On the Impact of Feature Dependencies when Maintaining Preprocessor-based Software Product Lines

Márcio Ribeiro
Informatics Center
Federal University of Pernambuco
Recife, Brazil
mmr3@cin.ufpe.br

Felipe Queiroz
Informatics Center
Federal University of Pernambuco
Recife, Brazil
fbq@cin.ufpe.br

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

Társis Tolêdo
Informatics Center
Federal University of Pernambuco
Recife, Brazil
twt@cin.ufpe.br

Claus Brabrand
IT University of Copenhagen (ITU)
Copenhagen, Denmark
brabrand@itu.dk

Sérgio Soares
Informatics Center
Federal University of Pernambuco
Recife, Brazil
scbs@cin.ufpe.br

ABSTRACT

During Software Product Line (SPL) maintenance tasks, Virtual Separation of Concerns (VSoC) allows the programmer to focus on one feature and hide the others. However, since features depend on each other through variables and control-flow, feature modularization is compromised since the maintenance of one feature may break another. In this context, *emergent interfaces* can capture dependencies between the feature we are maintaining and the others, making developers aware of dependencies. To better understand the impact of feature dependencies during SPL maintenance, we have investigated the following two questions: how often methods with preprocessor directives contain feature dependencies? How feature dependencies impact maintenance effort when using VSoC and emergent interfaces? Answering the former is important for assessing how often we may face feature dependency problems. Answering the latter is important to better understand to what extent emergent interfaces complement VSoC during maintenance tasks. To answer them, we analyze 43 SPLs of different domains, size, and languages. The data we collect from them complement previous work on preprocessor usage.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*complexity* ; D.3.3 [**Programming Languages**]: Language Constructs and Features—*patterns*

General Terms

Measurement, Design, Experimentation

GPCE '11 Portland, USA

1. INTRODUCTION

A Software Product Line (SPL) is a family of software systems developed from reusable assets. These systems share a common set of features that satisfy the needs of a particular market segment [3]. By reusing assets, it is possible to construct products through features defined according to customers' requirements [16]. In this context, features are the semantic units by which we can differentiate programs in a SPL [19].

To implement features, developers often use preprocessors [7, 11, 2] and associate conditional compilation directives like `#ifdef` and `#endif` to encompass feature code. Despite their widespread use, preprocessors have several drawbacks, including no support for separation of concerns [18]. To overcome this, researchers have proposed Virtual Separation of Concerns (VSoC) [7] as a way of allowing developers to hide feature code not relevant to the current task, reducing some of the preprocessor drawbacks. The main idea is to provide developers with a way of focusing on one feature implementation without being distracted by others [6]. However, VSoC is not enough to provide feature modularization, which aims at achieving independent feature comprehensibility, changeability, and development [15].

In particular, these modularity problems arise because of shared elements among features such as variables and methods. In general this leads to subtle dependencies like when a feature assigns a value to a variable which is subsequently used by another feature. These feature dependencies might cause behavioral problems during SPL maintenance since the programmer may not be aware of them, as illustrated by two scenarios we cover in this paper: (i) maintenance of one feature only works for some products; and (ii) maintenance of one feature makes another not work.

To minimize these problems, we proposed the idea of *emergent interfaces* [17]. The idea is to capture dependencies between the feature a programmer is maintaining and the others. These interfaces emerge and give information about other features we might impact with our current maintenance task. Developers then become aware of the dependencies and, consequently, might avoid the maintainability problems described in the aforementioned scenarios. No-

tice that developers still have the VSoC benefits. Emergent interfaces complement VSoC in that in addition to hiding feature code, they provide dependency information.

Given the problem caused by feature dependencies and two approaches that provide benefits on feature modularity, we focus on the following research questions:

- **Question 1:** how often methods with preprocessor directives contain feature dependencies?
- **Question 2:** how feature dependencies impact maintenance effort when using VSoC and emergent interfaces?

Answering **Question 1** is important to assess to what extent dependencies is a problem in practice. In other words, how important this problem is. Answering **Question 2** is important to better understand to what extent emergent interfaces may complement VSoC during maintenance tasks.

Inspired by recent work [13, 14], we answer **Question 1** by analyzing 43 software product lines taken from different domains, size, and languages (C and Java). In particular, we built a tool —based on [13]— to compute data with respect to preprocessor usage and feature dependencies.

To answer **Question 2**, we use the same 43 product lines to investigate and compare maintenance effort when using VSoC and our emergent interface approach. For example, when the programmer changes the value of a variable, he needs to analyze whether or not the new value impacts other features. Thus, he should check each feature and determine possible dependencies. To perform this evaluation, we randomly select methods and variables from those SPLs. From one particular variable, we estimate the developer effort required to search for dependencies of the variable being maintained.

In Section 2, we present motivating examples to illustrate behavioral problems caused by feature dependencies. Then, in Section 3, we briefly introduce emergent interfaces. After that, we discuss the study settings in Section 4 and present the main contributions of this paper:

- data on preprocessor usage that reveals to what extent feature dependencies occur in practice (complementing previous work [13, 14]); and
- a comparison of VSoC and emergent interfaces in terms of maintenance effort.

2. MOTIVATING EXAMPLES

Virtual Separation of Concerns (VSoC) reduces some of the preprocessor drawbacks by allowing us to hide feature code not relevant to the current maintenance task [7]. Using this approach, developers can maintain a feature without being distracted by other features [6]. However, we show here that VSoC is not enough to provide feature modularization, which aims at achieving independent feature comprehensibility, changeability, and development [15].

To illustrate the maintenance problems we mentioned in the introduction, we now discuss two scenarios likely to occur when using VSoC. Although we focus on VSoC, these scenarios can happen with simple preprocessor directives[1] like `#ifdef`.

[1]In fact, such preprocessors problems are reported in bug tracking systems.

Please note that the maintenance tasks we focus on here cause behavioral problems to the product line.

2.1 Scenario 1: Maintenance of one feature only works for some products

The first example comes from the *best lap*[2] product line. *Best lap* is a casual race game where the player tries to achieve the best time in one lap and qualify for the pole position. It is highly variant due to portability constraints: it needs to run on numerous platforms. In fact, the game is deployed on 65 devices [1].

In this game, there is a method responsible for computing the game score, as illustrated in Figure 1. The method contains a small rectangle at the end, representing a hidden feature that the developer is not concerned with and thus not seeing. Notice that there are no `#ifdef` statements. Instead, the VSoC approach relies on tools that use background colors to represent features, which helps on not polluting the code with preprocessors directives [7].

The hidden feature—in our example named *arena*—is an optional feature responsible for publishing the scores obtained by the player on the network. This way, players around the world are able to compare their results. The method also contains a variable responsible for storing the player's total score (`totalScore`).

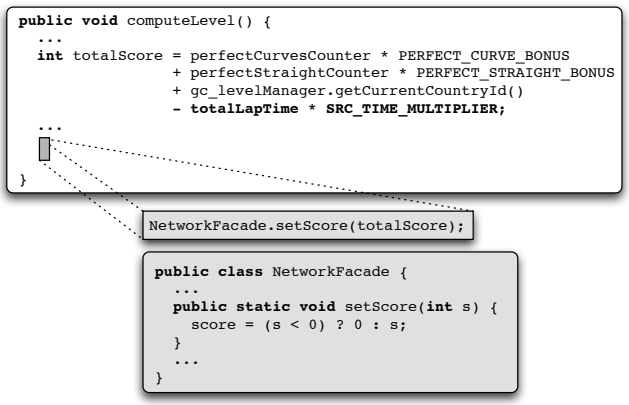


Figure 1: Maintenance only works for some products.

Now, suppose the developer were to implement the following new requirement in the (mandatory) *score* feature: let the game score be not only positive, but also negative. Also, suppose that the developer is using VSoC, so that there are hidden features throughout the code, including *arena*. The developer might well judge that they are not important for the current task. To accomplish the task, he localizes the *maintenance point* (the `totalScore` assignment) and changes its value (see the bold line in Figure 1). Building a product with the *arena* feature enabled and running it may make the developer incorrectly assume that everything is correct, since the negative total score correctly appears after the race. However, when publishing the score on the network, he notices that the negative score is in fact stored as zero (see the expanded *arena* code). Consequently, the

[2]Best lap is a commercial product developed by Meantime Mobile Creations.

maintenance was only correctly accomplished for products without *arena*.

Because there are hidden features, the developer might be unaware of another feature he is not maintaining uses `totalScore` and thus also needs to be changed accordingly to correctly complete the maintenance task. In fact, the impact on other features leads to two kinds of problems. The first one is **late error detection** [6], since we can only detect errors when we eventually happen to build and execute a product with the problematic feature combination (here, any product with *arena*). Second, developers face **difficult navigation** throughout the code. Searching for uses of `totalScore` might increase developer effort. Depending on the number of hidden features, the developer needs to consider many locations to make sure the modification did not impact other features. However, it is possible that some—or even all—features might not need to be considered if they did not use the variables that were modified. Besides, some features are mutually exclusive in that, for instance, the presence of feature *A* prohibits the presence of feature *B*. In our particular case, if we are maintaining feature *A*, then there is no need for the developer to also consider feature *B*. Nevertheless, because this information might not be explicit in code, the developer is susceptible to consider code unnecessarily, increasing maintenance effort.

2.2 Scenario 2: Maintenance of one feature makes another not work

Our second scenario presents an example based on the *TaRGeT*[3] product line. By using *TaRGeT*, we can automatically generate test cases from use cases. So, we have a form in which users can edit use cases. Here, developers reported a bug at the editing use case screen: the system shows unconditionally an error message due to wrongly fulfillment of use case information (see the left side of Figure 2). In this context, a developer responsible for fixing the problem needs to implement the following new requirement: the system should point out which field of the use case screen the user need to fill in again due to validation errors. The idea is to paint the field (in red, for instance) so as to alert the user so that he can correct it.

To fix the bug, an `if` statement is enough. To implement the new requirement, he changed `String` to `String[]`, as illustrated at the right side of Figure 2. This way, he can use the array to store both the error message and the problematic field.

In the same method he is changing, however, there is an optional feature responsible for generating PDF files from the use case, in case no errors were found. From the GUI perspective, this feature consists of a small button at the top of the edit use case screen. The developer is unaware of this feature, so he did not realize that the maintenance introduced a problem in it. Since `error` is now an array, it will never be equal to the empty string, which means that the *PDF* button will never be enabled (see the *PDF* feature code expanded in Figure 2). This now means that PDF documents will no longer be generated. Again, we have the **late error detection** problem. Besides, the **difficult navigation** problem occurs since the method contains three features. Navigating throughout them in search of dependencies may be time consuming. Further, developers are likely to analyze unnecessary features. For example, `error` is not used in the *black* feature.

[3]We do not use *TaRGeT* in our evaluation because very few features use preprocessors. The majority of the features are implemented with components and aspects.

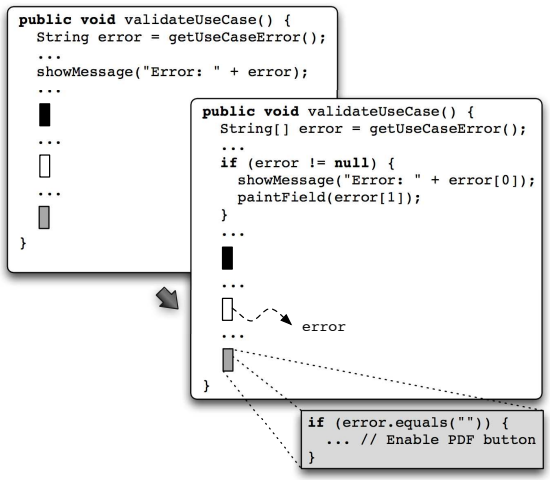


Figure 2: PDF feature does not work anymore.

3. EMERGENT INTERFACES

The problems discussed so far occur when features *share* elements such as variables and methods. In this paper, whenever we have such sharing, we say that there is a *feature dependency* between the involved features. For instance, a mandatory feature might declare a variable subsequently used by an optional feature (see `totalScore` and `error` in Figures 1 and 2, respectively). We thus have a mandatory/optional feature dependency. We can also have feature dependencies like optional/optional and optional/mandatory.

Previously, we presented an approach named Emergent Interfaces [17] intended to help developers avoid the problems related to feature dependencies. The idea consists of determining, on demand, and according to a given maintenance task, interfaces for feature implementations. Such interfaces are neither predefined nor have a rigid structure. Instead, they **emerge** to provide information to the developer on feature dependencies, so he can avoid introducing problems to other features. Our idea complements VSoC in the sense that we still have the hiding benefits (which is important for comprehensibility), but at the same time we show the dependencies between features.

To do so, emergent interfaces capture dependencies between the feature we are maintaining and the remaining ones. In other words, when maintaining a feature, interfaces emerge to give information about other features we might impact with our maintenance. To consider features, emergent interfaces rely on feature code already annotated. We can, for example, use Colored IDE (CIDE) [7], a tool that enables feature annotations by using colors and implements the VSoC approach. To capture dependencies, emergent interfaces rely on feature-sensitive data-flow analysis [17]. In particular, we keep data-flow information for each possible product configuration. This means that our analyses take feature combinations into consideration.

To better illustrate how emergent interfaces work, consider **Scenario 1** of Section 2.1, where the developer is

supposed to change the `totalScore` value. The first step when using our emergent approach consists of selecting the maintenance point. The developer is responsible for such a selection (see the dashed rectangle in Figure 3) which in this case is the `totalScore` assignment. Then, we perform code analysis based on data-flow analysis to capture the dependencies between the feature we are maintaining and the other ones. Finally, the interface emerges.

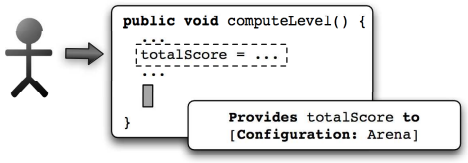


Figure 3: Emergent interface for Scenario 1.

The interface in Figure 3 states that maintenance may impact products containing the *arena* feature. In other words, we *provide* the actual `totalScore` value to the *arena* feature. The developer is now aware of the dependency. Reading the interface is important for **Scenario 1**, since the emerged information alerts the developer that he should also analyze the hidden *arena* feature. When investigating, he is likely to discover that he also needs to modify *arena*, and thus avoid the **late error detection** problem.

Note that the code might have many other hidden features with their own intricate dependencies, making code navigation difficult. In this context, consider **Scenario 2** presented in Section 2.2. In this scenario there are three features. So, we have the **difficult navigation** problem: searching for dependencies is time consuming. Emergent interfaces assist with this problem since they indicate precisely the product configurations the developer needs to analyze. Thus, our interfaces focus on the configurations we indeed might impact, avoiding developers from the task of analyzing unnecessary features, which is important to minimize the **difficult navigation** problem. As Figure 4 depicts, the interface focuses on the *white* and *gray* (*PDF*) features, since they use `error`. Now, the developer is aware of the `error` variable usage in both optional features. Again, the developer would probably discover he also needs to modify the *gray* (*PDF*) feature, thereby minimizing the **late error detection** problem.

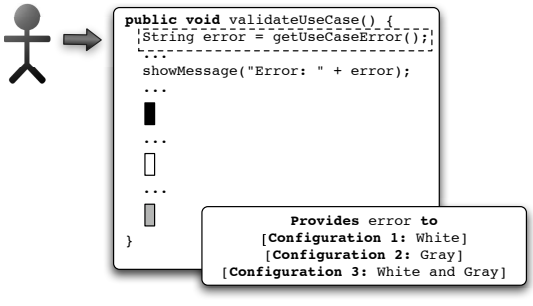


Figure 4: Emergent interface for Scenario 2.

4. STUDY SETTINGS

After showing emergent interfaces and how they can deal with feature dependencies, we now present the details on how we performed our study to answer the two research questions we focus on this paper.

The study is based on 43 software product lines from different domains, sizes, and languages (C and Java). They range from simple product lines to complex ones such as linux. The majority is written in C and all of them contain several features implemented using conditional compilation directives.

We present the selected product lines in Table 1 (at the end of the paper). To compute feature dependencies, we built a tool based on a recent work [13]. We use this tool to compute metrics such as the *number of methods with preprocessor directives (MDi)* and the *number of methods with feature dependencies (MDe)*. Using these metrics, we are able to assess how often feature dependencies occur in the product lines investigated in this paper.

Given the feature dependencies we have in all of these product lines, we evaluate *how* they impact on maintenance effort when using VSoC and emergent interfaces. Our aim consists of understanding to what extent the latter may help complement the former.

Figure 5 shows how we perform our evaluation. The code presented in this figure is based on the *xterm* product line and we are using VSoC to hide features. In this particular case, we have four hidden code fragments[4], which consist of three features (*black*, *gray*, and *white*). The developer is supposed to maintain the `screen` variable (changing `TScreen`, changing the parameter `xw` etc). Notice that there is a fragment outside the `screen` scope. Thus, when maintaining such a variable, the developer does not need to analyze that fragment.

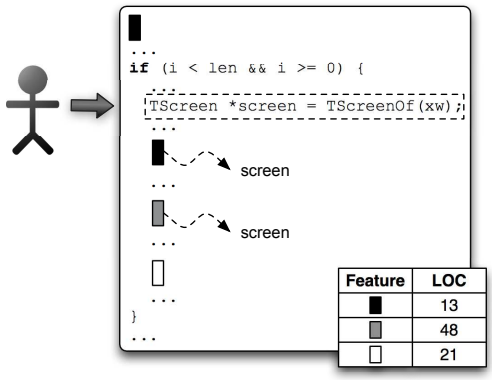


Figure 5: Maintaining the screen variable.

In this context, when using VSoC, the developer does not know anything about the hidden features. Hence, he might need to analyze each hidden fragment to be sure that the maintenance he performed does not impact them. For this particular example, he would analyze three fragments and three features that together correspond to 82 (13 + 48 + 21) source lines of code. On the other hand, emergent interfaces do not hide everything. They still use VSoC but at the same time provide information about dependencies among features, which might be valuable to decrease maintenance effort. For this example, the interface would point out that only two features (*black* and *gray*) use the `screen` variable.

[4]We denote by *fragment*, any preprocessor directive such as `#ifdef`, `#else`, `#elif`, and so forth.

This information is important since the developer would then analyze only two fragments (instead of three) and two features (instead of three). This means that 21 source lines of code (*white* feature) can be discarded from this analysis. This way, our study covers the following three metrics: *source lines of code (SLoC)*, *number of fragments (NoFa)*, and *number of features (NoFe)*. We detail the results of our toy example in Table 2.

Approach	SLoC	NoFa	NoFe
VSoC	82	3	3
Emergent Interfaces	61	2	2

Table 2: VSoC *versus* Emergent Interfaces.

In this paper, we estimate maintenance effort by means of the number of source lines of code, fragments, and features we should analyze during a maintenance task. Therefore, the higher these metrics, the greater the maintenance effort. So, we use *SLoC*, *NoFa*, and *NoFe* to compare maintenance effort when using VSoC and emergent interfaces. Notice that the same effort can be observed regardless of the approach we choose. This happens when emergent interfaces point out feature dependencies in all fragments we indeed have to analyze. Emergent interfaces either reduce the maintenance effort or it remains the same as using VSoC. They decrease the effort when at least one fragment does not have dependencies with the feature we are maintaining.

To perform the effort study we propose, we randomly select methods with feature dependencies and then compute the aforementioned metrics for each approach. We select the methods from the 43 product lines presented in Table 1.

To have a representative study, we now need to tackle the problem of *which* methods we should select to perform the evaluation. On the one hand, we believe that if we select only methods with many fragments, we are favoring emergent interfaces, since the probability of finding at least one fragment with no feature dependency increases. On the other hand, if we select only methods with few fragments (one for instance), we cannot show differences between both approaches since the effort would often be the same. In this way, we need to select the methods carefully. To guarantee the selection of methods with both characteristics, we divide them in two groups:

- **Group 1:** methods with 1 or 2 fragments; and
- **Group 2:** methods with more than 2 fragments.

We chose 2 as our threshold (to divide our groups) because the differences between both approaches appear from this value. In methods with feature dependencies, both approaches always have the same effort when we have only one fragment (same *SLoC*, $NoFa = 1$, and $NoFe = 1$).

Now that we have the groups defined, we randomly select methods accordingly. Firstly, we decided to pick three methods per product line. Since methods of **Group 1** are more common, we would have two methods of **Group 1** and only one of **Group 2**. However, depending on the product line, the quantity of methods of both groups varies significantly. For example, when considering the *libxml2*, we have 953 methods in **Group 1** and 125 methods in **Group 2**. So, we rather select the methods proportionally according to each product line (instead of three methods for all product lines). In this way, for *libxml2*, we select eight method of **Group 1** and one of **Group 2**.

So, the basic idea consists of selecting methods with feature dependencies to fit both groups proportionally according to each product line. Because a method may have more than one variable with dependency, we also randomly select one variable per method. Then, we start our effort evaluation from that variable taking its scope into consideration.

Last but not least, we also consider the Monte Carlo approach with three replications. The idea consists of repeating the whole evaluation three times so that we can take the average of three independent observations.

We summarize how we perform our evaluation as an algorithm (see Algorithm 1).

Algorithm 1 General algorithm of our effort estimation.

```
while we do not reach 3 replications do
  for each product line do
    - Randomly select methods with feature dependencies
    proportionally to fit the groups;
    for each method do
      - Randomly select a variable;
      - From this variable, compute the effort (SLoC,
      NoFa, and NoFe) of both approaches.
    end for
  end for
end while
```

5. RESULTS AND DISCUSSION

After discussing the study settings, in this section we answer the two research questions (Sections 5.1 and 5.2) based on the results obtained from our empirical study. Last but not least, we present in Section 5.3 the threats to validity.

5.1 Question 1

The first question we address in this paper is: **how often methods with preprocessor directives contain feature dependencies?**

To answer this question, we use the *number of methods with preprocessor directives (MDi)* and the *number of methods with feature dependencies (MDe)*. According to the results presented in Table 1, these metrics vary significantly across the product lines. Some product lines have few directives in their methods. For instance, only 2% of *irssi* methods have directives. On the other hand, this number is much bigger in other ones, like *python* (27.59%) and *mobile-rss* (27.05%). Following the convention "average ± standard deviation", our data reveal that $11.26\% \pm 7.13\%$ of the methods use preprocessors.

Notice that the *MDe* metric is low in many product lines. However, we compute this metric with respect to all methods. Rather, if we take only methods with directives into consideration, we conclude that, when maintaining features—in other words, when maintaining code with preprocessor directives—the probability of finding dependencies increases a lot. Taking the *gnumeric* product line as an example, only 4.91% of its methods have directives and 2.24% have feature dependencies. Therefore, almost half of methods with directives (45.56%) have feature dependencies (see column *MDe/MDi* in Table 1, which stands for *MDe* divided by *MDi*). Our data reveals that $65.92\% \pm 18.54\%$ of the methods with directives have dependencies. Therefore, feature dependencies are indeed common in the product lines we analyze.

5.2 Question 2

The second question is the following: **how feature dependencies impact on maintenance effort when using VSoC and emergent interfaces?**

To answer this question, we performed an evaluation with three replications. For each replication, we randomly select 122 methods from all product lines. As mentioned, we select all methods proportionally according to each particular product line to fit the two groups. Table 3 illustrates the number of product lines with their respective methods proportions according to each group. For example, in 13 product lines we select two methods of **Group 1** and one of **Group 2**. Only one product line (*sendmail*) has more methods of **Group 2**. This is consistent with our previous claim that methods of **Group 1** are more common.

Number of SPLs	Group 1	Group 2
23	1	1
13	2	1
3 (gimp, gnumeric, lampiro)	3	1
2 (parrot, linux)	4	1
1 (libxml2)	8	1
1 (sendmail)	1	5

Table 3: Number of SPLs with their respective methods proportions.

As mentioned, to estimate maintenance effort, we consider three metrics: *SLoC*, *NoFa*, and *NoFe*. We illustrate the results for each replication and each metric in Figure 6. Each bar summarizes one particular metric for all 122 methods. The idea is to **summarize** the effort of both approaches and then compare them. As can be seen, emergent interfaces reduced the effort in all replications and metrics. Taking the average of the three replications, when using emergent interfaces, developers would analyze 35% less fragments; 25% less features; and 35% less source lines of code.

We already expected that the effort reduction for features would be smaller when compared to the fragments reduction. Obviously, when developers, based on the interfaces information, discard fragments from their analyses to achieve a particular maintenance task, they also discard lines of code. However, this is not true for features, since we might have two fragments of the same feature, which means that discarding one fragment does not necessarily mean discarding the whole feature from the analysis.

When considering the number of methods, developers have less effort in 33% of methods for replication 1, in 34% for replication 2 and in 39% of methods for replication 3. However, this result is not interesting when analyzed in isolation. But when we cross these numbers with the number of product lines we achieve maintenance effort gains; we can see that these methods are scattered throughout the majority of the product lines we analyze. This indicates that emergent interfaces might indeed reduce maintenance effort in different situations such as product line domains, code sizes, languages, and so forth. Table 4 illustrates, for each replication, the number of product lines where emergent interfaces reduce the effort in at least one method.

Table 4 illustrates the total of methods in which emergent interfaces reduce the effort. Table 5 distributes these methods into the respective groups they belong to. As can be seen, the majority of the methods where emergent interfaces reduce effort are concentrated in **Group 2** (the one where methods have more than 2 fragments).

Rep.	Methods(Less effort)	SPLs(Less effort)
1	40 (33%)	34 (79%)
2	41 (34%)	36 (84%)
3	47 (39%)	36 (84%)

Table 4: Total of Methods and SPLs where emergent interfaces reduced effort.

Methods(Less effort)	Group 1	Group 2
40 (33%)	7	33
41 (34%)	7	34
47 (39%)	14	33

Table 5: Distribution of methods into their groups.

Previously, we mentioned we could favor emergent interfaces in case of selecting only methods with many fragments (in our case, only methods of **Group 2**). We believe this is true because when the number of fragments increases, the probability of finding at least one fragment with no feature dependency increases as well. In this case, the maintenance effort is smaller when compared to VSoC. The results presented in Table 5 suggest that our claim might be true.

Nevertheless, in order to further support this claim, we analyzed the methods of **Group 2** more deeply. Such a group has 47 methods for all replications and they have more than 2 fragments. According to Table 5, emergent interfaces reduce effort in 33, 34, and 33 methods for each replication. So, we achieve maintenance effort reduction in 70%, 72%, and 70% of the methods of this group. By analyzing our data, we can see that, in general, when the number of fragments increases, the percentage of methods in which we achieve maintenance effort reduction also increases (see Figure 7). For example, if we take only methods with more than 4 fragments into consideration, we have effort reduction in 83%, 82% and 84% of those methods.

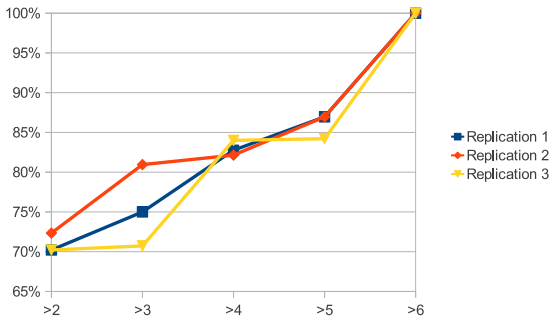


Figure 7: Estimating maintenance effort reduction when increasing the number of fragments.

Emergent interfaces achieve maintenance effort reduction in 35.25% ± 3.6% of the randomly selected methods. The reduction happens in 82%±2.7% of the SPLs we studied. Thus, our results suggest that the interfaces can reduce maintenance effort in SPLs with different characteristics.

Now, we answer the **Question 2** for each approach.

How feature dependencies impact on maintenance effort when using VSoC? Because VSoC does not provide

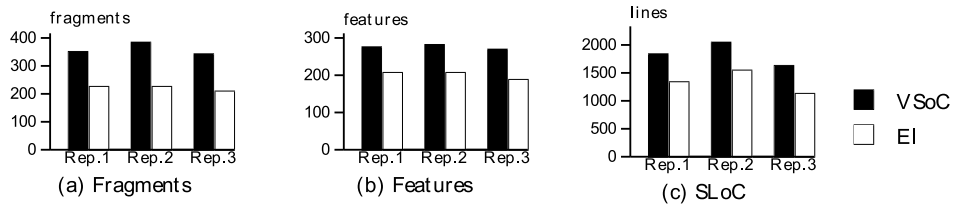


Figure 6: Fragments, features and *SLoC* that developers should analyze in the selected methods when using VSoC and emergent interfaces.

any information about the existence or absence of feature dependencies, developers need to check this in the existing fragments and features. If we have many of them, the effort increases. However, notice that in 64.75% of the methods we analyze, the effort estimation is the same when compared to emergent interfaces. So, the negative impact on maintenance effort when using VSoC is not so common.

How feature dependencies impact on maintenance effort when using emergent interfaces? Based on our study, we can conclude that the more significative gains achieved by emergent interfaces can be observed specially in methods with many fragments. However, only 38% of the methods belongs to **Group 2**, so methods with many fragments occur occasionally. Nevertheless, although we neither evaluated nor focused on methods without dependencies, emergent interfaces also provide benefits in such cases. Figure 8 illustrates a method from *berkeley db*. The developer is supposed to change the value of the `pBt` variable and no feature uses it. Notice that when there is no dependency, the emergent interface is empty, so there is no need to check any fragment or feature. In contrast, VSoC does not provide this information, which may lead developers to analyze unnecessary code (features *black*, *white*, and *gray*).

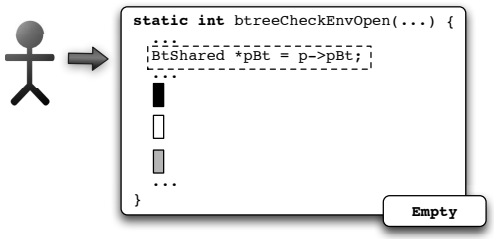


Figure 8: Variable with no dependency.

5.3 Threats to validity

Metrics and effort estimation. The metrics we use in this paper are not sufficient to measure how much the maintenance effort reduces. Instead, they can estimate it. However, they are able to show differences between emergent interfaces and VSoC. Although not sufficient, the metrics are still useful to understand the benefits provided by emergent interfaces. Actually, we are aware of metrics that better measure effort (e.g., time). However, our effort estimation seems plausible since the time would be proportional to the number of artifacts (fragments, features, SLoC) that the developer needs to analyze.

Unavailable feature models. We do not have access to the feature model of all SPLs, so the results of our three metrics (*SLoC*, *NoFa*, and *NoFe*) can change due to feature model constraints we are not aware of. Nevertheless, we believe that this fact changes our effort results only slightly, because the majority of methods we use in our evaluation belongs to **Group 1** (methods with 1 or 2 fragments). Since the number of fragments is small in **Group 1**, it is difficult to find constraints between two features within a method.

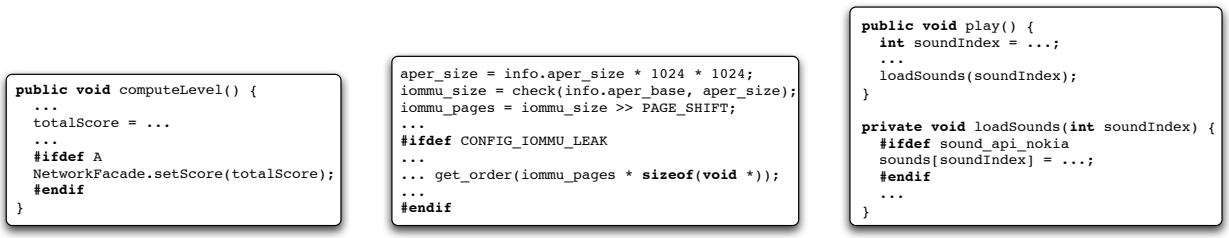
Highlighting tools. When using preprocessors without the VSoC support, highlighting tools are helpful to identify variable usage. Hence, it is possible to find dependencies as well. However, besides losing the VSoC benefits (all features would be shown), highlighting tools are purely syntactical so it does not take flow and feature information into consideration. For example, we might select a variable in feature *A* and the tool highlights variable usage in feature *B*. Since they can be mutually exclusive due to a feature model constraint, the tool points out a false positive, which means that the dependency does not exist.

Dependencies. Our tool computes only *simple dependencies*, as showed in Figure 9(a). However, there are more dependencies neglected by our tool, such as *chain of assignments* (Figure 9(b)) and *interprocedural* (Figure 9(c)). In the first, we have a chain because if we change the *aper_size* value, its new value contributes to define the *iommu_size* value which, in turn, defines the value of *iommu_pages*. And we use this variable in another feature. Moreover, our tool does not consider interprocedural dependencies, as illustrated in Figure 9(c). Note we pass a variable as a method parameter and we use it in another feature in the target method. Since both kinds of dependencies are not present in our statistics, we believe that the real number of dependencies we present to answer **Question 1** is even higher.

6. RELATED WORK

Analyses on preprocessor-based SPLs. There is research on assessing the way developers use preprocessors in SPLs. Recently, researchers [13] created and computed many metrics to analyze the feature code scattering and tangling when using conditional compilation directives. To do so, they analyzed 40 software product lines implemented in C. They formulated research questions and answered them with the aid of a tool. We complement this work by taking feature dependencies into consideration. Also, we provide data in different product lines (the ones written in Java).

Researchers [14] examined the use of preprocessor-based code in systems written in C. Directives like `#ifdefs` are indeed powerful, so that programmers can make all kinds of annotations using them. Hence, developers can introduce subtle errors like annotating a closing bracket but not the opening one. This is an "undisciplined" annotation. Disciplined annotations hold properties useful for preprocessor-



(a) Simple dependency. (b) Chain of Assignments. (c) Interprocedural.

Figure 9: Dependencies from Best Lap, Kernel, and Juggling, respectively.

aware parsing tools so we can represent annotations as nodes in the AST. They found that the majority of the preprocessor usage are disciplined. Specifically, they found that the case studies have 84.4% of their annotations disciplined. We also analyzed several systems, but we focus on dependencies among features implemented with preprocessors.

Another study concerning preprocessor usage in C systems [4] points out that, despite their evident shortcomings, the controlled use of preprocessors can improve portability, performance, or even readability. They found that most systems analyzed make heavy use of preprocessor directives. Like our work, they compute the occurrence of conditional compilation directives as well. We did not find a lot of preprocessor usage. However, we focus only on methods. In contrast, they focus on the entire code (not only methods) and analyze many other kinds of preprocessors (like macros).

We complement these studies providing more data with respect to preprocessors usage. Besides, we estimate maintenance effort when using VSoC and emergent interfaces.

Safe composition. The scenarios we focus on this paper show behavioral problems that can arise when maintaining features in preprocessor-based SPLs. Among other possible scenarios, maintenance in a feature can also break compilation of another. Existing works detect such type errors; the safe composition problem. Safe composition relates to safe generation and verification of properties for SPL assets: i.e., providing guarantees that the derivation process generates products with properties that are obeyed [9, 8].

Safe composition is proposed for the Color Featherweight Java (CFJ) calculus [5]. This calculus establishes type rules to ensure that CFJ code only generates well-typed programs. TypeChef [9] is another type checker that aims at identifying errors in SPLs implemented with the C preprocessor. By using TypeChef, we do not need to generate all variants of the SPL. It relies on the concept of partial preprocessing where macros and file inclusions are processed while the directives that control the actual variability are not. The remaining code is then parsed. The generated AST contains information about the `#ifdefs`, in which reference analysis can be performed to then solve whether all variants are well typed or not.

Emergent interfaces can help in the sense of preventing type errors, since the interface would show the dependencies between the feature we are maintaining and the remaining ones. Nonetheless, safe composition approaches are complementary, since if the developer ignores the feature dependency showed by the interfaces and introduces a type error, these approaches catch them after the maintenance task.

Data-flow analysis for maintenance. Recent work [12] observed developers facing problems of understanding code during maintenance tasks. They found that a significant amount of a developer's work involves answering "reachability questions". This question is a search across the code for statements that match the search criteria. They observed that developers often inserted defects because they did not answer the reachability question successfully. Bringing to our context, we could search for dependencies. If we cannot answer where they are or which features they belong to, we can introduce errors in the SPL. Notice that this is similar to our scenarios and to the **late error detection** and **difficult navigation** problems.

During testing activities, there are features whose presence or absence do not influence some of the test outcomes, which makes many feature combinations unnecessary for a particular test, reducing the effort when testing SPL. This idea of selecting only relevant features for a given test case was proposed in a recent work [10]. The work uses data-flow analysis to recover a list of features we reach from a given test. Since the analysis yields only reachable features, we discard the other ones. Then, we use the reachable features as well as the feature model to discover the combinations we should test, reducing the number of combinations to test. In some sense, the data-flow analysis considers features (the reachable ones). But it is not completely feature-sensitive, since feature model information is not used during the data-flow analysis. In contrast, the data-flow analyses of our emergent approach are feature sensitive. They take feature and feature model information into consideration during the analyses. We detail these ideas elsewhere [17].

7. CONCLUDING REMARKS

In this paper, we presented an analysis on the impact of feature dependencies during maintenance of preprocessor-based SPLs. Firstly, we presented two scenarios that can introduce behavioral errors in product lines due to such dependencies. Then, we focused on two research questions. To answer them, we built a tool to collect data from 43 product lines of different domains, sizes, and languages. The data correspond to preprocessor usage and to what extent feature dependencies occur in practice. They reveal that $65.92\% \pm 18.54\%$ of the methods with directives have dependencies. So, feature dependencies are reasonably common in the product lines studied.

Besides, we performed an empirical study to assess the impact that feature dependencies may cause on maintenance effort when using two approaches: VSoC and emergent in-

terfaces. We estimated effort by using three metrics that essentially counts the number of artifacts that the developer needs to analyze during a maintenance task. We observed that emergent interfaces achieved effort reduction in 35.25% ± 3.6% of the methods we studied. Also, we found that the more significative reductions can be observed specially on the presence of methods with many fragments and features. However, it is important to note that these methods occur occasionally. So, in the majority of the analyzed methods (64.75%), the effort estimation is the same for both approaches. This way, the negative impact on maintenance effort when using VSoC is not so common.

Our data complement previous work on preprocessor usage. In addition, we present an evaluation that is helpful to understand to what extent emergent interfaces complement VSoC in the maintenance effort context.

8. ACKNOWLEDGMENTS

We would like to thank CNPq, a Brazilian research funding agency, and 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. Also, we thank SPG[5] members for feedback and fruitful discussions about this paper.

9. REFERENCES

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

[2] V. Alves, P. M. Jr., L. Cole, P. Borba, and G. Ramalho. Extracting and Evolving Mobile Games Product Lines. In *Proceedings of the 9th International Software Product Line Conference (SPLC'05)*, volume 3714 of *LNCS*, pages 70–81. Springer-Verlag, September 2005.

[3] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[4] M. D. Ernst, G. J. Badros, and D. Notkin. An empirical analysis of c preprocessor use. *IEEE Transactions on Software Engineering*, 28:1146–1170, December 2002.

[5] C. Kästner and S. Apel. Type-checking software product lines - a formal approach. In *Proceedings of the 23rd International Conference on Automated Software Engineering (ASE'08)*, pages 258–267. IEEE Computer Society, September 2008.

[6] C. Kästner and S. Apel. Virtual separation of concerns - a second chance for preprocessors. *Journal of Object Technology*, 8(6):59–78, 2009.

[7] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, pages 311–320, New York, NY, USA, 2008. ACM.

[8] C. Kästner, S. Apel, T. Thüm, and G. Saake. Type checking annotation-based product lines. *ACM Transactions on Software Engineering and Methodology (TOSEM'11)*, 2011.

[9] A. Kenner, C. Kästner, S. Haase, and T. Leich. Typechef: toward type checking #ifdef variability in c. In *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development (FOSD'10)*, pages 25–32, New York, NY, USA, 2010. ACM.

[10] C. H. Kim, D. Batory, and S. Khurshid. Reducing combinatorics in testing product lines. In *Proceeding of the 10th International Conference on Aspect Oriented Software Development (AOSD'11)*, New York, NY, USA, 2011. ACM. To appear.

[11] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi. A Case Study in Refactoring a Legacy Component for Reuse in a Product Line. In *Proceedings of the 21st International Conference on Software Maintenance (ICSM'05)*, pages 369–378, Washington, DC, USA, 2005. IEEE Computer Society.

[12] T. D. LaToza and B. A. Myers. Developers ask reachability questions. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE '10)*, pages 185–194, New York, NY, USA, 2010. ACM.

[13] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10)*, pages 105–114, New York, NY, USA, 2010. ACM.

[14] J. Liebig, C. Kästner, and S. Apel. Analyzing the discipline of preprocessor annotations in 30 million lines of c code. In *Proceeding of the 10th International Conference on Aspect Oriented Software Development (AOSD'11)*, pages 191–202, New York, NY, USA, March 2011. ACM.

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

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

[17] M. Ribeiro, H. Pacheco, L. Teixeira, and P. Borba. Emergent Feature Modularization. In *Onward! 2010, affiliated with ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH'10)*, pages 11–18, New York, NY, USA, 2010. ACM.

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

[19] S. Trujillo, D. Batory, and O. Diaz. Feature refactoring a multi-representation program into a product line. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE'06)*, pages 191–200, New York, NY, USA, 2006. ACM.

A. Online Appendix

We invite researchers to replicate our study. All results are available at: `http://www.cin.ufpe.br/~mmr3/gpce2011`. *Best lap* and *juggling* product lines are commercial products. Hence, we cannot distribute their source code.

[5] `http://www.cin.ufpe.br/spg`

System	Version	Domain	Language	MDe	MDi	MDe/MDi	NoM
berkeley db	5.1.19	database system	C	7.66%	9.07%	84.46%	10636
cherokee	1.0.8	webserver	C	6.37%	8.91%	71.52%	1773
clamav	0.96.4	antivirus program	C	7%	9.35%	74.92%	3284
dia	0.97.1	diagramming software	C	1.94%	3.04%	63.75%	5262
emacs	23.2	text editor	C	2.45%	5.59%	43.8%	4333
freebsd	8.1.0	operating system	C	6.57%	8.98%	73.2%	130307
gcc	4.5.1	compiler framework	C	4.55%	5.95%	76.4%	50777
ghostscript	9.0	postscript interpreter	C	5.76%	7.25%	79.44%	17648
gimp	2.6.11	graphics editor	C	1.85%	2.87%	64.48%	16992
glibc	2.12.1	programming library	C	5.38%	10.03%	53.67%	7748
gnumeric	1.10.11	spreadsheet application	C	2.24%	4.91%	45.56%	8711
gnuplot	4.4.2	plotting tool	C	10.14%	15.41%	65.83%	1804
httpd (apache)	2.2.17	webserver	C	9.34%	12.19%	76.59%	4379
irssi	0.8.15	IRC client	C	1.44%	2%	71.93%	2843
linux (kernel)	2.6.36	operating system	C	3.68%	4.9%	75.09%	208047
libxml2	2.7.7	XML library	C	22.9%	26.92%	85.07%	5324
lighttpd	1.4.28	webserver	C	11.79%	16.73%	70.5%	831
lynx	2.8.7	web browser	C	15.03%	21.41%	70.18%	2349
minix	3.1.1	operating system	C	2.99%	4.53%	65.96%	3114
mplayer	1.0rc2	media player	C	8.82%	12%	73.51%	11730
openldap	2.4.23	LDAP directory service	C	9.91%	12.82%	77.33%	4026
openvpn	2.1.3	security application	C	14.7%	17.95%	81.91%	1694
parrot	2.9.1	virtual machine	C	1.38%	6.12%	22.52%	1813
php	5.3.3	program interpreter	C	8.89%	11.78%	75.51%	10436
pidgin	2.7.5	instant messenger	C	3.38%	5.26%	64.3%	10965
postgresql	8.4.5	database system	C	4.5%	6.33%	71.14%	13199
privoxy	3.0.16	proxy server	C	17.84%	20.95%	85.15%	482
python	2.7	program interpreter	C	5%	27.59%	18.14%	12590
sendmail	8.14.4	mail transfer agent	C	0.84%	4.52%	18.52%	1195
sqlite	3.7.3	database system	C	9.06%	10.64%	85.19%	3807
subversion	1.6.13	revision control system	C	2.66%	4.03%	65.99%	4894
sylpheed	3.0.3	e-mail client	C	5.15%	7.57%	68%	3634
tcl	8.5.9	program interpreter	C	8.4%	10.65%	78.91%	2761
vim	7.3	text editor	C	5.76%	11.05%	52.14%	6354
xfig	3.2.5b	vector graphics editor	C	2.37%	3.93%	60.24%	2112
xinelib	1.1.19	media library	C	6.91%	9.88%	70.01%	10501
xorgserver	1.7.1	X server	C	7.39%	10.15%	72.76%	11425
xterm	2.6.1	terminal emulator	C	20.46%	24.63%	83.08%	1080
bestlapcc	1.0	mobile game	Java	11.95%	20.7%	57.75%	343
juggling	1.0	mobile game	Java	11.14%	16.71%	66.67%	413
lampiro	10.4.1	mobile instant messenger	Java	0.33%	2.6%	12.5%	1538
mobilemedia	0.9	mobile XXX application	Java	5.8%	7.97%	72.73%	276
mobile-rss	1.11.1	mobile feed application	Java	23.84%	27.05%	88.11%	902

Table 1: MDi: Methods with Directives; MDe: Methods with Dependencies; NoM: Number of Methods.