The road to feature modularity? In *Proc. SPLC Workshop on Feature-Oriented Software Development (FOSD)*, New York, Sept. 2011. ACM Press.

[18] C. Kästner, S. Apel, T. Thüm, and G. Saake. Type checking annotation-based product lines. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 21(3), 2012.

[19] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *Proc. of the 14th International Symposium on Foundations of Software Engineering (FSE)*, pages 1–11, New York, NY, USA, 2006. ACM.

[20] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect–Oriented Programming. In *Proc. of the 11th European Conference on Object–Oriented Programming (ECOOP)*, pages 220–242, 1997.

[21] D. Le, E. Walkingshaw, and M. Erwig. #ifdef confirmed harmful: Promoting understandable software variation. In *IEEE International Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 143–150, 2011.

[22] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *Proc. of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 105–114, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-719-6. doi: http://doi.acm.org/10.1145/1806799.1806819.

[23] R. Lopez-Herrejon, D. Batory, and W. Cook. Evaluating support for features in advanced modularization technologies. In *Proc. of the 19th European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *Lecture Notes in Computer Science*, pages 169–194, Berlin/Heidelberg, 2005. Springer-Verlag. ISBN 3-540-27992-X.

[24] A. Rashid, T. Cottenier, P. Greenwood, R. Chitchyan, R. Meunier, R. Coelho, M. Sudholt, and W. Joosen. Aspect-oriented software development in practice: Tales from aosd-europe. *Computer*, 43(2):19 –26, 2010. ISSN 0018-9162.

[25] M. Ribeiro, H. Pacheco, L. Teixeira, and P. Borba. Emergent Feature Modularization. In *Onward!, affiliated with ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH)*, pages 11–18, New York, NY, USA, 2010. ACM.

[26] 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. In *Proc. of the 10th ACM International Conference on Generative Programming and Component Engineering (GPCE)*, pages 23–32, Portland, Oregon, USA, 2011. ACM.

[27] N. Singh, C. Gibbs, and Y. Coady. C-CLR: A tool for navigating highly configurable system software. In *Proc. AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, page 9, New York, 2007. ACM Press. ISBN 1-59593-657-8. doi: http://doi.acm.org/10.1145/1233901.1233910.

[28] H. Spencer and G. Collyer. #ifdef considered harmful, or portability experience with C news. In *Proc. of the Usenix Summer 1992 Technical Conference*, pages 185–198, Berkeley, CA, USA, June 1992. Usenix Association.

[29] F. Steimann. The paradoxical success of aspect-oriented programming. In *Proc. of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA)*, pages 481–497, New York, 2006. ACM Press. ISBN 1-59593-348-4. doi: http://doi.acm.org/10.1145/1167473.1167514.

[30] F. Steimann, T. Pawlitzki, S. Apel, and C. Kästner. Types and modularity for implicit invocation with implicit announcement. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(1):Article 1; 43 pages, 2010. doi: http://doi.acm.org/10.1145/1767751.1767752.

[31] M. Störzer and C. Koppen. Pcdiff: Attacking the fragile pointcut problem, abstract. In *European Interactive Workshop on Aspects in Software*, Berlin, Germany, September 2004.

[32] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe composition of product lines. In *Proc. of the 6th International Conference on Generative Programming and Component Engineering (GPCE)*, pages 95–104, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-855-8. doi: http://doi.acm.org/10.1145/1289971.1289989.

[33] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering (ICSE'81)*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press. ISBN 0-89791-146-6. URL `http://portal.acm.org/citation.cfm?id=800078.802557`.

[34] W. Wulf and M. Shaw. Global variable considered harmful. *SIGPLAN Notices*, 8(2):28–34, 1973.

[35] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How do fixes become bugs? In *Proc. of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (ESEC/FSE)*, pages 26–36, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0443-6.