Feature Maintenance with Emergent Interfaces

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

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

Christian Kästner
Carnegie Mellon University
Pittsburgh, USA

ABSTRACT

Hidden code dependencies are responsible for many complications in maintenance tasks. With the introduction of variable features in configurable systems, dependencies may even cross feature boundaries, causing problems that 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 tool-based solution to support developers in recognizing and dealing with feature dependencies: emergent interfaces. Emergent interfaces are inferred on demand, based on feature-sensitive *intraprocedural* and *interprocedural* data-flow analysis. They emerge in the IDE and emulate modularity benefits 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.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques

General Terms

Experimentation

Keywords

Product Lines, Interfaces, Preprocessors, Controlled Experiments

1. INTRODUCTION

During maintenance, developers often introduce errors into software systems when they fail to recognize module and feature dependencies [10]. 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 [29] 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 [48] between separate feature developers, changing one feature might be the correct action for maintaining that feature, but might bring undesirable consequences to the behavior of other features. 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 in which features were implemented using the C preprocessor, we found that cross-feature dependencies are frequent in practice [38].

To reduce this feature-dependency problem, we propose a technique called *emergent interfaces* (introduced in a vision paper [37]) that establishes interfaces for feature code on demand and according to a given code-change task. An emergent interface is an abstraction of the data-flow dependencies of a feature, consisting of a set of provides and requires clauses that describe such dependencies. We call our technique *emergent* because, instead of writing interfaces, developers 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 a better chance of avoiding errors [49]. Emergent interfaces may also reduce code-change effort: Instead of searching for dependencies throughout the code and reasoning about requirements-level feature constraints, developers can rely on proper tool support to infer interfaces. We implemented emergent interfaces in a tool, *Emergo*, available as an Eclipse plug-in for Java. Emergo performs feature-sensitive data-flow analysis to infer 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 two software product lines. The studied product lines are implemented with preprocessor-like variability mechanisms, which are widely used to implement compile-time variability in industrial practice, despite their lack of modularity. 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. We first conducted the experiment in one institution, recruiting graduate students as subjects, and then replicated it with undergraduate students in another institution.

Our experiment reveals that, in our settings, emergent interfaces significantly reduce maintenance effort for tasks involving *inter-*

procedural feature dependencies, which cross method boundaries. Both experiment rounds reveal that developers were, on average, 3 times faster completing our code-change tasks when using emergent interfaces. As for tasks involving only *intraprocedural* dependencies, differences are statistically significant only in one round, in which we on average observe a 1.6 fold improvement in favor of emergent interfaces. In line with recent research [49], in both rounds we observe that presenting feature dependencies helps developers detecting and avoiding errors, regardless of the kind of dependency.

In summary, we make the following contributions:

- An introduction to emergent interfaces and a complete implementation in Emergo, supporting not only *intraprocedural* analysis, but also the 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, demonstrating significant potential.

The idea of emergent interfaces was first introduced in a vision paper, with an early prototypical implementation approximating *intraprocedural* data-flow analysis [37]. Subsequently, we statically analyzed the potential impact of emergent interfaces in 43 open source projects [38] and investigated feature-sensitive data-flow analyses [7]. 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 complements them with a novel empirical evaluation.

2. MAINTAINING PRODUCT LINES

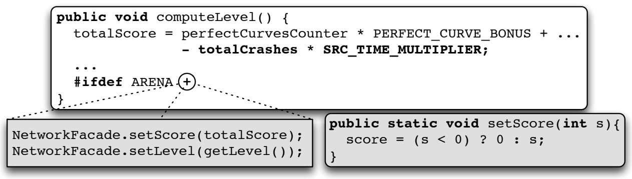
Inadequate modularity mechanisms plague many languages and cause many implementation problems. Emergent interfaces are applicable to many situations where explicit interfaces between code fragments are lacking. While we will hint at many other use cases, to illustrate and explore the idea of emergent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

Configurable 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 potential number of them, rapid feedback by testing is often too expensive or even 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 motivates players to achieve the best circuit lap time and therefore qualify for the pole position. Due to portability constraints, the game is developed as a product line and is deployed on 65 different devices. 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 (non-annotated code, or annotated with mandatory features), 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 products.



```
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: 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` checks for positive values and prevents submitting negative scores to the network server.

The cause of the problem is that the *ARENA* implementation extends the score behavior and is therefore affected by the change. 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 enforce the invariant that 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 small. 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 introduce errors—that potentially only manifest in few variants—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 and conditional-compilation constructs. Developers report that a variable `status` in the common code is reported as unused. In fact, such warnings are commonly found in many bug reports of preprocessor-based systems.[4] 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

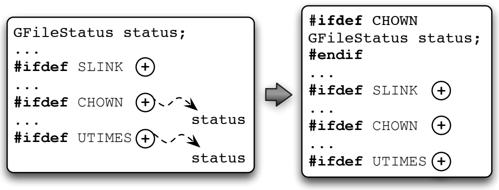
[1]`http://www.meantime.com.br/`

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

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

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

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 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 might, 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. These are easy to detect in small and simple methods with little variability, but might be complicated in larger code bases and when using other language mechanisms that separate feature code (see Section 2.4).

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 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 distinct code units 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 enabling flexibility [15]. To mitigate the problem, several extensions to aspect-oriented languages have been proposed to declare interfaces between concerns [43, 21, 1, 16, 35, 32]. Without such interfaces in AspectJ, our 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* [4], *delta-oriented programming* [39], or even just *subclassing* [31] 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 [22, 2, 28, 17]. For example, in CIDE [22], 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 [2, 28]. 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. In this sense, 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 code might declare a variable subsequently used by an optional feature.

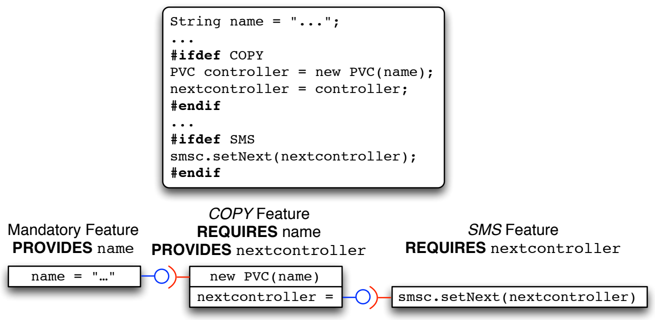
There are several paths to attack this problem of cross-feature dependencies. The typical language-designer approach is to introduce additional modularity concepts into the programming language and make control-flow and data-flow explicit in interfaces [27]. With *emergent interfaces*, we pursue an alternative tool-based direction, which works with existing languages and existing implementations and infers interfaces on demand.

An emergent interface is an abstraction of the data-flow dependencies of a feature. It consists of a set of provides and requires clauses that describe such dependencies. In this work, we consider cross-feature dependencies as data dependencies arising from definitions introduced by a feature and used in others, and vice-versa. Each provides and requires clause expresses data-dependency information obtained through *def-use* chains. So, an emergent interface states that features provide data to others and require data from others, making explicit data dependencies between the involved features.

For example, when considering feature *COPY* in Figure 3—from the *MobileMedia* product line [14]—, the emergent interface at the bottom states that *COPY requires* data from the common code (the mandatory feature) through the `name` variable; and that *COPY provides* data to the *SMS* feature. This way, when maintaining a feature, developers may ask for an emergent interface to be aware of cross-feature dependencies. Now, they can prevent situations where features would not work properly, like when assigning a new controller to `nextcontroller` not suitable for the *SMS* feature.

3.1 Inferring Emergent Interfaces on Demand

Like every ordinary interface, an emergent interface abstracts



```
String name = "...";
...
#ifdef COPY
PVC controller = new PVC(name);
nextcontroller = controller;
#endif
...
#ifdef SMS
smsc.setNext(nextcontroller);
#endif
```

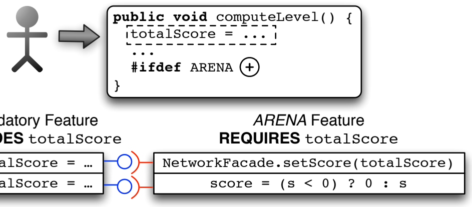
Figure 3: Emergent interface for the *COPY* feature.

implementation details—feature implementation details—, exposing only the data that features provide to and require from each other. Differently from ordinary interfaces, an emergent interface does not need to be written manually by developers. Instead, they are inferred and emerge on demand in the IDE to improve feature understanding and give support for specific development tasks. When analyzing the inferred interface, developers become aware of cross-feature dependencies and may have better chance of not introducing errors to other features.

Despite improving feature understanding, emergent interfaces do not provide guarantees such as stable contracts or means to detect violations as normal written interfaces do. However, we argue that writing and maintaining potentially large, fine-grained, and low-level interfaces between crosscutting features that may change frequently is a hard task. With emergent interfaces, developers get rid of this task, because interfaces are inferred. Besides, it is important to infer interfaces *on demand*. As mentioned, cross-feature dependencies are common in practice [38], so, inferring interfaces in the first place for entire features will probably yield a large set of provides and requires clauses, being difficult to read and understand. In contrast, inferring interfaces on demand from parts of a feature implementation improves readability and understandability and can help developers to focus on a specific local task.

To better illustrate how emergent interfaces are inferred from parts of a feature implementation, we return to *Scenario 1* from the previous section, where the developer is supposed to change how the total score is computed. Before changing this computation, the developer may ask for an emergent interface to support this particular code change task. To ask for an interface, developers select part of a feature implementation, possibly consisting of non-contiguous code blocks within a method, that contains definitions or uses of variables. Interpreted as *maintenance points*, the selection drives our tool (see Section 3.4) to obtain cross-feature dependencies between these points and the other features. In this context, emergent interfaces help developers to make code changes once they identify the maintenance points. They do not contribute to finding the maintenance points in the first place, though.

In our example, the developer is only interested in changing the total score computation. So, she selects only part of the mandatory feature implementation (see the dashed rectangle in Figure 4). Then, data-flow analyses are performed to capture cross-feature dependencies between the feature part she is maintaining and the other features. Finally, the interface emerges stating that there is data provided by the common code (the mandatory feature) that is required by the *ARENA* feature—the `totalScore` current's value. Figure 4 illustrates the emergent interface for this scenario. The interface is an abstraction of the data dependencies between part of the common code (the `totalScore` assignment) and *ARENA*, containing two provides and requires clauses (see the bottom of Figure 4). The first clause states that the mandatory feature provides data to the *ARENA* feature, which uses the value of `totalScore` when calling the `setScore` method. The second clause states that this data also reaches a ternary statement, which is inside the `setScore` method. This happens due to the binding between `totalScore` and `s`, according to Figure 1. Thus, the code change task may impact the behavior of products containing the *ARENA* feature. The developer is now aware of cross-feature dependencies. When investigating them, she is likely to discover she also needs to modify the *ARENA* code to avoid introducing an error. Finally, note that there is no data from other features required by the `totalScore` assignment at the mandatory feature, supporting hiding or separating those features.



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

Figure 4: Emergent interface for Scenario 1.

3.2 Abstracting Feature Code

As mentioned, emergent interfaces abstract feature implementation details. For example, although *ARENA* has many lines of code scattered throughout many files, the emergent interface focuses only on the lines that indeed are data dependent on the `totalScore` value, helping developers to abstract the remaining lines and files during this particular task.

In addition, emergent interfaces can help to prevent developers from analyzing unnecessary features and their associated code, which is important to decrease code change effort. In particular, we believe that our interfaces can help on making the idea of virtual separation of concerns (see Section 2.4) realistic. That is, we can hide features and rely on emergent interfaces to only show the ones we need. For instance, consider *Scenario 2* (Section 2.2). Here, an emergent interface would show that only *CHOWN* and *UTIMES* features require `status`. So, we could keep *SLINK* hidden, since it is not related to the current task.

3.3 Avoiding Invalid Dependencies

Product lines may establish explicit requirements-level feature constraints in terms of variability models [34]. By using these models, it is possible to evaluate whether a feature combination is valid or not. Emergent interfaces work both in the presence and in the absence of variability models. If they are available, emergent interfaces can take them into account, preventing developers from reasoning about requirements-level feature constraints and even from assuming invalid cross-feature dependencies in case of mutually exclusive features (which may cause potential errors). For example, with two mutually exclusive features *A* and *B*, developers might assume that changing the `x = 0` assignment in feature *A* may lead to problems in feature *B*, which contains the `m(x)` statement. However, since the involved features are mutually exclusive, we have an empty interface: There is no data with respect to the `x` variable from feature *A* that reaches feature *B*, and vice-versa in any valid feature combination. So, code-change tasks in the former feature do not impact the latter.

3.4 Implementation: Emergo

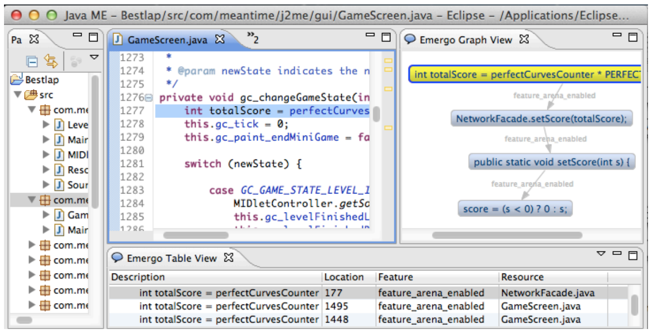
We implemented the idea of emergent interfaces in an Eclipse plug-in named Emergo. Emergo computes emergent interfaces based on feature dependencies between methods or within a single method, by using *interprocedural* or *intraprocedural* feature-sensitive data-flow analysis [7, 5]. More specifically, we use the *reaching definitions* analysis through *def-use* chains. The feature-sensitive approach is capable of analyzing all configurations of a product line without having to generate all of them explicitly.

Although our examples refer to code that implement features, Emergo is actually more general and considers code associated with feature expressions such as "*B && C*". Also, because we use the feature-sensitive approach, Emergo analyzes the *def-use* chains of each configuration. So, Emergo seeks for definitions of a feature used by another for each configuration. For example, if `int x = 0` is the maintenance point (see the code snippet to the right), there is a cross-feature dependency when $\neg A \land B \land C$, since data is required by `m(x)` in such a configuration. In case there is a variability model available, Emergo checks if this dependency is possible or not. If yes, it shows such a dependency at the tool UI.

```
int x = 0;
#ifdef A
x = 1;
#endif
#ifdef B && C
m(x);
#endif
```

The capacity of analyzing all products without the need to generate them in a brute-force fashion increases performance [7, 5], 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 compare 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 [8].

Figure 5 shows a screenshot of Emergo. After the developer found and selected the maintenance point in Line 1277 (the `totalScore` assignment), Emergo shows an emergent interface using a table view and a graph view. We read the first row of the table as follows: there is data (the `totalScore` value) provided by the common code that is required by the *ARENA* feature in Line 177 of the `NetworkFacade` class.



```
1273    *
1274    * @param newState indicates the n
1275    */
1276 private void gc_changeGameState(in
1277     int totalScore = perfectCurves
1278     this.gc_tick = 0;
1279     this.gc_paint_endMiniGame = fa
1280
1281     switch (newState) {
1282
1283         case GC_GAME_STATE_LEVEL_I
1284             MIDletController.getSc
1285             this.gc_levelFinishedL
```

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

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

Figure 5: Using Emergo for Scenario 1.

Initially, Emergo shows all cross-feature dependencies in both views. To focus on a particular dependency, the developer can click on the corresponding table row, and then Emergo shows the data path associated with the dependency of that row in the graph view.

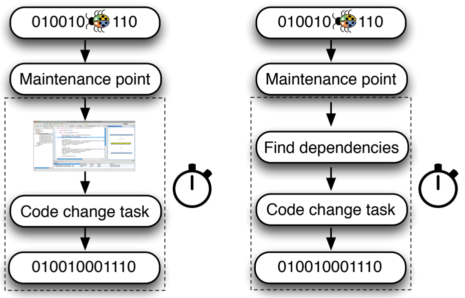


Figure 6: Dashed rectangles represent the time we count (with and without emergent interfaces).

In our example in Figure 5, the developer selected the first row of the table, so the graph shows the path from the maintenance point to Line 177 of the `NetworkFacade` class. Emergo also provides navigation support either by clicking on the table rows or on the graph nodes, enabling developers to quickly jump within in the IDE.

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 investigate and compare code-change 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 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 6 illustrates our setup with and without emergent interfaces. Note that we do not measure the time needed to find the maintenance point (we actually provide the maintenance point with our task description as we describe later). While finding the maintenance point may represent a significant part of a code-change task 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, eliminating noise that would not contribute to our analysis.

To answer Question 2 (correctness), we measure how many incorrect solutions a developer committed during a code change task (number of errors). We consider all human actions as an error that introduce one or more defects into the code. As described later, for some tasks, we provide automated feedback to the experiment participants, so a participant can continue the task 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 three times. 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—attendants of a course on experimental software engineering lead by an independent lecturer—at Federal University of Pernambuco, Brazil (*Round 1*). Finally, we replicated the experiment with 14 UROP (Undergraduate Research Opportunity Program) students at Federal University of Alagoas, Brazil (*Round 2*). In both rounds, around half of the participants were part-time students with professional experience—varying from few months to many years of experience. All participants 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-configurable commercial product line that has about 15 KLOC. The latter, which has about 3 KLOC, is an academic product line for applications that manipulate photo, music, and video on mobile devices [14]. It contains feature restrictions and has been used in previous studies [14, 36].

We ask participants to perform a number of code-change tasks in each of the product lines. 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 affected by cross-feature dependencies, since this is the kind of context where Emergo can help. Note that emergent interfaces target a specific class of problems; for other maintenance tasks we would not expect any benefit. We argue that our task selection represents 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 have tasks that require participants to implement a new requirement (requiring *interprocedural* analysis of the existing source code) and tasks that require participants to 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 next.

Task 1 - New requirement for Best Lap. The new requirement for *Best Lap* is similar to our motivating *Scenario 1*: 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 cross-feature 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 `totalScore` and the rest of the code. Otherwise, the participant is free to use standard tools like find/replace and highlighting. This setup 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 cross-feature dependencies, and update conditional statements (here, only one `if` statement). However, in contrast to 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 unused-variable warnings for two variables: `tires` and `xP`. 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.

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 tasks 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 use Emergo at the start of the experiment. We performed warmup tasks together in a class context but did not evaluate their results.

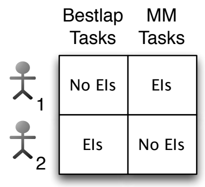
4.4 Hypotheses

Based on our goals and tasks, we evaluate the following hypotheses: *(H1 - Effort)* With emergent interfaces, developers spend less time to complete 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 code-change tasks involving feature dependencies.

4.5 Design

To evaluate our hypotheses, 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 (see figure to the right). As a result, each participant performs both kinds of tasks on each product line and has the opportunity to use emergent interfaces in one of the tasks. No participant performs 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 ANOVA. To give relevance to the ANOVA [6], we use the Bartlett, Box Cox, and Tukey tests to verify variance homogeneity, normal distribution,

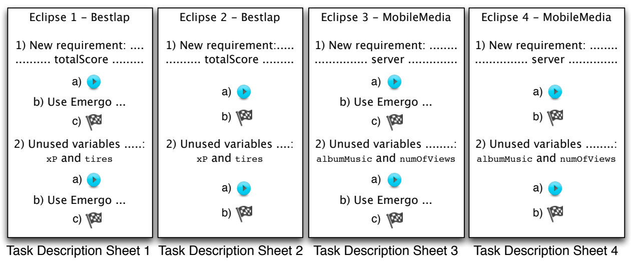

Eclipse 1 – Bestlap
1) New requirement: .... totalScore ....
a)
b) Use Emergo ...
c)
2) Unused variables .....: xP and tires
a)
b) Use Emergo ...
c)
Eclipse 2 – Bestlap
1) New requirement:..... .... totalScore ....
a)
b)
2) Unused variables ....: xP and tires
a)
b)
Eclipse 3 – MobileMedia
1) New requirement: .... server ....
a)
b) Use Emergo ...
c)
2) Unused variables ....: albumMusic and numOfViews
a)
b) Use Emergo ...
c)
Eclipse 4 – MobileMedia
1) New requirement: .... server ....
a)
b)
2) Unused variables ....: albumMusic and numOfViews
a)
b)
Task Description Sheet 1
Task Description Sheet 2
Task Description Sheet 3
Task Description Sheet 4

Figure 7: Task description sheets we distributed.

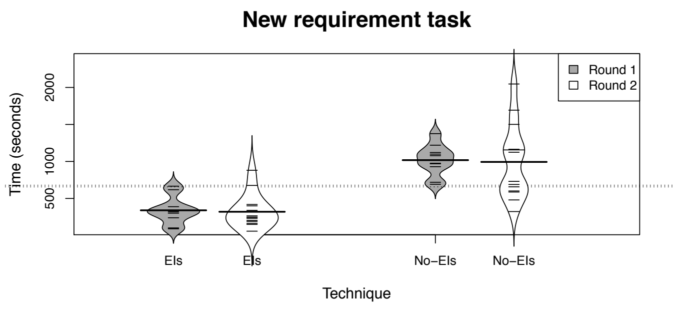


Figure 8: Time results for the new-requirement task.

and model additivity, respectively. We follow the convention of considering a factor as significant when *p-value* < 0.05.

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 maintenance point (a variable assignment). 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 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 the task description sheets we distributed in Figure 7. We represent the steps that participants should follow as "a", "b", and "c". Notice that we associate each sheet with a different Eclipse.

We also partially automated measuring the number of errors. For new-requirements tasks (Tasks 1 and 2, presented in Section 4.3), the plug-in automatically checks the submitted solution by compiling the code and running test cases (require about 1 second), as soon as a 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 whether one or both variables are correctly fixed after the experiment. 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*), 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. In each round, all participants worked at the same time in the same room under the supervision of two experimenters.

4.7 Execution and Deviations

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 (All data, materials, tasks, plug-ins, and R scripts are available at `http://twiki.cin.ufpe.br/twiki/bin/view/SPG/EmergentInterfaces`). We proceed separately with the two kinds of tasks, reporting results from both rounds.

5.1 New-Requirement Tasks

We plot the times for both new-requirement tasks (1 and 2) 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. According to an ANOVA test, we obtain statistically significant evidence that our interfaces reduce effort in both new-requirement tasks, *p-value = 2.237e-05*. The key results were confirmed in the replication (participants with emergent interfaces were, on average, 3.1 times faster), *p-value = 5.343e-05*.

In Figure 9, we plot the number of errors 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

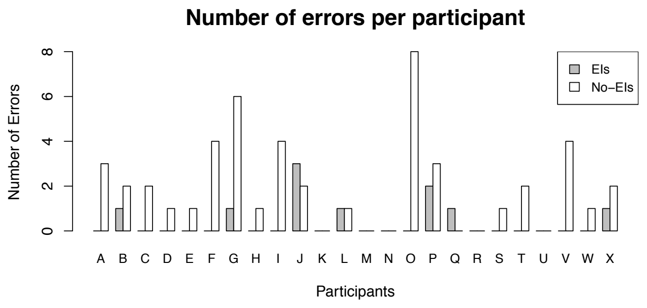


Figure 9: Number of errors for the new-requirement task in both rounds. A-J: Round 1; K-X: Round 2.

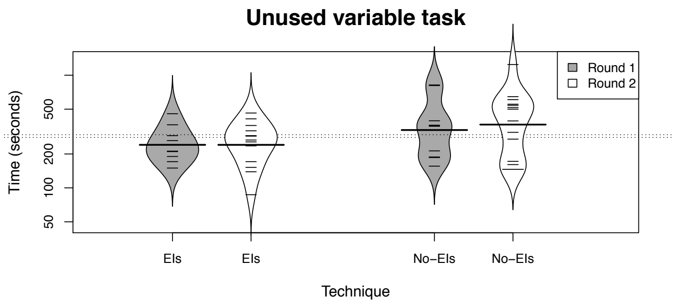


Figure 10: Time results for the unused-variable task.

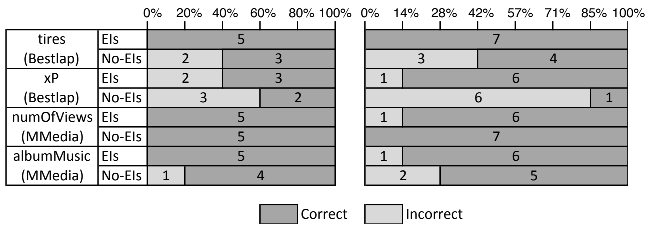


Figure 11: Number of errors for the unused-variable task.

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 statistically 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 took 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, 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 also analyze measurements with an added 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. We add 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 [18]).

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 our interfaces reduce effort only in the second round (*p-value = 0.016*; for the first round, *p-value = 0.1306*). The statistical results are stable for the original time (time until they think they are done) and the adjusted time. Regarding the adjusted time in the second round, 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 adjusted 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 in both rounds.

We plot the number of errors 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.

5.3 Meta-Analysis

To identify other tendencies, we also performed a meta-analysis, where we combine the results from both rounds. The time differences are statistically significant for both kinds of tasks. Here we used the adjusted time for the unused variable task.

5.4 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 cross-feature 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. Our conclusion is 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 (*inter* or *intraprocedural*, method size, 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 [49]. 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 the problematic feature combination. Due to 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 [49]: "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 observe again that the longer methods of *Best Lap* are more prone to errors than the shorter methods of *MobileMedia*.

Outlook. Our experiment considered the influence of emergent interfaces in a specific scenario of preprocessor-based product lines. As we argued in Section 2, the idea 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 tasks involving cross-feature dependencies in aspect- or feature-oriented implementations.

Also, emergent interfaces have capabilities we did not explore in our experiment. For instance, product lines often make explicit requirements-level feature constraints in terms of variability models. To identify a cross-feature dependency, the underlying feature-sensitive data-flow analysis of Emergo can take such constraints into account mechanically and automatically exclude infeasible paths, whereas a developer needs to manually compare `#ifdef` annotations on code statements with constraints specified in these models. Further, 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.

5.5 Threats to Validity

First, our experiment is limited to a specific implementation technique and specific code-change scenarios; 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 accomplish). Nevertheless, we can often find bug reports[5] regarding undeclared, uninitialized, and unused problems of single variables as well as of their uses along the code in practice. Moreover, we assume that many bigger tasks can be seen as a sequence of smaller tasks such as the ones we consider here. So, if we provide benefits for small tasks, it is plausible to consider that we can sum up their times and observe benefits for the whole task. Thus, we might carefully extrapolate our results to some kinds of bigger tasks as well. Also, we analyze tasks on unfamiliar code, whereas in practice developers might remember cross-feature dependencies from knowledge gained in prior tasks.

Third, recruiting students instead of professional developers 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 [9], 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.

Fifth, emergent interfaces depend on data-flow analysis, which can be 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 we add to wrong answers for the unused-variable task might be controversial, because 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 with both the original and adjusted data. In addition, not considering the time to find the maintenance points might raise a threat in the sense that developers could learn about feature dependencies in advance when finding these points.

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

Preprocessor-based variability. Preprocessor-based variability is common in industry, even though its limitations regarding feature modularity are widely known and criticized [41, 12]. In this implementation form, no interfaces exist between features. Emergent interfaces follow a line of research that tries 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 [22]. The version editor [2], C-CLR [40], and the Leviathan file system [19] show projections of variable source code along similar lines. Similar ideas have also been explored outside the product-line

[5] `https://bugzilla.gnome.org/show_bug.cgi?id=580750, 445140, 309748, and 461011`

context most prominently in Mylyn [25], which learns from user behavior and creates task-based views. Also in this context, emergent interfaces can help to make dependencies visible.

Along those lines, 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, 24, 46]. The underlying analysis of Emergo follows the general idea of whole-product-line analysis, but extends prior work 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 [13]. Further, Le et al. have shown in a controlled experiment that hiding irrelevant code fragments can improve understanding of product lines [28]—a result that aligns with an ex-post analysis of using the version editor, showing productivity increases [2]. 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 [33] and programming language design [30]. The research field has received significant attention with the focus on crosscutting concerns in the context of aspect-oriented programming [26]. 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, 16, 20, 32], typically adding quite heavyweight language constructs. In contrast, our idea relies on tools to infer interfaces on demand. So, developers do not need to write them in advance.

Conceptual Modules [3] support analyzing the interface of a specific module—also using *def-use* chains internally. Our idea extends conceptual modules by considering feature relationships. Where conceptual modules were evaluated regarding correctness in case studies, we contribute a controlled experiment to evaluate correctness and reduced effort. Slicing [47] also uses data-flows but requires program transformations to yield a executable program (not feature aware) to support maintenance tasks. Emergent interfaces pursue an alternative, tool-based strategy with no program transformations, 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 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, 16, 11], but it can provide an interesting compromise between specification effort and usability [23].

Hidden dependencies. Hidden dependencies are known to be problematic. This can be traced back to avoiding global variables [48], 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 [49], 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). Current mainstream 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 a vision paper [37]. The prototype tool that we introduced was based on CIDE [22] 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. Then, 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 [29] conservatively approximating *intraprocedural* data-flow using proxy metrics (unsound, but sufficient to approximate the frequency of the problem). Further, 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 parallel, we investigated precise and efficient mechanisms for *feature-sensitive* data-flow analyses [7, 8, 5]. These advances now form the technical infrastructure with which emergent interfaces are inferred 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 our interfaces for code-change tasks in a controlled experiment with human participants.

7. CONCLUDING REMARKS

In this paper, we present emergent interfaces that emulate missing interfaces in many product-line implementations. We provide Emergo, a complete version of a tool capable of inferring interfaces from data-flow analysis on demand. Emergent interfaces raise awareness of cross-feature dependencies that are critical for maintaining (configurable) software systems. With a conducted and replicated controlled experiment, we evaluate to what extent such tool support can help 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 investigating emergent interfaces for other implementation techniques.

8. ACKNOWLEDGMENTS

We would like to thank CNPq and the National Institute of Science and Technology for Software Engineering (INES), funded by CNPq and FACEPE, grants 573964/2008-4 and APQ-1037-1.03/08, for partially supporting this work (`http://www.ines.org.br/`). Kästner's work was partially supported by ERC grant #203099 and NSF grant CCF-1318808. Also, we thank SPG (`http://www.cin.ufpe.br/spg`) members for the fruitful discussions about this paper. We also thank Janet Siegmund and all anonymous reviewers for the comments that certainly helped to improve the paper.

9. 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, 2005.

[2] 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.

[3] 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. IEEE Computer Society, 1998.

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

[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 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 355–364. ACM, 2013.

[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. ACM, 2012.

[8] 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.

[9] R. P. Buse, C. Sadowski, and W. Weimer. Benefits and barriers of user evaluation in software engineering research. In *Proc. of the 26th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 643–656. ACM, 2011.

[10] 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. ACM, 2011.

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

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

[13] 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*, 18(4):699–745, 2012.

[14] 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. ACM, 2008.

[15] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Proc. of the OOPSLA Workshop on Advanced Separation of Concerns*, 2000.

[16] 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.

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

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

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

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

[21] 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. ACM, 2011.

[22] 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. ACM, 2008.

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

[24] 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):14:1–14:39, 2012.

[25] 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. ACM, 2006.

[26] 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. Springer, 1997.

[27] S. Krishnamurthi, K. Fisler, and M. Greenberg. Verifying aspect advice modularly. In *Proc. of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 137–146. ACM, 2004.

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

[29] 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. ACM, 2010.

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

[31] 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.

[32] A. C. Neto, R. Bonifácio, M. Ribeiro, C. E. Pontual, P. Borba, and F. Castor. A design rule language for aspect-oriented programming. *Journal of Systems and Software*, 86(9):2333–2356, 2013.

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

[34] K. Pohl, G. Bockle, and F. J. van der Linden. *Software Product Line Engineering*. Springer, 2005.

[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. Springer-Verlag, 2008.

[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 *Comp. of the 25th ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 11–18. ACM, 2010.

[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 International Conference on Generative Programming and Component Engineering (GPCE)*, pages 23–32. ACM, 2011.

[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. Springer-Verlag, 2010.

[40] N. Singh, C. Gibbs, and Y. Coady. C-CLR: A tool for navigating highly configurable system software. In *Proc. of the AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, page 9. ACM, 2007.

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

[42] F. Steimann. The paradoxical success of aspect-oriented programming. In *Proc. of the 21st ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 481–497. ACM, 2006.

[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):1:1–1:43, 2010.

[44] M. Störzer and C. Koppen. PCDiff: Attacking the fragile pointcut problem. In *European Interactive Workshop on Aspects in Software*, 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. ACM, 2007.

[46] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A classification and survey of analysis strategies for software product lines. *ACM Computing Surveys*, 2014. To appear.

[47] M. Weiser. Program slicing. In *Proc. of the 5th International Conference on Software Engineering (ICSE)*, pages 439–449. IEEE, 1981.

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

[49] 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. ACM, 2011.