Emergent Feature Modularization

Márcio Ribeiro[1] **, Paulo Borba**[1] **(Advisor) , Claus Brabrand**[2] **(Co-Advisor)**

[1]Universidade Federal de Pernambuco (UFPE), Recife, Brasil

[2]IT University of Copenhagen (ITU), Copenhangen, Denmark

{mmr3, phmb}@cin.ufpe.br, brabrand@itu.dk

Abstract. *Many current implementation techniques for software families and product lines lack interfaces, which could make code level feature dependencies explicit and avoid errors introduction. As alternative to changing the implementation approach, we introduce a tool-based solution to support developers in recognizing and dealing with feature dependencies: emergent interfaces, which are computed on demand using feature-sensitive dataflow analysis. They emerge in the IDE and emulate benefits of modularity not available in the host language. In a controlled experiment, our interfaces improved performance of maintenance tasks by up to 3 times and also reduced errors introduction.*

Resumo. *Muitas técnicas para implementação de famílias e linhas de produtos de software carecem de interfaces, que poderiam explicitar dependências entre features e evitar introdução de erros. Nesse contexto, apresentamos uma solução baseada em ferramenta que permite aos desenvolvedores reconhecer e lidar com dependências entre features: interfaces emergentes, que são computadas sob demanda utilizando-se análises de fluxo de dados sensíveis a features e emergem no ambiente de desenvolvimento para emular benefícios de modularidade não presentes na linguagem de programação. Em um experimento controlado, desenvolvedores que usaram nossa abordagem durante tarefas de manutenção foram 3 vezes mais produtivos e introduziram menos erros.*

1. Introduction

Developers often introduce errors to software systems when they fail to recognize module and feature dependencies. This problem is particularly critical for families and Software Product Lines (SPLs), in which features can be enabled and disabled, and market and technical needs constrain how they can be combined. In this context, features often crosscut each other [Liebig et al. 2010] and share program elements like variables and methods [Ribeiro et al. 2011a, Ribeiro et al. 2012], without proper modular support from a notion of interface between features. So developers could easily miss dependencies, such as when a feature assigns a value to a variable read by another feature. Thus, changing the assigned value might be the correct action for maintaining one feature, but might bring undesirable consequences to the other one.

To reduce this feature dependency problem, we propose a technique named *emergent interfaces*, that establishes, on demand and according to a given maintenance task, interfaces to feature code [Ribeiro et al. 2010]. We call our technique "emergent" because, instead of writing interfaces manually, developers can request (by using our tool, Emergo [Ribeiro et al. 2011b]) interfaces on demand; that is, interfaces emerge to give support for specific maintenance tasks. To do so, emergent interfaces capture dependencies between the feature we are maintaining and the others we might impact. Thus, developers become aware of the dependencies, and may have better chance of not introducing errors [Yin et al. 2011]. To capture feature dependencies, we propose and implement feature-sensitive dataflow analysis [Brabrand et al. 2012], that analyzes families and SPLs in a less costly way than compiling and analyzing each product separately.

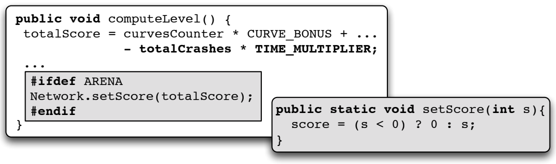
To evaluate the potential of emergent interfaces to reduce errors and development effort, we conducted and replicated a controlled experiment. We focus on the study of feature maintenance tasks in SPLs implemented with preprocessors, which are widely used to implement variability in industrial practice. Our experiment reveals that, considering the selected kinds of system and developers, emergent interfaces help to reduce effort for tasks involving *interprocedural* dependencies, which cross method boundaries. As for tasks involving only *intraprocedural* dependencies, we confirm statistical significance in only one round. In line with recent research [Yin et al. 2011], in both rounds we observe that presenting feature dependencies help developers to detect and avoid errors.

Contributions. The concept of emergent interfaces to support developers when maintaining features and the Emergo tool; data on how often feature dependencies occur in practice (to assess the problem relevancy); empirical evidence that our interfaces can reduce developers effort and errors introduction; and the feature-sensitive dataflow analysis. **Novelty.** Our idea is out of the mainstream (not human-produced interfaces) and was first published in Onward!, whose CFP looks for "*grand visions and new paradigms that could make a big difference in how we will one day build software.*" Our interfaces were the original motivation to feature-sensitive dataflow analysis, which lead to international collaborations and published papers in conferences with high impact factor, such as AOSD (one of the five best papers; invited and published in TAOSD) and PLDI (see `http://arnetminer.org/page/conference-rank/html/PL,SE.html`). **Awards.** Our PhD proposal was the best one at OOPSLA Doctoral Symposium 2010, worthing the ACM SIGPLAN John Vlissides Award (first brazilian awarded, see `http://www.sigplan.org/Awards/Vlissides/Main`). Also, Emergo was awarded as the best tool at Congresso Brasileiro de Software (CBSoft 2011).

2. Problem

To better explain the issues due to the lack of feature modularity, we outline one concrete scenario. Although we could illustrate these issues in different contexts, we choose a critical one: industrial software families and SPLs, that can easily have hundreds of features with a large number of possible products. In this context, code level feature dependencies may cross feature boundaries, so that a variable changed in one feature is read by another feature. If developers miss feature dependencies like this on, they might maintain a feature and actually break another one. This problem is shared by implementation approaches that support some form of crosscutting, such as aspects. In industrial practice, a more common scenario, and the one we focus here, is to use `#ifdefs`, where optional code fragments are merely annotated in base code [Liebig et al. 2010].

Motivating Example. Our example comes from the *Best Lap* commercial car racing game (developed by *Meantime Mobile Creations*) that motivate players to qualify for the pole position. To compute the score, developers implemented the variable `totalScore` to store the player's total score (see the figure to the right). Next to the common code base, there is optional code that belongs to feature *ARENA* (in gray). This feature publishes high scores on a network server and, due to resource constraints, is not available in all product configurations. To add penalties when crashing the car, suppose now that a developer has to let the game score be not only positive, but also negative. To accomplish the task, she localizes the *maintenance points*, here just the `totalScore` assignment, and changes its value (see the bold line). When executing products without the *ARENA* feature, we observe the desired behavior. But, due to the *ARENA* associated `setScore` method, users using a configuration that includes that feature may report that negative scores are not submitted to the network server. This was not noticed by the developer, who did not realize she had to change code of another feature: she would have to change part of the *ARENA* code to not check the invariant that all scores are positive. In this context, searching for feature dependencies might increase developers effort since they have to make sure that the modification does not impact other features. Also, if they miss a dependency, they can easily introduce errors by not properly completing a maintenance task, for example.



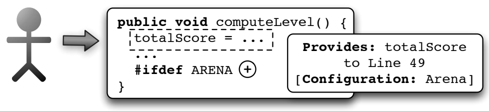
```
public void computeLevel() {
 totalScore = curvesCounter * CURVE_BONUS + ...
              - totalCrashes * TIME_MULTIPLIER;
 ...
 #ifdef ARENA
 Network.setScore(totalScore);
 #endif
}
```

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

Problem's Dimension. Whenever we have features sharing elements (i.e., variables, methods), we say that there is a *feature dependency* between the involved features. To assess how often these dependencies occur in practice, we analyze 43 preprocessor-based families and SPLs with a total of over 30 million lines of code, including *Linux*, *Freebsd*, *gcc*, *vim*, and *Best Lap* [Ribeiro et al. 2012]. Even just looking only at methods (*intraprocedural* analysis), between 1 and 24 percent of all methods contain dependencies; but, if we consider only methods with `#ifdefs`, typically *more than a half* also contain dependencies. **Impact:** these numbers serve as a lower bound, since *interprocedural* dependencies were not measured, but they show that the problem is so common in practice that building dedicated tool support can be beneficial for companies that use `#ifdefs`.

3. Emergent Feature Modularization

To help developers avoid feature dependencies problems, we propose a technique called emergent interfaces that establishes, on demand, interfaces to feature code [Ribeiro et al. 2010]. When developers are interested in dependencies from a specific code block, they ask the tool that implements the technique to compute interfaces. Then, interfaces emerge, giving support to maintain one feature without breaking others. To illustrate how emergent interfaces work, we revisit the *Best Lap* scenario, where the developer is supposed to change how the total score is computed. The first step when using our approach consists of selecting the maintenance points. She selects the `totalScore` assignment (see the dashed rectangle in figure to the right) and then our tool capture dependencies between the feature she is maintaining and the others. Finally, the interface emerges as shown in the right-hand side of the figure. The emerged interface states that the maintenance task may impact the behavior of products containing *ARENA*. So, the

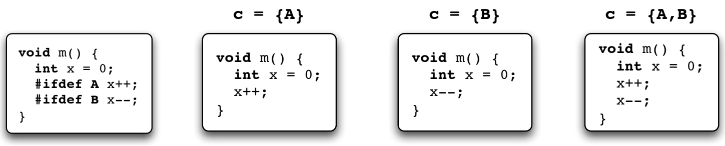


```
public void computeLevel() {
  totalScore = ...
  ...
  #ifdef ARENA ⊕
}
```

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

core functionality provides `totalScore` current's value whereas *ARENA* requires it. The developer is now aware of the dependency. When investigating it, she is likely to discover that she also needs to modify *ARENA* code to avoid introducing an error.

Feature-Sensitive Dataflow Analysis. To compute the interface, we can use conventional dataflow analysis, but generating all possible configurations and analyzing them individually increases costs: an SPL with c number of features will give rise to 2^c number of possible products (minus those invalidated by feature constraints). For the tiny SPL illustrated in Figure 1(a) that has two features, A and B, with the feature model $\psi_{\text{FM}} = A \vee B$, we have to build and analyze *three* products (see Figure 1(b)).



```
void m() {
  int x = 0;
  #ifdef A x++;
  #ifdef B x--;
}
```

c = {A}

```
void m() {
  int x = 0;
  x++;
}
```

c = {B}

```
void m() {
  int x = 0;
  x--;
}
```

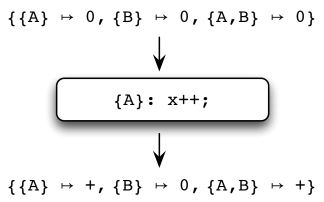
c = {A,B}

```
void m() {
  int x = 0;
  x++;
  x--;
}
```

(a) Toy SPL. (b) Each possible product of our toy SPL.

Figure 1. Generating and analyzing each product separately: high costs.

To minimize costs, we design and implement feature-sensitive dataflow analysis to analyze families and SPLs while staying within the framework of dataflow analysis [Brabrand et al. 2012, Brabrand et al. 2013, Bodden et al. 2013]. Here, there is no need to build all configurations and we need only one fixed-point computation. To explain our proposal, we use the sign analysis, used to analyze the sign of variables. We first lift the CFG to contain feature information. For example, the `x++` node includes $\{A\}$, since it is encompassed by `#ifdef (A)`. Then, we lift the lattice, $\mathcal{L}$, such that it has one element per valid configuration: $[\![\psi_{\text{FM}}]\!] \rightarrow \mathcal{L}$. An example element of this lattice is: $\{\{A\} \mapsto +, \{B\} \mapsto -, \{A,B\} \mapsto \texttt{0/+}\}$ which corresponds to: for $\{A\}$, x is positive "+"; for $\{B\}$, x is negative "−"; and for $\{A,B\}$, it is zero-or-positive "0/+". Finally, we lift the transfer functions to work on elements of the lifted lattice in a point-wise manner. We apply the functions only on the configurations for which the statement is executed. Consider the statement "`#ifdef (A) x++;`". The effect of the lifted function on the lattice element $\{\{A\} \mapsto \texttt{0}, \{B\} \mapsto \texttt{0}, \{A,B\} \mapsto \texttt{0}\}$ is depicted to the right. We apply the function to each of the configurations for which the formula A is satisfied. Since $\{A\} \subseteq \{A\}$ and $\{A\} \subseteq \{A,B\}$, the function is applied to the lattice values of $\{A\}$ and $\{A,B\}$ with resulting value: $f_{\text{x++}}(\texttt{0}) = +$. The same does not happen for configuration $\{B\}$, since it does not satisfy the formula $\{A\} \not\subseteq \{B\}$, so its value is left unchanged with value $\texttt{0}$.



Emergo. We implemented the emergent concept in a Eclipse tool named Emergo (available at `http://www.cin.ufpe.br/~emergo`). Emergo computes interfaces between methods or within a single one, by using *interprocedural* or *intraprocedural* reaching definitions feature-sensitive dataflow analysis. So we can consider only valid feature combinations, preventing developers from reasoning about feature constraints and even from assuming invalid dependencies in case of mutually exclusive features.

4. Evaluation

Previously we suggest that our interfaces can make feature maintenance tasks faster and less error prone. We evaluate these hypotheses in a controlled experiment in two rounds.

Goal, Questions, and Metrics. We focus on maintenance of preprocessor-based SPLs with and without our interfaces, evaluating them from developers' perspective and observing effort and errors they commit. We investigate the following questions: *Do emergent interfaces reduce effort during maintenance tasks involving feature code dependencies in preprocessor-based systems? Do emergent interfaces reduce the number of errors during maintenance tasks involving feature code dependencies in preprocessor-based systems?* To answer them, we measure the time to accomplish a maintenance task and how many incorrect solutions the developer committed during the task (number of errors, NE).

Participants. In a first pilot study, we tested the experimental design with 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*). Half of the participants had professional experience.

Experimental Material and Tasks. We use two preprocessor-based SPLs as experimental material: *Best Lap* and *MobileMedia*. To cover different use cases, we ask participants to perform two kinds of maintenance tasks: to implement a new requirement (requiring *interprocedural* analysis) and to fix unused variables (requiring *intraprocedural* analysis).

Design. To evaluate our research questions, we use a standard *Latin Square design*. So, each participant performs both kinds of tasks on each SPL and also uses and not uses emergent interfaces at some part of the experiment, but no participant will perform the same task twice (with both treatments), which avoids corresponding carry-over effects, such as learning. The design blocks two factors: participant and tasks. For this design, we perform an analysis of variance (ANOVA), following the convention *p-value* < 0.05.

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*). By preparing the Eclipses, we prevent participants from using Emergo when they are not supposed to. All Eclipses have an additional plug-in with two buttons: a *Play/Pause* button for participants to start/stop the chronometer; and a *Finish* button to submit a solution. Before the experiment execution, we perform warmup tasks.

Results. We plot the times for both new-requirement tasks in Figure 4 (left-hand). Here we use beanplot batches, where each batch shows individual observations as small horizontal lines and the density trace forms the batch shape. In Round 1 (see the legend), 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 (3.1 times faster), despite the different student levels. According to an ANOVA test, we obtain statistically significant evidence that emergent interfaces reduce effort in both new-requirement tasks. For the unused-variable task, the use of emergent interfaces adds little: the difference between the treatments is smaller (see Figure 4, right-hand). In fact, we obtain statistically significant evidence only in Round 2 (*1.68 times* faster; *1.5 times* in Round 1). Regarding NE, in Round 1, only one participant committed more errors when using emergent interfaces, and all of them committed errors without. Round 2 roughly confirms the results.

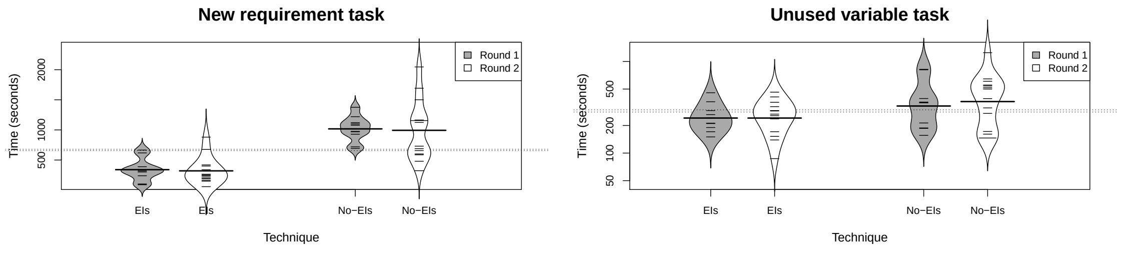


Figure 2. Time results for both tasks: new-requirement and unused-variable

5. Concluding Remarks

We introduce emergent interfaces (and feature-sensitive dataflow analysis) that raise awareness of feature dependencies, improving feature modularity and complementing previous work [Kästner et al. 2008, Baniassad and Murphy 1998]. Our results shows that our interfaces decrease developers effort when faced with *interprocedural* dependencies while also reducing errors introductions during our tasks. All material is available at `http://www.cin.ufpe.br/~mmr3/phd-thesis/`

References

Baniassad, E. L. A. and Murphy, G. C. (1998). Conceptual module querying for software reengineering. In *ICSE*, pages 64–73.

Bodden, E., Tolêdo, T., Ribeiro, M., Brabrand, C., Borba, P., and Mezini, M. (2013). Analyzing software product lines in minutes instead of years. In *PLDI*. To appear.

Brabrand, C., Ribeiro, M., Tolêdo, T., and Borba, P. (2012). Intraprocedural dataflow analysis for software product lines. In *AOSD*, pages 13–24.

Brabrand, C., Ribeiro, M., Tolêdo, T., Winther, J., and Borba, P. (2013). Intraprocedural dataflow analysis for software product lines. *Transactions on AOSD*, 10:73–108.

Kästner, C. et al. (2008). Granularity in Software Product Lines. In *ICSE*, pages 311–320.

Liebig, J., Apel, S., Lengauer, C., Kästner, C., and Schulze, M. (2010). An analysis of the variability in forty preprocessor-based software product lines. In *ICSE*, pages 105–114.

Ribeiro, M. et al. (2011a). On the impact of feature dependencies when maintaining preprocessor-based software product lines. In *GPCE*, pages 23–32.

Ribeiro, M. et al. (2012). On the impact of feature dependencies when maintaining preprocessor-based software product lines. *ACM SIGPLAN Notices*, 47:23–32.

Ribeiro, M., Pacheco, H., Teixeira, L., and Borba, P. (2010). Emergent Feature Modularization. In *Onward!*, pages 11–18.

Ribeiro, M., Toledo, T., Borba, P., and Brabrand, C. (2011b). A tool for improving maintainabiliy of preprocessor-based product lines. In *Tools Session (CBSoft 2011)*.

Yin, Z., Yuan, D., Zhou, Y., Pasupathy, S., and Bairavasundaram, L. (2011). How do fixes become bugs? In *ESEC/FSE*, pages 26–36.