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 provide a comprehensive 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* data-flow analysis. They emerge in the IDE and emulate benefits of modularity not available in the host language. To evaluate the potential of emergent interfaces, we conducted and replicated a controlled experiment, and found, in the studied context, that emergent interfaces can improve performance of code change tasks by up to 3 times while also reducing the number of errors.

1. INTRODUCTION

Developers often introduce errors into software systems during maintenance when they fail to recognize module and feature dependencies [11]. This problem is particularly critical for configurable 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 [31] and share program elements like variables and methods [38], without proper modularity support from a notion of interface between features. In such context, developers can easily miss cross-feature dependencies, such as a feature assigning a value to a variable read by another feature. As there is no mutual agreement [46] between separate feature developers, changing one feature might be the correct action for maintaining the 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 large-scale open-source implementations based on preprocessor mechanisms, we found that cross-feature dependencies are frequent in practice [38].

To reduce this feature dependency problem, we propose a technique called *emergent interfaces* that establishes, on demand and according to a given code change task, interfaces to feature code (introduced previously in a vision paper [37]). An emergent interface is a list of contracts stating the code lines data dependent on the code lines we are supposed to change during the task. We call our technique emergent because, instead of writing interfaces manually, developers can request interfaces on demand; that is, interfaces emerge to support a specific code change task. This way, developers become aware of feature dependencies, and may have better chance of not introducing errors [47]. Our interfaces may also help to reduce code change effort. In fact, instead of searching for dependencies throughout the code, and reasoning about requirements-level feature constraints, developers can rely on proper tool support that computes interfaces.

We implemented emergent interfaces in a tool *Emergo*, available as Eclipse plugin for Java. Emergo performs feature-sensitive data-flow analysis to compute interfaces on demand, both at *intraprocedural* and at *interprocedural* level.

A key novelty in this paper is an empirical evaluation of emergent interfaces as provided by Emergo. We conducted and replicated a *controlled experiment* on feature-related code change tasks in software product lines. The product lines are implemented with a preprocessor-like variability mechanisms, which are widely used to implement compile-time variability in industrial practice, despite their lack of modularity mechanisms. In particular, we evaluate emergent interfaces by answering two research questions: *Do emergent interfaces reduce effort during code change tasks involving feature code dependencies in preprocessor-based systems? Do emergent interfaces reduce the number of errors during code change 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, recruiting graduate students as subjects, and then replicate it in another institution with undergraduate students.

Our experiment reveals that, in our setting, emergent interfaces significantly reduce maintenance effort for tasks involving *interprocedural* dependencies, which cross method boundaries. Both experiment rounds reveal that developers were, on average, 3 times faster completing our code change

ICSE '14 Hyderabad, India

tasks 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 [47], 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:

- A comprehensive introduction to emergent interfaces and a complete implementation of the concept in Emergo, supporting both *intraprocedural* analysis but also on the much more powerful *interprocedural* analysis.
- An empirical evaluation assessing the potential of emergent interfaces. 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.

The idea of emergent interfaces was first introduced in an Onward! paper [37], with an early prototypical implementation approximating *intraprocedural* data-flow analysis. Subsequently, we statically analyzed the potential impact of emergent interfaces in 43 open source projects [38] and investigated feature-sensitive data-flow analyses [8]. This paper brings together these results and reports on a significantly revised and extended version of Emergo that supports precise, feature-sensitive, and *interprocedural* data-flow analysis and additionally complements them with a novel significant empirical evaluation.

2. MAINTAINING PREPROCESSOR-BASED PRODUCT LINES

Inadequate modularity mechanisms are plaguing many languages and cause many implementation problems. Emergent interfaces are applicable to many situations, where explicit interfaces between code fragments are lacking. While we will hint at many other use cases, to illustrate and explore the idea of emergent interfaces, we look at a scenario that is especially challenging to developers involving non-modular code fragments and variability: preprocessor-based product lines.

Variable systems, especially in the form of software product lines, are challenging, because code fragments are configurable and may not be included in all product configurations. That is, developers need to reason about potentially different control and data flows in different configurations. At the same time, when variability is implemented with preprocessor directives, code fragments belonging to a feature are marked (annotated) but not encapsulated behind an interface. Therefore the control and data flow across feature boundaries is implicit (but common, as we found in a previous study [38]). 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 (as we illustrate next with two scenarios), but due to the sheer number of them, rapid feedback by testing is not possible for all products.

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]. The game is written in Java and uses the Antenna c-style preprocessor.[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.

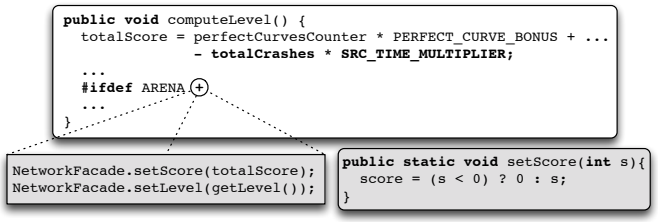

Figure 1: Code change only works for some products. *ARENA* feature code in gray.

In this scenario, consider the following planned change. To add penalties in case the player often crashes the car, let the game score be not only positive, but also negative. To accomplish the task, a developer localizes the *maintenance points*, in this case the `totalScore` assignment, and changes the computation of scores (see the bold line in Figure 1). Products without the *ARENA* feature now enjoy the new functionality, but unfortunately the change is incomplete for products with the *ARENA* feature. In the implementation of feature *ARENA*, method `setScore` again checks for positive values and unintentionally prevents submitting 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 dependency if the method is not so big. However, in more complex code or even alternative implementation approaches that separate the feature implementation (see Section 2.4 below) the dependencies across feature boundaries might be harder to track.

In this context, searching for cross-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 code change task, for example.

2.2 Scenario 2: Fixing an unused variable

Our second scenario is based on a bug report from *glibc*.[3] This project is structured with several preprocessor macros

[1] `http://www.meantime.com.br/`

[2] `http://antenna.sourceforge.net/wtkpreprocess.php`

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

and conditional-compilation constructs. Developers report that a variable `status` in the common code base is reported as unused. Investigating the problem, we find that `status` is declared in all configurations, but only used when features *CHOWN* or *UTIMES* are selected, as shown in Figure 2 (left-hand side). When we compile the product line without either feature, the compiler issues an unused-variable warning.

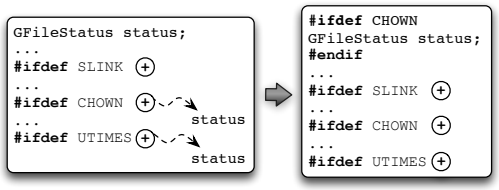

Figure 2: Wrong fixing of an unused variable.

To fix the bug report, a developer would typically look for uses of the variable. If she does not carefully look across feature boundaries, she can easily introduce an error. The problem can even be worse when there are requirements-level dependencies between features, e.g., that *SLINK* cannot be selected without *CHOWN*.

In an unsuccessful attempt to fix the warning, the developer could, for example, detect only 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. Again, these are easy to detect in small and simple methods, but might be complicated in larger code bases and other language mechanisms that actually separate the feature code (see Section 2.4 below).

2.3 Cross-Feature Dependencies in the Wild

Initially, the previously shown examples seem pretty specific, requiring preprocessor directives and data-flow dependencies. To quantify their frequency, we have previously conducted a conservative study [38] mechanically mining the code base of 43 highly configurable software systems with a total of over 30 million lines of code, including *Linux*, *Freebsd*, *postgres*, *sendmail*, *gcc*, and *vim*. All these common open-source systems make heavy use of preprocessor directives for configuration (for features and portability). Even just looking conservatively at *intraprocedural* data-flow within individual methods, between 1 and 24 percent of all methods in the studied systems contain cross-feature dependencies; however, typically more than half of the methods with `#ifdef` directives also contain cross-feature dependencies. These numbers only serve as a lower bound, since *interprocedural* data-flow between methods was not measured but likely causes additional cross-feature dependencies. These results show that the problem, even though quite specific, is so common in practice that building dedicated tool support can be beneficial for a wide range of code change tasks.

2.4 Beyond preprocessors

We illustrate the problem for preprocessor-based product lines, but in fact other implementation approaches suffer from limited modularity mechanisms, especially implementation approaches supporting some form of crosscutting. While variability can make cross-feature dependencies harder to detect, it is by no means necessary.

One of the well-known and controversially discussed examples is *aspect-oriented programming* in the style of AspectJ. With AspectJ, code of features (or more generally concerns) is separated into a distinct code unit, the aspect, and reintroduced in a weaving step. The control-flow or data-flow between aspects and base code is not protected by explicit interfaces—a fact for which AspectJ was repeatedly criticized [44, 42], but which was also discussed as strength enabling flexibility [16].[4] The first example works just as well with an aspect injecting the *ARENA* code instead of an in-place `#ifdef` block.

Other structure-driven composition mechanisms, such as *feature-oriented programming* [5], *delta-oriented programming* [39], or even just inheritance with *subclassing* [33] exhibit similar potential problems.

Finally, also in the context of preprocessor-based implementations, recent advances support some separation of feature code. 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 [23, 3, 30, 18]. For example, in CIDE [23], 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. In our examples, we have already shown the collapsed versions of `#ifdef` statements with a $\oplus$ marker indicating additional code. Virtual separation in this form has been shown to allow significant understandability and productivity gains [3, 30]. However, hiding also has a similar effect as moving code into an aspect: it is no longer visible locally (except for a marker indicating hidden code) and there is no interface describing the hidden code. This way, virtual separation makes the problem of cross-feature dependencies even worse.

3. EMERGENT INTERFACES

The problems discussed so far occur when features *share* elements such as variables and methods, raising cross-feature dependencies. For instance, the common base code might declare a variable subsequently used by an optional feature (see `totalScore` and `status` in Figures 1 and 2, respectively).

In this context, there are several paths to attack the problem with unclear cross-feature dependencies outlined in the previous section. The typical language-designer approach is to introduce additional modularity concepts into the programming language and make control-flow and data-flow explicit in interfaces, such as [29]. With *emergent interfaces*, we pursue an alternative tool-based direction, which also works with existing languages and existing implementations and infers interfaces on demand.

[4]Several extensions to aspect-oriented languages have been discussed that require declaring explicit interfaces between concerns [43, 22, 1, 17, 35]; here we take an alternative tool-based approach.

To better explain what is an emergent interface, we first introduce maintenance points. *Maintenance points are the code lines that the developer wants to change, for which she is interested in the cross-feature dependencies.* Also, we name *impacted points the code lines we potentially impact if we change the maintenance points.* As any other code line, all these lines might be associated with feature expressions by using `#ifdef` statements. Notice that one maintenance point—one code line—may impact many other code lines.

To define an emergent interface, let MP be the set of maintenance points and IP be the set of potentially impacted points. Also, let $FE(line)$ be a function that returns the feature expression associated with a given code line. This way, *an emergent interface is a list of contracts in terms of "IP_i requires MP_j" emerging only if* $FE(MP_j) \not\equiv FE(IP_i)$ *and* $FE(MP_j) \wedge FE(IP_i)$ *is a valid feature expression.* Notice that for each contract of the list we have two elements: the maintenance point (the one that *provides* data); and the impacted point (the one that *requires* data). In other words, an emergent interface contains a list of contracts stating the code lines data dependent on the maintenance points. However, to emerge, each contract of the list must satisfy two constraints: the maintenance point and impacted point must have not equivalent feature expressions; and the conjunction of these expressions must be valid. The not equivalent constraint is necessary to compute cross-feature dependencies and the conjunction constraint prevents developers from assuming invalid dependencies.

Now, when developers are interested in dependencies from specific maintenance points, they can ask the tool that implements the technique to compute interfaces, pointing out feature dependencies. The interfaces emerge on demand, giving support for developers to maintain one feature without breaking others, since now they are aware of cross-feature dependencies. That is, the interfaces are inferred and shown in the IDE environment, instead of being written manually by developers.

To better illustrate how emergent interfaces work, we now refer to Figure 3, which illustrates two maintenance tasks. When considering task 1, suppose there is something wrong with the initialization of variables `x`, `y`, and `k`. So, we should change them. Regarding task 2, we should change the `return` expressions of function `h`. For task 1, we select the following maintenance points (see the non-contiguous lines in dashed rectangles): lines 12, 13, 14, and 27 of `File F1`; afterwards, for task 2, we select lines 83 and 85 of `File F2`.

The emergent interfaces for tasks 1 and 2 contain six contracts and one contract, respectively. Each interface details, for each maintenance point, the potentially impacted statement, its line, its file, and the feature configuration in which the maintenance point impacts such a statement. We assume that the dots "..." in Figure 3 do not contain any assignment to the variables we are interested during our maintenance tasks.

According to the emergent interface 1, if we change the statement `int x = 0` we impact the statements in lines 53 and 87, since they use the value of `x` and there is no path from the maintenance point to these lines where we assign a new value to `x`. Because we impact the statement `z = x + 9`, we transitively impact line 88—`print(z)` statement—, since our maintenance in variable `x` contribute to define the value of `z` as well. This happens when we enable features `B`, `C`, and `D` (see column "Feature Config."). When considering `y = g()` we impact line 87 in products with `!A && C && D`. Due to the new assignment to `y` in feature `A`, this situation only occurs in case feature `A` is not enabled. Finally, if we change the assignment to variable `k` in line 14 we impact lines 27 in `File F1` and 33 in `File F2`. To impact line 33, the product must have feature `B` enabled and feature `E` not enabled. Notice that the maintenance point `k++` does not impact any line, including `i(k)`. According to the feature constraints at the bottom of Figure 3, features `A` and `B` are mutually exclusive, so they do not impact each other. Therefore, we do not emerge any contract for this case—$FE(\texttt{i(k)}) \wedge FE(\texttt{k++}) = A \wedge B$ is not valid—preventing developers from assuming a non-existing dependency. Notice that constraints like this one might not be explicit in source code or even are unknown by developers.

According to the emergent interface 2, we impact only one point, i.e., line 87 in `File F1`. The statement `return m + n` impacts the call to function `h` in products with `!F && C && D`. The statement `return m` does not impact such a call because if we enable feature `F`, we must disable feature `C` (see the constraint in Figure 3).

Now, because developers are aware of cross-feature dependencies pointed by the emergent interface, they have better chances of not introducing errors to the code lines dependent on the maintenance points. In addition, developers do not need to reason about feature constraints hardly ever explicit in code; emergent interfaces already does it. Emergent interfaces also bring benefits to virtual separation and even for testing activities. We can improve comprehensibility (we hide the features not dependent of our task and focus on the ones we really care) and reduce effort by testing only the feature configurations pointed by our interfaces.

The tool we present next compute interfaces for a given set of maintenance points, helping developers to make code changes once they identify these points. Emergent interfaces do not contribute to finding the maintenance points in the first place, through.

3.1 Implementation: Emergo

We implemented the concept of emergent interfaces in an Eclipse-based 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 data-flow analysis [8, 6]. 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). Again, developers might assume that changing the `k++` assignment in feature A—in Figure 3—may lead to problems in feature B (see the `i(k)` statement). Since the involved features are mutually exclusive, Emergo would not emerge any interface, which means that code change tasks in the former feature do not impact the latter, and vice-versa.

To illustrate Emergo in action, we return to our scenarios from the previous section. Consider *Scenario 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*. In our case, the developer manually selects the `totalScore` assignment as maintenance point. Then, she would rely on Emergo to analyze the code to capture dependencies between the feature she is maintaining and the others. Finally, the interface emerges stating

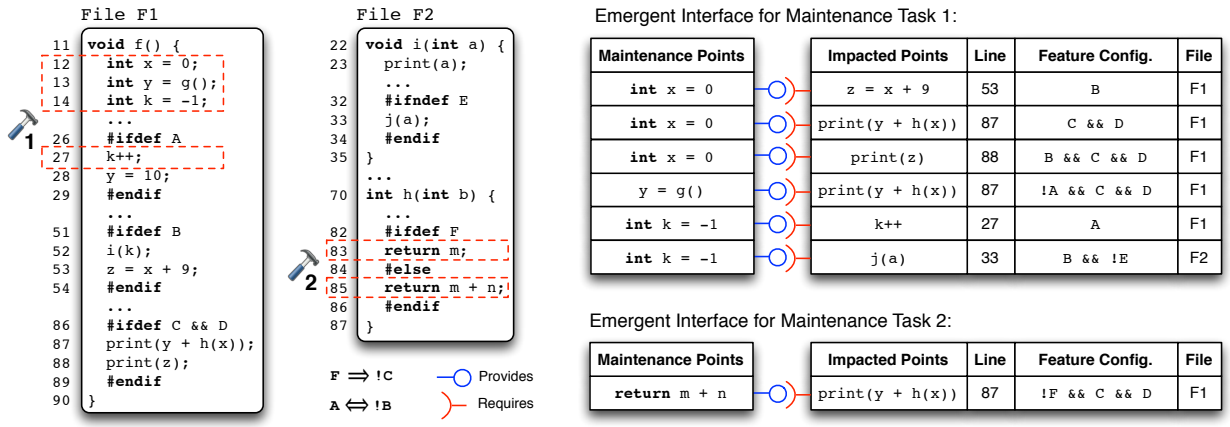

Figure 3: Examples of maintenance points and their respective emergent interfaces.

that the code change task may impact the behavior of products containing the *ARENA* feature. So, the common code base 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.

Figure 4 illustrates a screen shot of Emergo. After the developer found and selected the maintenance point in Line 1277 (the `totalScore` assignment), 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 points. Since there is no `#ifdef` statement encompassing the maintenance point, we associate it with the *mandatory* feature. We show the potentially impacted configuration in the "Feature" column, here with the *ARENA* feature. The table view also shows the exact (impacted) lines of code that contain uses of `totalScore` as well as their respective 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 products with 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 one, she can click on the corresponding table line and Emergo automatically removes unrelated dependencies of the graph. This means we only show the path associated with the dependency of that table line. According to Figure 4, she selected the first line of the table. So, the graph now has only the path from the maintenance point to line 177 of the `NetworkFacade` class.

Also, developers can reach the impacted feature code in the IDE editor 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.

Emergo can also help on preventing developers from analyzing unnecessary features and their associated code, which is important to decrease code change effort. In particular, 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 *CHOWN* and *UTIMES*. So, we could keep *SLINK* hidden, since it is not related to the current task.

Currently, our implementation has some limitations. To compute dependencies, we rely only on data-flow information, so we only compute data dependencies. Also, developers can only select maintenance points that range from simple variable assignments to code blocks that do not span to another method.

3.2 Underlying Analysis

The current implementation of Emergo computes interfaces based on data using *interprocedural* and *intraprocedural* feature-sensitive *reaching-definition analysis*. So, from the maintenance points, we consider the reached program statements and their associated feature expressions to form our emergent interfaces. 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 [8, 6], which is important for interactive tools like ours that need to provide quick responses to developers requests. To perform the feature-sensitive analysis, we annotate the control-flow graph with feature information, lift the lattice to contain a mapping of sets of configurations to lattice values, and lift the transfer functions to figure out whether or not apply the ordinary function. The lifted function lazily splits the sets of configurations in two disjoint parts, depending on the feature expression annotated with the statement being analyzed: a set for which the function should be applied; and a set for which it should not [9].

4. EXPERIMENTAL DESIGN

In the previous section we suggest that emergent interfaces can make feature code change 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 code change effort and introduced errors when maintaining preprocessor-based product lines

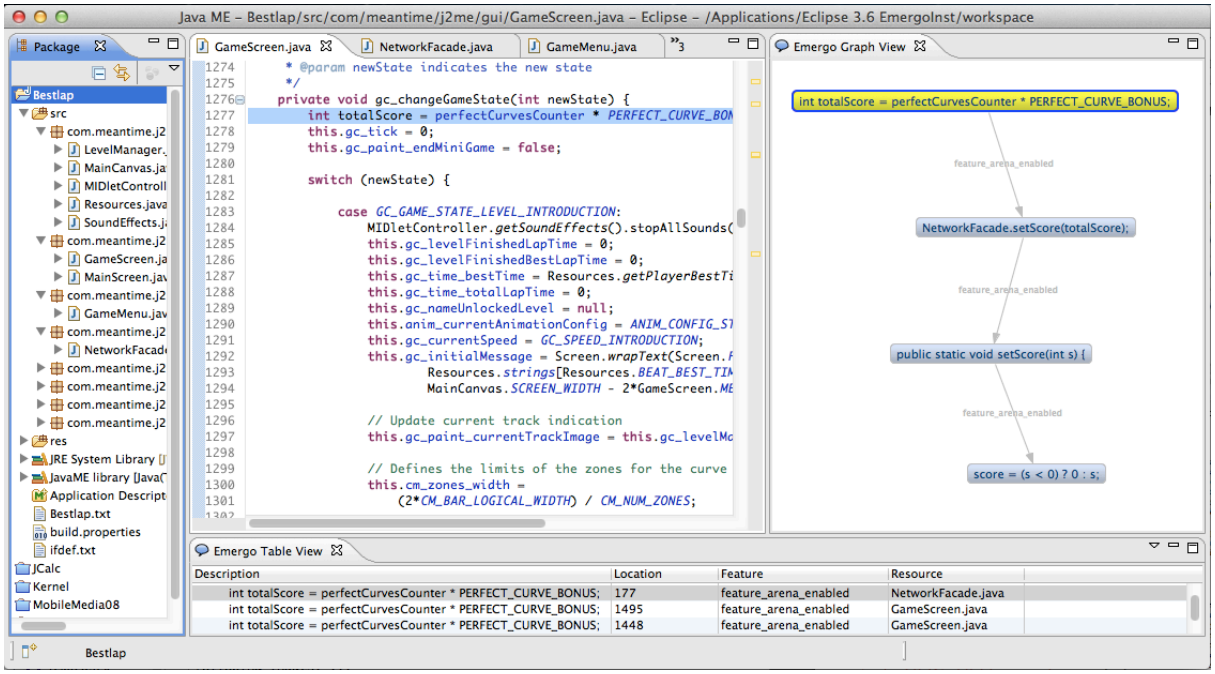

Figure 4: Emergent interface for Scenario 1 using Emergo.

with and without our 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 code change tasks involving feature code dependencies in preprocessor-based systems?*
- Question 2: *Do emergent interfaces reduce the number of errors during code change tasks involving feature code dependencies in preprocessor-based systems?*

To answer Question 1 (effort), we measure the time required to find cross-feature dependencies and to change the impacted features to accomplish a code change task. Figure 5 illustrates our setup with and without emergent interfaces. Note that our setup does not measure the time needed to find the maintenance point (we actually provide the maintenance point with our task description as we describe below). While finding the maintenance point may dominate the entire tasks in a real-world setting, emergent interfaces do not contribute to that part. Hence, we measure only the part of the maintenance task after the maintenance point was identified. This focus of our measurement eliminates noise that would not contribute to our analysis.

To answer Question 2 (correctness), we measure how many incorrect solutions the developer committed during a code

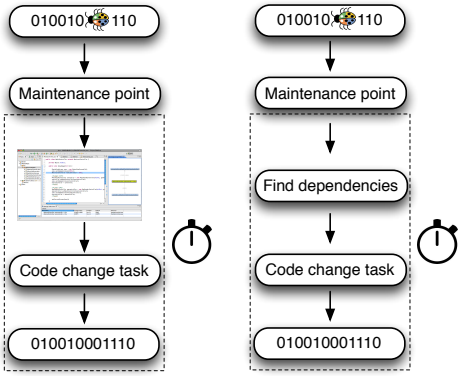

Figure 5: Dashed rectangles represent the time we count (with and without emergent interfaces).

change 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 described below, for some tasks, we provide automated feedback to our participants, so the participant can retry after an incorrect attempt. Other tasks are evaluated manually after the experiment, so participants have only one attempt.

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 rounds, 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 (Undergraduate Research Opportunity Program) students and volunteered to participate. All took part voluntarily and were informed they could stop participating at any time, but nobody did.

4.3 Material and Code Change 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 [15]. It contains feature restrictions and has been used in previous research studies [15, 36].

We ask participants to perform a number of code change 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. We selected tasks that are particularly affected by cross-feature dependencies, where we expect that our tool can contribute to a better understanding. Note that emergent interfaces target a specific class of problems; for other maintenance tasks we would not expect any benefit. We believe that our task selection can represent typical cross-feature problems as outlined in Section 2.

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. All task descriptions are available online, see Appendix A.

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 cross-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.[5] 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.

Emergo computes dependencies for these tasks using *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 similarities, but they are not equivalent. Actually, these differences—methods size, method call depths to reach the impacted feature, and number of conditionals to change—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.

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.

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 code change tasks involving feature dependencies in both product lines.
- *H2 - Error introduction:* With emergent interfaces, developers commit less errors in both product lines when performing the code change tasks 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

[5]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`

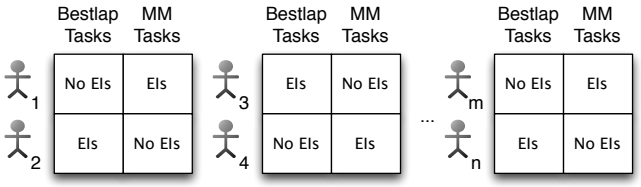

Figure 6: Layout of our experiment design: Latin squares.

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* [7]. 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 (see Figure 6). 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 code change tasks.

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 [7], we use the Bartlett, Box Cox, and Tukey 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 $p\text{-}value < 0.05$ [7].

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 virtual separation, 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.

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

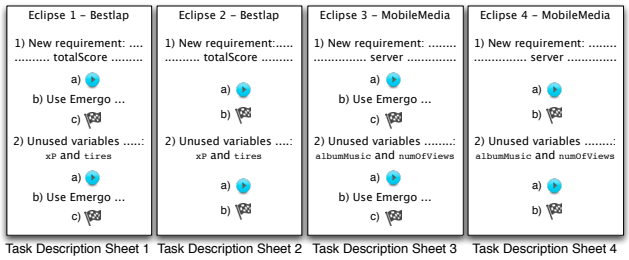

Figure 7: Task description sheets we distributed to the participants.

starting with the task after reading its description and *Finish* when done, and to use *Pause* for breaks (for asking questions during the experiment, for example). To collect qualitative data, in the pilot study we also recorded the screen.

To illustrate the tasks from the participant's perspective, we summarize in Figure 7 the task description sheets we distributed. We represent the steps that participants should follow as "a", "b", and "c". Notice that we associate each sheet with a different Eclipse instance.

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 recordings) 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` feature expression in the task description sheets. For example, to fix the unused variable illustrated in Section 2.2, they can write "`CHOWN || UTIMES`" in the sheet.

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 prepro-

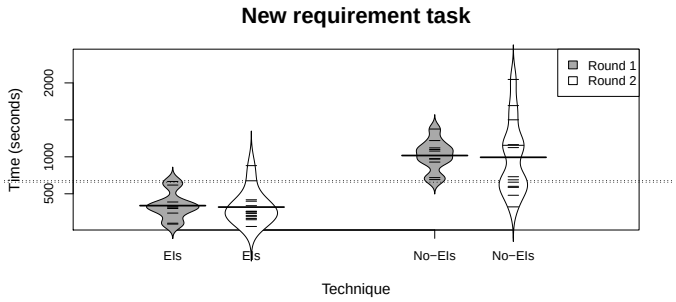

Figure 8: Time results for the new-requirement task in both rounds.

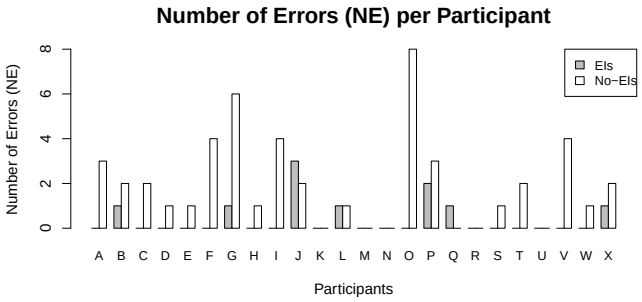

Figure 9: Number of Errors for the new-requirement task in both rounds. A-J: Round 1; K-X: Round 2.

cessors, 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 (all raw results are available online, see Appendix A). 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 Round 1 (see the 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 (see Appendix B) 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, potentially because they 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 number of errors because we have many zero samples, being hard to observe a tendency and draw significant conclusions.

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. We make this important decision after reviewing the screen recordings from the pilot study. When fixing the unused variable problem, participants spend time since they miss statements such as `#endif` and tokens like "//" and "#", essential to compile the code and run the test, but typically less common when somebody is more familiar with the used notation. Because including this time would introduce bias into our results, we ask participants to write the `#ifdef` feature expression in the task description sheets, not in the source code. Thus, all participants finished the unused-variable task, but some 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 and consequently finished the task earlier.

Regarding the measured time, it actually only reflects the time participants need until they think they are done. To reflect incorrect solutions in the time, we could add a time penalty for incorrect tasks that simulates the extra time participants would have needed, if we mechanically reported the error or if they found the problem unfixed in practice. As time penalty for an incorrect solution, we add the half the standard deviation of all participants times. We analyze both the original time (time until they think they are done) and the adjusted time with the penalty for incorrect tasks (which can be seen as a form of sensitivity analysis [19]).

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. The statistical results are stable for the original time (time until they think they are done) and the adjusted time. 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.

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,

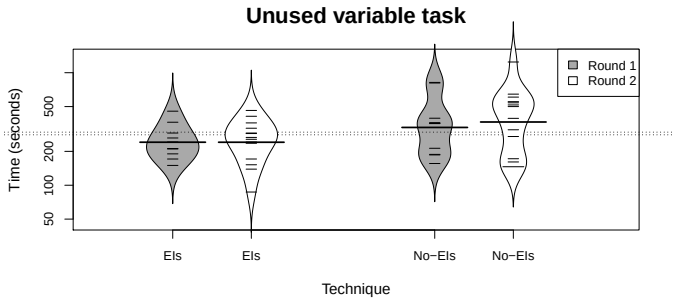

Figure 10: Time results for the unused-variable task in both rounds.

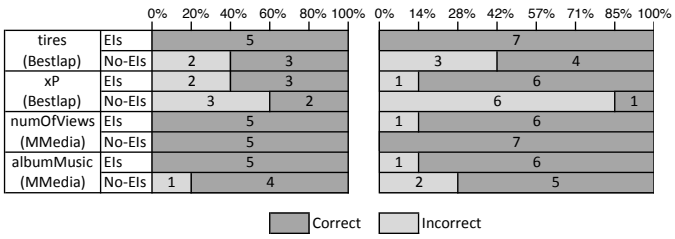

Figure 11: Number of Errors for the unused-variable task in both rounds.

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.

To identify other tendencies, we also performed a meta-analysis, where we combine and analyze both rounds results. The time differences are statistically significant for both tasks.

5.3 Interpretation

Effort reduction.

Regarding Question 1 (effort reduction), 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), the results are stable across both rounds.

We regard this as a confirmation that emergent interfaces make cross-feature dependencies explicit and help our participants to concentrate on the task, instead of navigating throughout the code to find and reason about dependencies.

Additionally, 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 only code of a single method, where tasks involving *interprocedural* analysis show higher speedups. We argue that the effect is general to tasks with *interprocedural* dependencies, since they are more difficult to follow without tool support. In contrast, emergent interfaces contribute comparably little over simple textual search tools when applied in the local context of a function, especially small ones. Still, we can carefully interpret our results as suggesting that the effort gains might depend on the method complexity and size in the *intraprocedural* context: speedups were considerably higher in the *Best Lap* task, where variables were placed in longer methods.

In all cases, the performance gained from emergent interfaces outperforms the extra overhead required to compute them. Overall, we conclude that, for code change tasks involving cross-feature dependencies, emergent interfaces can help to reduce effort, while the actual effect size depends on the kind of task (*interprocedural* or *intraprocedural*, method size and complexity, etc).

Correctness.

Regarding Question 2 (reducing the number of errors made), our experiment suggests 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 [47]. 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 code change in one feature but, due to cross-feature dependencies, she needs to change some other feature as well. If she does not change it, she introduces an error. To discover this kind of incomplete fix, developers should compile and execute one product with the problematic feature combination. Since there are so many potential product combinations, they might discover the error too late.

Given that emergent interfaces make developers aware of cross-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 [47]: "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*.

Outlook.

First, Our experiment considered the influence of emergent interfaces in a specific scenario of preprocessor-based product lines. As we argued in Section 2, the concept can be generalized to enhance other implementation mechanisms with insufficient modularity mechanisms. Of course, we cannot simply transfer our results to these settings, but we are confident that, in follow-up experiments, we could find similar improvements also for task involving cross-feature dependencies in aspect- or feature-oriented implementations.

Second, our tasks were tailored to the specific capabilities of emergent interfaces. As discussed earlier, emergent interfaces address a narrow but important class of problems. We believe that it may actually be beneficial to run emergent interfaces in the background during any code changes and notify the user in any case the change would involve cross-feature dependencies. This way, a tool could lurk unnoticed in the background but would become active for specifically the kind of tasks we analyzed in our experiment to bring corresponding benefits regarding efficiency and correctness to those changes.

Finally, emergent interfaces have several capabilities that we did not explore in our experiment. For instance, in product lines often dependencies between features exist. Whereas

a developer needs to manually search for dependencies and compare `#ifdef` annotations on code statements with dependencies specified elsewhere, the underlying feature-sensitive data-flow analysis of Emergo can take such dependencies into account mechanically and automatically exclude infeasible paths (see Figure **??**). Furthermore, even situations where Emergo derives empty interfaces can provide valuable information to users, indicating that they can stop their search; a fact that we did not evaluate yet.

5.4 Threats to validity

As in every experiment, there are several threats to validity. Most importantly, our experiment is limited to a specific implementation technique and specific code change scenarios, as discussed in Section 5.3; generalization to aspect-oriented languages and others requires further investigation; generalization to arbitrary maintenance tasks is not intended.

Second, our code change tasks are relatively simple (they take only few minutes to be accomplished). Nevertheless, we can still find bug reports regarding undeclared, uninitialized, and unused problems of single variables as well as of their uses along the code.[6] Moreover, 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. Thus, we might carefully extrapolate our results to bigger tasks as well. Also, we analyze tasks on unfamiliar source code, whereas in practice developers might remember cross-feature dependencies from knowledge gained in prior tasks.

Furthermore, recruiting students instead of professional developers as participants threats external validity. 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 [10], 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.

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

Emergent interfaces depend on data-flow 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, precalculation in the background or in nightly builds, 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 seems 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] See `https://bugzilla.gnome.org/show_bug.cgi?id=580750`, `https://bugzilla.gnome.org/show_bug.cgi?id=445140`, `https://bugzilla.gnome.org/show_bug.cgi?id=309748`, `https://bugzilla.gnome.org/show_bug.cgi?id=461011`

6. RELATED WORK

Preprocessor-based variability.

Preprocessor-based variability is common in industry (in fact likely the most common implementation form for product lines), even though its limitations regarding feature modularity are widely known and criticized [41, 13]. In this implementation form no interfaces exist between features.

Emergent interfaces follow a line of research works that try to provide tool-based solutions to help practitioners cope with existing preprocessor-infested code. Virtual separation was explored with the tool CIDE, which can hide files and code fragments based on a given feature selection [23]. The version editor [3], C-CLR [40], and the Leviathan file system [20] 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 [27], 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.

Along those lines, several researchers investigated close-world whole-product-line analysis techniques that can type check or model all configurations of a product line in an efficient way [45, 25, 26]. The underlying analysis of Emergo follows the general idea of whole-product-line analysis, but extends prior work this to data-flow analysis.

In our evaluation, we investigated only the influence of emergent interfaces, but not of other facets of preprocessor usage or virtual separation, which have been explored in prior complementary studies. Specifically, Feigenspan et al. have shown in a series of controlled experiments that different representations of conditional-compilation annotations can improve program comprehension [14]. Furthermore, Le et al. have shown in a controlled experiment that hiding irrelevant code fragments can improve understanding of product lines [30]—a result that analyzes with an ex-post analysis of using the version editor showing significant productivity increases [3]. These results complement each other and help building a tool environment for efficient maintenance of preprocessor-based implementations.

Feature modularity.

Separating concerns in the implementation and hiding their internals has a long history in software-engineering research [34] and programming language design [32]. The research field has received significant attention with the focus on crosscutting concerns in the context of aspect-oriented programming [28].

Early work on aspect-oriented programming was often

criticized for neglecting modularity with clear interfaces [44, 42], whereas more recently many researchers have investigated how to add additional interface mechanisms [1, 43, 17, 21], typically adding quite heavyweight language constructs. In contrast, our idea relies on tools to emerge interfaces on demand. So, developers do not need to write them in advance.

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.

Emergent interfaces pursue an alternative, tool-based strategy, leaving the languages as is (at least until mainstream languages support modular crosscutting implementations), but providing tool support for developers. Eventually both directions may converge by using emergent interfaces to infer interfaces (similar to type inference, and with similar tradeoffs).

Overall, 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 can 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, 43, 17, 12], but it can provide an interesting compromise between specification effort and usability [24].

Hidden dependencies.

Hidden dependencies are known to be problematic. This can be traced back to avoiding global variables [46], 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 [47], 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.

Prior work on emergent interfaces.

We have first proposed emergent interfaces in an Onward! paper [37]. The prototype tool we introduced was based on CIDE [23] to annotate features and the reaching-definition analysis was approximated and unsound. It was neither *interprocedural* nor even feature-sensitive, checking only whether the maintenance-point annotation was different of the reached statements' annotation. Subsequently, we assessed how often cross-feature dependencies occur in practice by mechanically mining 43 software systems with preprocessor variability [38], using an srcML-based infrastructure [31] conservatively approximating *intraprocedural* data-flow using proxy metrics (unsound, but sufficient to approximate the frequency of the problem). Furthermore, we estimated potential effort reduction by a tool like Emergo, by simulating code change tasks: we randomly selected variables from the 43 systems and estimated developers effort by counting how many `#ifdef` blocks they would analyze with and without emergent interfaces, showing a potential for significant reduction. In another branch of work, we investigated precise and efficient mechanisms for *feature-sensitive* data-flow analyses [8, 9, 6]. These advances now form the technical infrastructure from which emergent interfaces are computed precisely (without unsound approximations of prior work). In this paper, we bring together these results and focus on the originally envisioned application: emergent interfaces. We present a significantly revised and extended version of Emergo that uses *intraprocedural* and *interprocedural* analysis, and, for the first time, evaluate the actual benefit of emergent interfaces for code change tasks in a controlled experiment with human participants.

7. CONCLUDING REMARKS

In this paper we provide a full comprehension of emergent interfaces that emulate missing interfaces in many product line implementations. We provide a complete version of a tool capable of computing interfaces from data-flow analysis on demand. Emergent interfaces raise awareness of feature dependencies that are critical for maintaining (variable) software systems. With a conducted and replicated controlled experiment, we evaluate to what extent such tool support can help on achieving better feature modularization. Our study focuses on feature code change 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 code change effort by emergent interfaces, when faced with *interprocedural* dependencies. Similarly, our study suggests a reduction in errors made during those code change tasks. In future work, we will focus on scaling the underlying data-flow analysis by trading off performance and precision, and investigate emergent interfaces for other implementation techniques.

8. 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] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering (TSE)*, 30(6):355–371, June 2004.

[6] 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.

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

[8] 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.

[9] C. Brabrand, M. Ribeiro, T. Tolêdo, J. Winther, and P. Borba. Intraprocedural dataflow analysis for software product lines. *Transactions on Aspect-Oriented Software Development (TAOSD)*, 10:73–108, 2013.

[10] 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.

[11] 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.

[12] 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.

[13] J.-M. Favre. The CPP paradox. In *Proc. of the European Workshop on Software Maintenance*, 1995.

[14] 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.

[15] 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.

[16] R. E. Filman and D. P. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. In *Workshop on Advanced Separation of Concerns (OOPSLA)*, October 2000.

[17] 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.

[18] F. Heidenreich, J. Kopcsek, and C. Wende. FeatureMapper: Mapping features to models. In *In Comp. of the 30th International Conference on Software Engineering (ICSE)*, pages 943–944, New York, 2008. ACM Press.

[19] T. Hill and P. Lewicki. *Statistics: Methods and Applications. A Comprehensive Reference for Science, Industry, and Data Mining*. StatSoft, Tulsa, OK, USA, 1st edition, 2006.

[20] W. Hofer, C. Elsner, F. Blendinger, W. SchrËEder-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.

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

[22] M. Inostroza, E. Tanter, and E. Bodden. Join point interfaces for modular reasoning in aspect-oriented programs. In *Proc. of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (ESEC/FSE)*, pages 508–511, New York, NY, USA, 2011. ACM.

[23] 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.

[24] 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.

[25] 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.

[26] C. Kästner, K. Ostermann, and S. Erdweg. A variability-aware module system. In *Proc. of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 773–792, New York, NY, 10 2012.

[27] 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.

[28] 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.

[29] S. Krishnamurthi, K. Fisler, and M. Greenberg. Verifying aspect advice modularly. *ACM SIGSOFT Software Engineering Notes*, 29(6):137–146, October 2004.

[30] 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.

[31] 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.

[32] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. Abstraction mechanisms in clu. *Communications of ACM*, 20(8):564–576, 1977.

[33] L. Mikhajlov and E. Sekerinski. A study of the fragile base class problem. In *Proc. of the 12th European Conference on Object-Oriented Programming (ECOOP)*, pages 355–382. Springer-Verlag, 1998.

[34] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *CACM*, 15(12):1053–1058, 1972.

[35] H. Rajan and G. T. Leavens. Ptolemy: A language with quantified, typed events. In *Proc. of the 22nd European conference on Object-Oriented Programming (ECOOP)*, pages 155–179, Berlin, Heidelberg, 2008. Springer-Verlag.

[36] 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.

[37] 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.

[38] 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.

[39] I. Schaefer, L. Bettini, F. Damiani, and N. Tanzarella. Delta-oriented programming of software product lines. In *Proc. of the 14th international conference on Software product lines (SPLC)*, pages 77–91, Berlin, Heidelberg, 2010. Springer-Verlag.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

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

[45] 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.

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

[47] 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.

APPENDIX

A. ADDITIONAL MATERIALS

All data, materials, tasks, plug-ins, and R scripts can be found at `http://www.cin.ufpe.br/~mmr3/icse2014/`

B. ANOVA TABLES

In this appendix, we provide all ANOVA results.

	Df	Sum Sq	Mean Sq	F value	*p-value*
Replica	4	276682	69171	2.3158	0.1451
Replica:Subject	5	193989	38798	1.2989	0.3528
SPL	1	5184	5184	0.1736	0.6879
Technique	1	2298420	2298420	76.9504	**2.237e-05**
Residuals	8	238951	29869		

Table 1: Round 1 - New-requirement task.

	Df	Sum Sq	Mean Sq	F value	*p-value*
Replica	6	414.16	69.03	1.9960	0.1452
Replica:Subject	7	390.18	55.74	1.6118	0.2230
Feature	1	2.88	2.88	0.0833	0.7778
Technique	1	1286.46	1286.46	37.2003	**5.343e-05**
Residuals	12	414.99	34.58		

Table 2: Round 2 - New-requirement task.

	Df	Sum Sq	Mean Sq	F value	*p-value*
Replica	4	57255	14314	0.4905	0.7435
Replica:Subject	5	260331	52066	1.7841	0.2223
Feature	1	61272	61272	2.0996	0.1854
Technique	1	82818	82818	2.8379	**0.1306**
Residuals	8	233465	29183		

Table 3: Round 1 - Unused-variable task.

	Df	Sum Sq	Mean Sq	F value	*p-value*
Replica	6	81.516	13.586	0.8605	0.549665
Replica:Subject	7	267.751	38.250	2.4227	0.085284
Feature	1	173.000	173.000	10.9578	0.006221
Technique	1	123.068	123.068	7.7951	**0.016285**
Residuals	12	189.455	15.788		

Table 4: Round 2 - Unused-variable task.