Improving modular reasoning on preprocessor-based systems

Jean Melo
Informatics Center
Federal University of Pernambuco
Recife, Brazil
Email: jccm@cin.ufpe.br

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

***Abstract*—Preprocessors are often used to implement the variability of a Software Product Line (SPL). Despite their widespread use, they have several drawbacks like code pollution, no separation of concerns, and error-prone. Virtual Separation of Concerns (VSoC) has been used to address some of these preprocessor problems by allowing developers to hide feature code not relevant to the current maintenance task. However, different features eventually share the same variables and methods, so VSoC does not modularize features, since developers do not know anything about hidden features. Thus, the maintenance of one feature might break another. Emergent Interfaces (EI) capture dependencies between a feature maintenance point and parts of other feature implementation, but they do not provide an overall feature interface considering all parts in an integrated way. Thus, we still have the feature modularization problem. To address that, we propose Emergent Feature Interfaces (EFI) that complement EI by treating feature as a module in order to improve modular reasoning on preprocessor-based systems. EFI capture dependencies among entire features, with the potential of improving productivity. Our proposal, implemented in an open source tool called Emergo, is evaluated with preprocessor-based systems. The results of our study suggest the feasibility and usefulness of the proposed approach.**

I. Introduction

A Software Product Line (SPL) is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from reusable assets [1]. The idea of SPL is the systematic and efficient creation of products based on strategic software reuse. By reusing assets, we can build products through features defined in accordance with customers' requirements [2]. In this context, features are the semantic units by which we can distinguish product line variants [3]. Feature models define the legal combinations of features by representing commonalities and variabilities of a product line [4].

In order to implement features, developers often use preprocessors [5], [6], [7]. Conditional compilation directives like `#ifdef` and `#endif` encompass code associated with features. Although preprocessor is widespread used for building SPL, its usage can lead to obfuscated source code reducing comprehensibility and increasing maintenance costs (i.e. code pollution), as a result, becoming error-prone [8], [9]. Besides that, preprocessors do not provide support for separation of concerns. In the literature, `#ifdef` directives are even referred to as "ifdef hell" [10], [11].

For this reason, Virtual Separation of Concerns (VSoC) [5] reduces some of the preprocessor drawbacks by allowing developers to hide feature code not relevant to the current maintenance task. VSoC provides to developer a way of focusing on one feature, which is important for his task at the moment [12]. In other words, VSoC is helpful to visualize a feature individually. However, this approach is not enough to provide feature modularization since a developer does not know anything about hidden features [13]. As a result, the developer might introduce some bugs in the system when he changes a variable or method of a determined feature. Features eventually share variables and methods. Several descriptions of feature interaction phenomena are given in the literature (e.g. in [14]). We refer to *feature dependency* whenever we have such sharing like when a feature assigns a value to a variable which is subsequently used by another feature. Thus, one change in one feature can lead to errors in others. Moreover, these errors can cause behavioral problems in the system [15]. In many cases, bugs are only detected in the field by customers post-release when running a specific product with the broken feature [12].

In order to minimize these problems, researchers have proposed the concept of Emergent Interfaces (EI) [16] to capture the dependencies between a feature part that the programmer is maintaining and the others. This approach is called emergent because the interfaces emerge on demand and give information to developer about other feature pieces that can be impacted. EI still have the VSoC benefits. Yet, they do not provide an overall feature interface considering all parts in an integrated way. In other words, EI have just captured dependencies among parts of a feature (not the feature as a whole) because they only know about the existing dependencies to one determined code encompassed with `#ifdef`. Likely, a feature is scattered across the source code and tangled with code of other features (through preprocessor directives) [17]. This way, each `#ifdef` represents one piece of the feature. Thus, there is no a complete understanding of a given feature. As a consequence, the programmer still might introduce errors in the system since he may not be aware of all dependencies of a given feature that he is maintaining. To sum up, EI only capture dependencies of one feature piece at a time.

To address this problem, we propose Emergent Feature Interfaces (EFI) that complement EI because instead of knowing about feature parts, EFI see a feature as a "component" which has provided and required interfaces in order to improve mod-

ular reasoning on preprocessor-based systems by looking for feature dependencies. Our proposal, implemented in an open source tool called Emergo,[1] is evaluated with preprocessor-based systems. The results of our study bring preliminary evidence about the feasibility and usefulness of the proposed concept.

The rest of the paper is structured as follows. Section II presents some motivating examples of real scenarios. Then, in Section III, we describe our proposal as well as the architectural design. After that, we discuss the case study in Section IV. Section V discusses about related work. Finally, in Section VI, we draw some conclusions and give directions for future work.

II. Motivating Example

Emergent Interfaces reduce Virtual Separation of Concerns problems by capturing data dependencies between a feature maintenance point and parts of other feature implementation of a software product line [16]. Using this approach, the developer can maintain a feature part being aware of the possible impacts in others [15]. Nevertheless, we show that EI are not enough to provide feature modularization, which aims at achieving independent feature comprehensibility, changeability, and development [13].

In this context, we present two maintenance scenarios in order to illustrate the problems mentioned in Section I and addressed in this paper. First, consider a JCalc[2]-based product line of a standard and a scientific calculator written in Java. The code of the product line is available here.[3] We transform the JCalc project in a SPL in order to utilize it only as running example that has mandatory, optional and alternative features, that is why we do not consider this product line in our case study. The JCalc class contains the *main* method responsible for executing the calculator (see Figure 1). As we can see, this method has three features: PT_BR, GUI, and LOOKANDFEEL. Notice that the features are tangled along the *main* method. Also, the GUI feature is scattered across the method twice.

Now, suppose a developer needs to maintain the GUI feature. First of all, he should look at the current feature to understand the role of the feature in its entirety, achieving independent feature comprehensibility. To do so, the developer must get the EI for each code encompassed with `#ifdef GUI`. The interfaces related to the first and second `#ifdef GUI` statements are shown in Figure 2 respectively. The first emerged information alerts the developer that he should also analyze the hidden LOOKANDFEEL feature whereas VSoC hides feature code not relevant to the current maintenance task, in this case, maintenance in the GUI feature. So, using EI the developer is aware of the dependencies that include the hidden features. It is important to stress that the GUI feature is scattered in other classes as well.

As we explain now, part of the information presented by the interface might be relevant to a developer, whereas other part might not be. For instance, the second emergent interface

```
public static void main(String args[]) {
    String title;

    //#ifdef (PT_BR)
    title = "JCalc - Calculadora Padrão e Científica";
    //#else
    title = "JCalc - Standard & Scientific Calculator";
    //#endif

    //#ifdef (GUI)
    JFrame frame = new JFrame(title);
    initResolution(frame);
    //#endif

    //#ifdef (LOOKANDFEEL)
    initLookAndFeel(frame);
    //#endif

    //#ifdef (GUI)
    frame.addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
            System.exit(0);
        }
    });
    JCalcStandardFrame myFrame = new JCalcStandardFrame();
    JPanel myPane = myFrame.getPane();

    frame.getContentPane().add(myPane, ...);
    frame.pack();
    frame.setVisible(true);
    //#endif
}
```

Fig. 1. Maintenance in the GUI feature

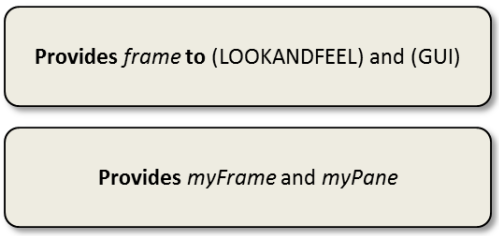

Fig. 2. EI for `#ifdefs GUI`

is irrelevant because these variables reported are not used in other features across the entire system. In addition, the first interface contains both important and not important information for the developer, since he wants to understand whether the code elements of the GUI feature impact other features. So, the second information ('Provides *frame* to (GUI)') from first interface is unnecessary since the developer only does not know anything about the feature LOOKANDFEEL. This means that the developer is aware of that the GUI feature uses the *frame* variable because GUI is being maintained by him. Thus, we have the polluted emergent interfaces problem.

Besides this problem of polluted interfaces, EI have another limitation due to the amount of preprocessor directives per feature. We illustrate that with a second scenario extracted

[1] https://github.com/jccmelo/emergo

[2] http://jcalculator.sourceforge.net/

[3] http://twiki.cin.ufpe.br/twiki/bin/viewfile/SPG/Emergo?rev=1;filename=JCalc.zip

from the Vim editor.[4] Vim is a highly configurable text editor built to enable efficient text editing.

Figure 3 depicts the source code for syntax highlighting of the Vim editor. The *highlight_changed* function translates the 'highlight' option into attributes and sets up the user highlights. According to Figure 3 the *highlight_changed* function has too many preprocessor directives (`#ifdefs`) which represent USER_HIGHLIGHT and FEAT_STL_OPT features.

```
int highlight_changed() {
  ...
  #ifdef USER_HIGHLIGHT
      char_u     userhl[10];
  #ifdef FEAT_STL_OPT
      int        id_SNC = -1;
      int        hlcnt;
  #endif
  #endif
  ...
  #ifdef USER_HIGHLIGHT
      ...
  #ifdef FEAT_STL_OPT
      ...
  #endif
  ...
  #ifdef FEAT_STL_OPT
      highlight_stlnc[i] = 0;
  #endif
  ...
  #ifdef FEAT_STL_OPT
      struct hl_group *hlt = HL_TABLE();
  #endif
  ...
  #ifdef FEAT_STL_OPT
      ...
  #endif
  ...
  #ifdef FEAT_STL_OPT
      highlight_ga.ga_len = hlcnt;
  #endif
  #endif /* USER_HIGHLIGHT */
  return OK;
}
```

Fig. 3. The *highlight_changed* function from the Vim editor

In this context, suppose a developer should study the FEAT_STL_OPT feature in order to implement a user requirement that consists to change this feature. Thus, he has to make use of the emergent interfaces generated for each code block encompassed with `#ifdef FEAT_STL_OPT`. After that, he has to memorize or join all interfaces gotten previously. In this function, he would have to observe six interfaces (vide Figure 3). This becomes worse as the scattering increases. An analysis done in forty preprocessor-based systems claims that a signicant number of feature incur a high scattering degree and the respective implementation scatters possibly across the entire system [18]. This process is time consuming, leading to lower productivity.

Because of the potentially hard work to get all emergent interfaces, the developer might forget some relevant information for maintaining the feature under his responsibility. For instance, in our example, the developer might overlook the information that the FEAT_STL_OPT feature provides *hlt* pointer to another feature, as shown in Figure 3. As a consequence, he might introduce bugs in some SPL variant by leading to late error detection [12], since we can only detect errors when we eventually happen to build and execute a product with the problematic feature combination. This means that the overall maintenance effort increases. All things considered, we have the second problem which is the lack of an overall feature interface for the whole feature, having only access to partial feature interfaces for each block of feature code.

In addition to EI of feature parts, there is an information overload since the interfaces are computed one by one and, then, the developer might join them. This joining process might be expensive and further some of these interfaces might have duplicate information. In face of that, the developer is susceptible to consider code unnecessarily, wasting time. The following section describes how we address these problems.

III. Emergent Feature Interfaces

To solve the aforementioned problems, we propose the idea of Emergent Feature Interfaces (EFI), an evolution of the Emergent Interfaces approach, which consists of inferring contracts among features and capturing the feature dependencies by looking at the feature as a whole. These contracts represent the provided and required interfaces of each feature. With these interfaces per feature, we can detect the feature dependencies (using sensitive-feature data-flow analysis). We use a broader term for contracts than "Design by contract" proposed by Meyer [19] since we infer the contracts by analyzing the code.

We establish contracts between the feature being maintained and the remaining ones through the interfaces. The concept of interfaces allows us to know what a given feature provides and requires from others. Considering the first interface of Figure 2, EI do not inform us about the required interfaces, but the GUI feature requires *title* variable from the PT_BR feature. Therefore, we improve EI by adding required interfaces for computing the emergent feature interfaces.

In addition to establishing contracts, EFI obtain all dependencies between the feature that we are maintaining and the remaining ones. These dependencies occur when a feature shares code elements, such as variables and methods, with others. In general, this occurs when a feature declares a variable which is used by another feature. For example, in our first motivating example (vide Figure 1), the *title* variable is initialized in the PT_BR feature (alternative feature) and, subsequently, used in GUI (mandatory feature). Thus, we can have feature dependencies like alternative/mandatory, mandatory/optional, and so on.

After capturing the feature dependencies, we use feature-sensitive data-flow analysis [20]. In doing so, we keep data-flow information for each possible feature combination.

To clarify the understanding, we present how emergent feature interfaces work. Consider the example with regard to JCalc product line of Section II, where a programmer is supposed to maintain the GUI feature. As our proposal derives

[4] http://www.vim.org/

from EI, the programmer still has to select the maintenance point but with a slight difference since he can also ask about a determined feature. In other words, the developer can select both a code block as a feature declaration (i.e. `#ifdef FeatureName`). The developer is responsible by the selection (see the dashed rectangle in Figure 4) which in this case is the `#ifdef GUI`. Then, we perform code analysis based on data-flow analysis to capture the dependencies between the selected feature and the other ones. Finally, the feature interface emerges.

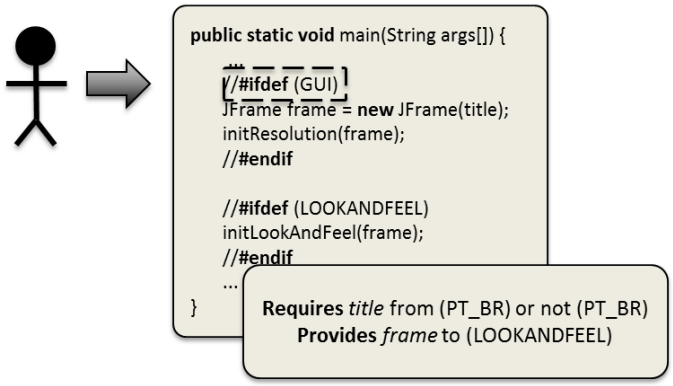

Fig. 4. Emergent feature interface for the GUI feature

The EFI in Figure 4 states that maintenance may impact products containing the LOOKANDFEEL feature. This means we provide the actual *frame* value to the LOOKANDFEEL feature and require the *title* value offered by the PT_BR feature or its alternative one. Reading this information, the developer is now aware of the existing contracts between GUI and LOOKANDFEEL features and also between GUI and PT_BR. The emerged information has been simplified because the developers only need to know the dependencies inter-feature, not intra-feature. That is, EFI only show dependencies between the feature he is maintaining and the remaining ones. We can check this in Figure 1 where the *frame* variable is used in other places but the emergent feature interface (see Figure 4) just exhibit *frame* dependency concerning the feature LOOKANDFEEL. The Figures 2 and 4 show the difference between EI and EFI clearly. Thus, the polluted emergent interfaces problem is addressed by cutting irrelevant information to programmer. This way, he focuses on information that should be analyzed avoiding considering code unnecessarily, wasting time.

It is important to stress that EFI compute both *direct* and *indirect* dependencies among features. In other words, if a variable A is set in FEATURE_1 and used in FEATURE_2 to assign its value in other variable B (B = A). Finally, FEATURE_3 uses B. In this case, EFI alert to programmer that FEATURE_1 has a direct dependency to FEATURE_2 (through A) as well as FEATURE_2 to FEATURE_3 (through B). Besides that, EFI provide the *indirect* dependency between FEATURE_1 and FEATURE_3 since FEATURE_3 depends on FEATURE_1 indirectly (transitivity property).

Note that the code might have many other `#ifdefs` making the interface's construction more complex. According to the results presented by the authors of the concept of EI, the most methods have several preprocessor directives [15]. This means that it is better look at the feature as a whole instead of seeing it part by part. In fact, it is easier to the developers understand the dependencies among features through a macro vision than get all interfaces one by one and, then, join them. For instance, in our second scenario (see Section II) there are several `#ifdefs` and, in special, the FEAT_STL_OPT feature is scattered across the *highlight_changed* function (see Figure 3). Instead the developer having to repeat six times the `#ifdef FEAT_STL_OPT` selection for obtaining all interfaces we provide an integrated way to avoid this information overload. This way, the developer only need to select `#ifdef FEAT_STL_OPT` one time, then the data-flow analysis is performed and, finally, the feature interface emerged (as shown in Figure 5). Note that no dependencies found between FEAT_STL_OPT and the remaining ones. Again, reading this information the developer already knows that the FEAT_STL_OPT feature neither provides any variable or method nor requires to other ones.

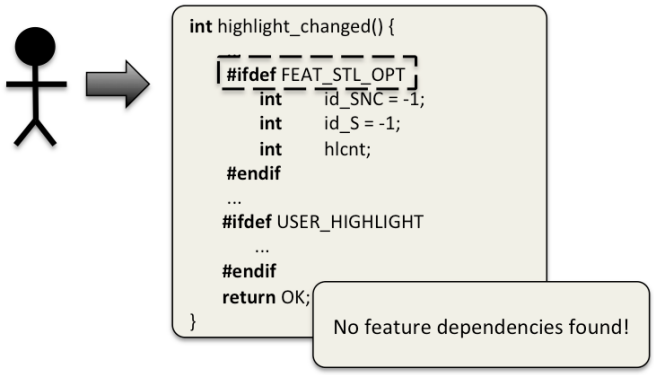

Fig. 5. Emergent feature interface for the FEAT_STL_OPT feature

Therefore, our idea complements EI in the sense that we evolve this approach taking into account the feature as a module. As a result, the developer sees the feature dependencies through a interface more clean and precise that aids him to understand the impact of a possible change in the source code during preprocessor-based system maintenance. In that sense, interfaces enable modular reasoning [17]: We can understand a feature in isolation without looking at other features. Thus, EFI help to solve the feature modularization problem, since the programmer achieves independent feature comprehensibility and, consequently, he can change a feature code aware of its dependencies, avoiding breaking the contracts among features [13]. This improvement is feasible and useful to improve modular reasoning on preprocessor-based systems, with the potential of improving productivity.

We present how EFI work in terms of implementation in the next section.

A. Implementation

To implement our approach, we built a tool called Emergo, an Eclipse plugin. It is available at: https://github.com/jccmelo/emergo.

The Figure 6 depicts both the Emergo's architecture and the data-flow from developer's selection up to the interface to appear. The architecture follows a heterogeneous architectural style based on the layered (GUI, Core, and Analysis) and independent components.

To show the process for getting EFI, we explain step by step the activity diagram-like (as seen in Figure 6).

First of all, the developer selects the maintenance point which indicates what code block or feature he is interested at the moment. Then, the **GenerateEIHandler** component sets up the classpath from the accessible information at the project. Besides that, it gets the compilation unit in order to know whether is a Java or Groovy project, treating each type of project in accordance with its peculiarities.[5] Meanwhile, we associate the maintenance point selection with a feature expression and compute the Abstract Syntax Tree (AST) **from Eclipse infrastructure**. After, we mark each tree node that represents the selected code by the developer. This marking of the AST nodes from text selection is important to bind AST node on Soot's Unit object later.

Incidentally, Soot [21] is a framework for analysis and optimization of Java code. It accepts Java bytecode as input, converts it to one of the intermediate representations, applies analyses and transformations, and converts the results back to bytecode. Since we consider SPLs, we use the Soot framework to execute feature-sensitive data-flow analysis [20] and then capture feature dependencies.

Before we apply data-flow analysis, the **Soot** component gets the classpath provided by the **GenerateEIHandler** and configures it by putting all bytecode in a specific place. Then, **Soot** loads the class that the developer is maintaining. This means, **Soot** gets the class bytecode and converts it into main intermediate representation of the Soot called Jimple, typed 3-address representation of Java bytecode. Jimple is suitable for decompilation and code inspection.

In addition to load the class, we use the bridge design pattern to deal the difference between Java and Groovy independently. This way, we can bind AST nodes on Soot Units which correspond to statements. After this step, we have a mapping between AST nodes and statements and, hence, we are able to get the units in selection.

This mapping is passed to **Instrumentor** that iterates over all units, look up for their feature expressions and add a new Soot Tag to each of them, and also computes all the feature expressions found in the whole body. Units with no feature expression receive an empty Soot Tag. The idea is to tag information onto relevant bits of code in order that we can then use these tags to perform some optimization in the dependency graph at the end.

After the bytecode instrumentation, we build the Control Flow Graph (CFG) and, then, run reaching definitions analysis through the **LiftedReachingDefinitions** component that uses the Soot data-flow framework. The Soot data-flow framework is designed to handle any form of CFG implementing the interface `soot.toolkits.graph.DirectedGraph`. It is important to stress that our reaching definitions analyses are feature-sensitive data-flow analysis [20]. This way, we keep data-flow information for each possible feature combination.

Then, the **DependencyGraphBuilder** component accepts mapping between AST nodes to statements, units in selection, CFG, and all possible feature combinations as input, iterates over the CFG for creating the nodes from units in selection which can represent use or definition. If the node is a definition we get all uses and for each use found we create one directed edge on the dependency graph which represents the EFI. Otherwise, we just get its definition and connect them. Recalling that both paths support transitivity property.

After the dependency graph is populated, we prune it to avoid duplicate edges and having more than one edge between two given nodes.

Finally, the **EmergoGraphView** component shows the dependency graph in a visual way where the developer becomes aware of the feature dependencies, with the potential of improving productivity. Besides this graph view, we also provide a table view. These information alert the developer about what interfaces might be broken if he changes the code in specific places.

B. Limitations and Ongoing work

Our tool currently implements the general algorithm to emerge feature interfaces. The main limitation when computing interfaces happens when we have mutually exclusive features. Although the data-flow analysis is feature-sensitive, the Emergo still searches uses of a determined definition in all feature expressions, whether alternative or not. Improving this computation is an ongoing work.

Also, we are working on interprocedural analysis for capturing dependencies among classes, packages, and components since a feature can be scattered in different places.

IV. Case Study

To assess the effectiveness and feasibility of our approach, we conducted a case study following guidelines from Runeson and Host [22]. Our evaluation addresses these research questions:

- **RQ1:** Is there any difference between Emergent Interfaces and Emergent Feature Interfaces?
- **RQ2:** How do Emergent Feature Interfaces' dependency detection compare to Emergent Interfaces?

Our study includes five preprocessor-based systems in total. All of these software product lines are written in Java and contain their features implemented using conditional compilation directives. These systems contain several features. For instance, the lampiro product line has 11 features and, the mobile media has 14 features. Among these systems, *Best lap* and *Juggling* product lines are commercial products.

We reuse the data produced by other research [15], whose authors proposed the EI concept. Table I shows the product lines, including their characteristics such as the amount of preprocessor directives utilized in the entire product line, among others. We count preprocessor directives like `#ifdef`, `#ifndef`, `#else`, `#if`, and so on. *MDi* stands for *number of methods with preprocessor directives*, for example, *Mobile-rss* has 244 methods with preprocessor directives (27.05%) from 902 existing methods in entire product line. And, *MDe* stands for *number of methods with feature dependencies*. In particular, we use *MDe* in order to select the methods with feature

[5] http://groovy.codehaus.org/Differences+from+Java

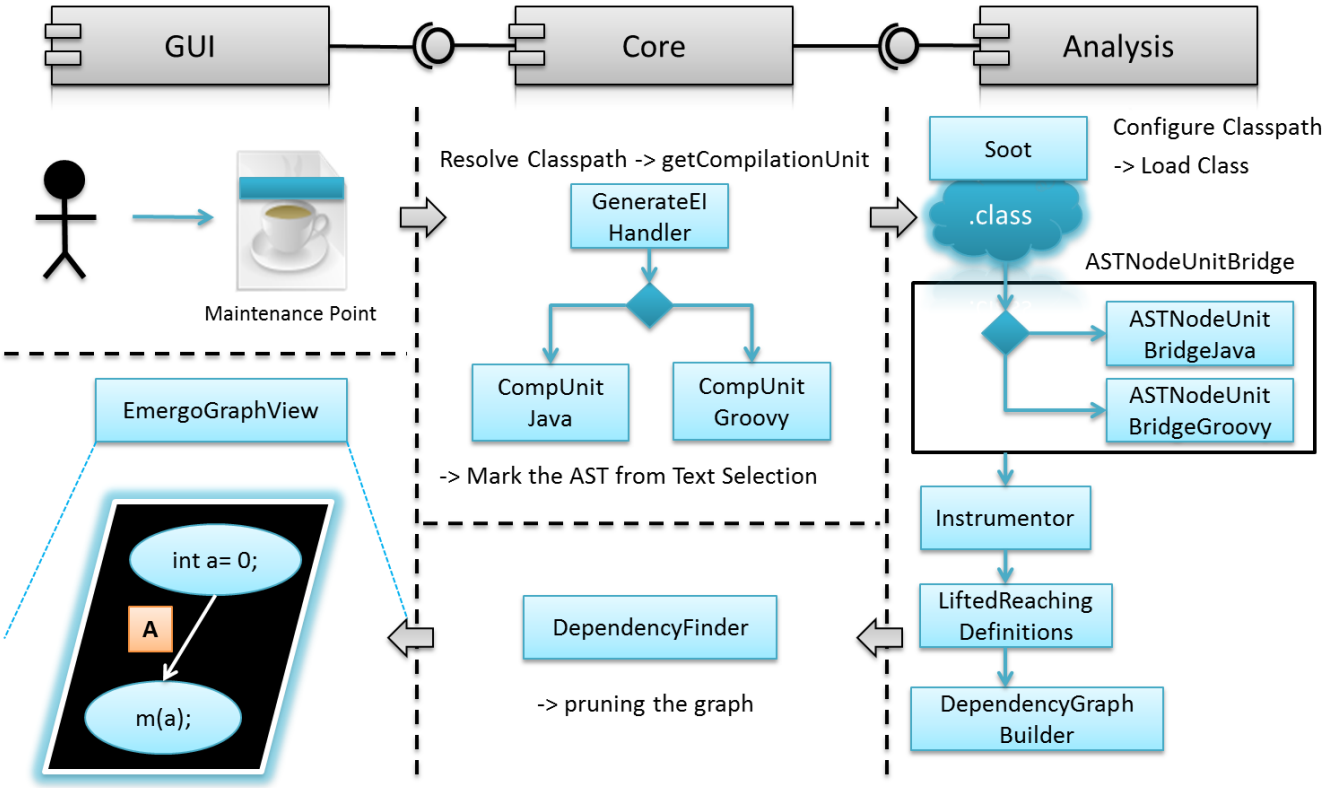

Fig. 6. Emergo's architecture and activity diagram-like

TABLE I. CHARACTERISTICS OF THE EXPERIMENTAL OBJECTS.

System	Version	MDi	MDe	# methods	# cpp directives
Best lap	1.0	20.7%	11.95%	343	291
Juggling	1.0	16.71%	11.14%	413	247
Lampiro	10.4.1	2.6%	0.33%	1538	61
MobileMedia	0.9	7.97%	5.8%	276	82
Mobile-rss	1.11.1	27.05%	23.84%	902	819

dependencies to answer our research questions. According to the presented data in Table I, these metrics vary across the product lines.

Given a maintenance point in some of these product lines, we evaluate what the difference between EI and EFI. Our aim consists of understanding to what extent the latter complement the former.

To answer these research questions, we randomly select a subset of methods with feature dependencies [15] and then compare the results produced by EFI to the results generated by EI. Also, note that the same set of selected methods is used to conduct the comparisons between EFI and EI. From these five experimental objects, we have 446 methods with preprocessor directives. We use a random way for getting ten methods that contains feature dependencies. Firstly, we decide to pick two methods per product line. Also, we randomly select the maintenance points. In doing so, we identify some valid maintenance points dismissing comment, whitespace and method/class declaration since our data-flow analysis is intraprocedural. Then, we compute EI and EFI for each maintenance point chosen. To select these methods and maintenance points, we use RANDOM.ORG[6] that offers true random numbers to anyone on the Internet.

After discussing the study settings, we present the results of our evaluation for each method with feature dependencies as shown in Table II. For each method selected, the table shows EI and EFI produced from the maintenance points. It is important to quote that depending on maintenance point selection the method might have not dependency among features. Although these ten methods contain feature dependencies, there are two cases where no dependencies were found, that is, these variables in selection are not neither used nor defined in another feature.

As can be seen, EI return 'No dependencies found!' in all the cases that the maintenance point is not an assignment. In other words, this suggests that our claim regarding to miss required interfaces on EI is true. On the contrary, EFI take into consideration both provided and required interfaces. Yet, whenever some method has not feature dependency the two approaches give the same interface. For instance, there is no difference between EI and EFI for the *Resource.save* method (vide Table II). Although this method has no dependencies, EFI look for the definition of the *playerID* variable across the method in order to alert the developer the backward contract (required interface) between the feature he is maintaining and the other. In this case, EFI returned 'No dependencies found!' because *playerID* variable is defined at the same feature that contains its use. Otherwise, EFI would return 'Requires *playerID* variable from feature X' where X represents the

[6]http://www.random.org/

feature name. This latter kind of case happened, for example, at the constructor of the class *ResourceManager* where EI did not find any dependencies whereas EFI found.

Besides that, EI do not provide support for feature selection as a maintenance point. This is bad since the developer might want to understand a feature as a whole before applying any change the code. For example, consider the Best lap product line' *MainScreen.paintRankingScreen* method, if the developer wanted to know what feature dependencies exist between the *device_screen_128x128* feature and the remaining ones, he should select all statements (one-by-one) within that feature. This is a potential hard work depending on the amount of statements of the feature. In this context, our approach is useful and feasible since EFI provide macro information per feature, improving modular reasoning on preprocessor-based systems (see the first line of the Table II).

Another important aspect is the simplified view that EI do not offer to developers. For instance, the *PhotoViewController.handleCommand* method has the *imgByte* declaration encompasses with `#ifdef sms || capturePhoto`. This variable is used in different places (`sms || capturePhoto` and `copyPhoto`). EI show both use places in their message whereas EFI only alert the developer about dependencies outside the current feature configuration. This way, the developer just needs to worry with the copyPhoto feature since he is aware of the feature that he is maintaining. Thus, we believe that a simplified and global view helps the developer, with the potential of improving productivity.

In summary, we believe that when the number of feature dependencies increases, our approach is better than EI because the probability of finding at least one required interface increases as well. In addition, when the number of feature dependencies increases EI might have too much information whereas EFI present a simplified view to the developer. At last, whenever the developer wants to see the feature dependencies of a specific feature EFI is the best option. Thus, the answer of the first research question is yes in cases where the maintenance point is not a assignment, including a particular occasion when the developer selects a feature such as `#ifdef device_screen_128x128`. The second question has already been responded along the previous paragraphs.

A. Threats to validity

To minimize selection bias, we randomly choose ten methods and the maintenance points. Yet, we get a subset of the product lines presented by Ribeiro *et. al* [15] in order to test our tool. For this, all five product lines selected are written in Java. Another threat is that we do not have access to the feature model of all SPLs, so the results can change due to feature model constraints, but we test both approaches (EI and EFI) of equal manner. In addition, we manually compute EI and EFI, as shown in Table II. This can contain some error, but we are very familiar with these approaches and we still revise each generated interface twice.

We acknowledge the fact that more studies are required to draw more general conclusions. However, the results bring preliminary evidence about the feasibility and usefulness of the proposed approach. Although we cannot respond questions like "Is maintenance easier using our approach than using EI?" and, "How feature dependencies impact on maintenance effort when using EI and EFI?" precisely, we believe that our study is an approximation to answer these questions because we provide more abstract and accurate interfaces than EI.

V. Related Work

Many works investigate incorrect maintenance [23], [24], [25]. Sliwerski *et al.* [25] proposed a way to find evidence of bugs using repositories mining. They found that developers usually perform incorrect changes on Friday. Anvik *et al.* [26] applied machine learning in order to find ideal programmers to correct each defect. On the other hand, Gu *et al.* [23] studied the rate of incorrect maintenance in Apache projects. The proposed work helps in the sense of preventing errors during SPL maintenance, since the interface would show the dependencies between the feature we are maintaining and the remaining ones.

Some researchers [27] studied 30 million code lines (written in C) of systems that use preprocessor directives. They found that directives as `#ifdefs` are important to write the code. But, the developers can easily introduce errors in the program. For example, open a parenthesis without closing it or even write a variable and then use it globally. The examples we focus on this paper show modularity problems that can arise when maintaining features in preprocessor-based systems. We analyze some software systems implemented with preprocessors and, then, we use these acquired knowledge to propose the Emergent Feature Interfaces concept. Also, we implement an Eclipse plug-in, Emergo, for helping the developers, avoiding breaking feature contracts.

Emergent Interfaces [16] allow us to capture the dependencies among code snippets of distinct feature parts. This approach is called emergent because the interfaces emerge on demand and give information to developer about other feature pieces which can be impacted. However, this approach still leave of capture some feature dependencies [15]. Moreover, it has just captured dependencies among parts of a feature (not treating the feature as a whole). Our proposal complements this one by capturing dependencies among entire features by providing an overall feature interface considering all parts in an integrated way. Thus, EFI improve modular reasoning on preprocessor-based systems, with the potential of improving productivity.

Recently some researchers [28] proposed analysis of exception flows in the context of SPL. For instance, a variable feature signaled an exception a different and unrelated variable feature handled it. When exception signalers and handlers are added to an SPL in an unplanned way, one of the possible consequences is the generation of faulty products. Our approach has a different manner for improving the maintainability of SPLs. We detect the feature dependencies by executing feature-sensitive data-flow analysis in order to improve modular reasoning on SPLs when evolving them. We do not consider implicit feature relation that comes about in the exceptional control flow since we just focus on dependencies among annotated features (with preprocessor directives).

Finally, using the feature model, it is known that not all feature combinations produce a correct product. Depending on the problem domain, selecting a feature may require

TABLE II. EVALUATION RESULTS.

System	Method	Maintenance Point	EI	EFI
Best lap	MainScreen.paintRankingScreen	#ifdef device_screen_128x128	Do not provide support for this selection!	Provides *rectBackgroundPosX, rectBackgroundPosY, positionPosX, loginScorePosX, etc* values to root feature.
Best lap	Resources.save	dos.writeUTF(playerID);	No dependencies found!	No dependencies found!
Juggling	TiledLayer.paint	firstColumn = (clipX - this.x)/this.cellWidth;	Provides *firstColumn* value to game_tiledlayer_optimize_backbuffer feature.	Provides *firstColumn* value to game_tiledlayer_optimize_backbuffer feature.
Juggling	Resouces.load	playerLogin = dis.readUTF();	No dependencies found!	Requires *dis* variable from root feature.
Lampiro	ChatScreen.paintEntries	int h = g.getClipHeight();	Provides h value to root feature.	Provides *h* value to root feature and requires *g* variable from root feature.
Lampiro	ResourceManager.ResourceManager	while ((b = is.read()) != -1) {...}	No dependencies found!	Requires *is* variable from GLIDER feature.
MobileMedia	PhotoViewController.handleCommand	byte[] imgByte = this.getCapturedMedia();	Provides *imgByte* value to configurations: [sms || capturePhoto], and [copyPhoto && (sms || capturePhoto)].	Provides *imgByte* value to copyPhoto feature.
MobileMedia	SelectMediaController.handleCommand	List down = Display.getDisplay(...);	Provides *down* value to Photo, MMAPI and Video features.	Provides *down* value to Photo, MMAPI and Video features.
Mobile-rss	UiUtil.commandAction	m_urlRrnItem = null;	No dependencies found!	No dependencies found!
Mobile-rss	RssFormatParser.parseRssDate	logger.finest("date=" + date);	No dependencies found!	Requires *date* variable from root feature.

or prevent the selection of others (e.g. alternative features). Features model is a mechanism for modeling common and variable parts of a SPL. Safe composition is used to ensure that all products of a product line are actually correct [29]. Thaker *et al.* [29] determined composition restrictions for feature modules and they used these restrictions to ensure safe composition. However, the developer does not know about any error in his maintenance before performing commit, since safe composition only catch errors after the maintenance task. Our approach differs from safe composition since we intend to use emergent feature interfaces to prevent errors when maintaining features. Moreover, some elements in our EFI deal with the system behavior (value assignment), rather than only with static type information. Nonetheless, safe composition is complementary because the developer may ignore the feature dependency showed by our approach and, then, introduce a type error. So, safe composition approaches catch it after the maintenance task.

VI. CONCLUSION

This paper presents emergent feature interfaces which might be applied to maintain features in product lines in order to achieve independent feature comprehensibility by looking for entire feature dependencies. We provide an overall feature interface considering all parts in an integrated way.

Features tend to crosscut a software product line's code, and thus providing, or computing, a complete interface for them is difficult, if not impossible. The idea is to provide a global view of a feature by abstracting away irrelevant details. We focus on capturing data dependencies, but our proposal can be extended to compute other kinds of interfaces, including dependencies related to exceptions, control flows, and approximations of preconditions and postconditions. The feature modularization problem can be seen in any system, since features can be annotated (explicit) on code base or not (implicit). This way, our solution is over techniques for implementing features in a system. But, for the time being, our tool only runs on condition that features are implemented using conditional compilation.

We also discusses our progress over EI by adding required interfaces, macro feature and simplified view of the existing dependencies the code, implemented in a tool called Emergo. After a selection, Emergo shows an EFI to the developer, keeping him informed about the contracts between the selected feature and the other ones, with the potential of improving productivity. Although we do not conduct a controlled experiment involving developers in order to claim more precisely whether using EFI is better than EI in terms of maintenance effort, we can claim (through our case study) that EFI have potential benefits to the developers leading to the productivity, since EFI' interfaces present information more global and accurate. The results of our study, on five SPLs, suggest the feasibility and usefulness of the proposed approach where the minimum result is equals to EI.

As future work, we intend to improve our tool with more robust emergent feature interfaces. Also, we are working to put interprocedural analysis on Emergo to capture feature dependencies among classes, packages and components. At last, we should conduct more studies, including a controlled experiment with developers, to draw more general conclusions.

ACKNOWLEDGMENTS

The authors would like to thank CNPq, FACEPE, and The National Institute of Science and Technology for Software Engineering (INES), for partially supporting this work. Also, we thank reviewers and SPG[7] members for feedback and rich discussions about this paper.

REFERENCES

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

[7] http://www.cin.ufpe.br/spg

[2] K. Pohl, G. Böckle, and F. J. v. d. Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

[3] 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*, ser. GPCE '06. New York, NY, USA: ACM, 2006, pp. 191–200. [Online]. Available: http://doi.acm.org/10.1145/1173706.1173736

[4] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Carnegie-Mellon University Software Engineering Institute, Tech. Rep., November 1990.

[5] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in software product lines," in *Proceedings of the 30th international conference on Software engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 311–320. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368131

[6] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi, "A case study in refactoring a legacy component for reuse in a product line," in *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, sept. 2005, pp. 369–378.

[7] E. Alves, P. Matos, L. Cole, P. Borba, and G. Ramalho, "Extracting and Evolving Mobile Games Product Lines," in *Proceedings of SPLC'05, LNCS 3714*. Springer-Verlag, 2005, pp. 70–81.

[8] H. Spencer, "ifdef Considered Harmful, or Portability Experience with C News," in *In Proc. Summer'92 USENIX Conference*, 1992, pp. 185–197.

[9] M. Krone and G. Snelting, "On the inference of configuration structures from source code," in *Proceedings of the 16th international conference on Software engineering*, ser. ICSE '94. Los Alamitos, CA, USA: IEEE Computer Society Press, 1994, pp. 49–57. [Online]. Available: http://dl.acm.org/citation.cfm?id=257734.257742

[10] J. M. Favre, "Understanding-in-the-large," in *Proceedings of the 5th International Workshop on Program Comprehension (WPC '97)*, ser. WPC '97. Washington, DC, USA: IEEE Computer Society, 1997, pp. 29–. [Online]. Available: http://dl.acm.org/citation.cfm?id=523511.837867

[11] D. Lohmann, F. Scheler, R. Tartler, O. Spinczyk, and W. Schröder-Preikschat, "A quantitative analysis of aspects in the ecos kernel," in *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, ser. EuroSys '06. New York, NY, USA: ACM, 2006, pp. 191–204. [Online]. Available: http://doi.acm.org/10.1145/1217935.1217954

[12] C. Kästner and S. Apel, "Virtual Separation of Concerns - A Second Chance for Preprocessors," *Journal of Object Technology*, vol. 8, no. 6, pp. 59–78, 2009.

[13] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, pp. 1053–1058, December 1972. [Online]. Available: http://doi.acm.org/10.1145/361598.361623

[14] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec, "Feature interaction: a critical review and considered forecast," *Comput. Netw.*, vol. 41, no. 1, pp. 115–141, Jan. 2003. [Online]. Available: http://dx.doi.org/10.1016/S1389-1286(02)00352-3

[15] 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 *Proceedings of the 10th ACM international conference on Generative programming and component engineering*, ser. GPCE '11. New York, NY, USA: ACM, 2011, pp. 23–32. [Online]. Available: http://doi.acm.org/10.1145/2047862.2047868

[16] M. Ribeiro, H. Pacheco, L. Teixeira, and P. Borba, "Emergent feature modularization," in *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, ser. SPLASH '10. New York, NY, USA: ACM, 2010, pp. 11–18. [Online]. Available: http://doi.acm.org/10.1145/1869542.1869545

[17] C. Kästner, S. Apel, and K. Ostermann, "The road to feature modularity?" in *Proceedings of the 15th International Software Product Line Conference, Volume 2*, ser. SPLC '11. New York, NY, USA: ACM, 2011, pp. 5:1–5:8. [Online]. Available: http://doi.acm.org/10.1145/2019136.2019142

[18] 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 International Conference on Software Engineering (ICSE)*, New York, NY, 5 2010, pp. 105–114.

[19] B. Meyer, "Applying "design by contract"," *Computer*, vol. 25, no. 10, pp. 40–51, Oct. 1992. [Online]. Available: http://dx.doi.org/10.1109/2.161279

[20] C. Brabrand, M. Ribeiro, T. Tolêdo, and P. Borba, "Intraprocedural dataflow analysis for software product lines," in *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, ser. AOSD '12. New York, NY, USA: ACM, 2012, pp. 13–24. [Online]. Available: http://doi.acm.org/10.1145/2162049.2162052

[21] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," in *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, ser. CASCON '99. IBM Press, 1999, pp. 13–. [Online]. Available: http://dl.acm.org/citation.cfm?id=781995.782008

[22] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Softw. Engg.*, vol. 14, no. 2, pp. 131–164, Apr. 2009. [Online]. Available: http://dx.doi.org/10.1007/s10664-008-9102-8

[23] Z. Gu, E. T. Barr, D. J. Hamilton, and Z. Su, "Has the bug really been fixed?" in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 55–64. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806812

[24] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram, "How do fixes become bugs?" in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 26–36. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025121

[25] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *SIGSOFT Softw. Eng. Notes*, vol. 30, pp. 1–5, May 2005. [Online]. Available: http://doi.acm.org/10.1145/1082983.1083147

[26] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *Proceedings of the 28th international conference on Software engineering*, ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 361–370. [Online]. Available: http://doi.acm.org/10.1145/1134285.1134336

[27] J. Liebig, C. Kästner, and S. Apel, "Analyzing the discipline of preprocessor annotations in 30 million lines of C code," in *Proceedings of the tenth international conference on Aspect-oriented software development*, ser. AOSD '11. New York, NY, USA: ACM, 2011, pp. 191–202. [Online]. Available: http://doi.acm.org/10.1145/1960275.1960299

[28] H. Melo, R. Coelho, and U. Kulesza, "On a feature-oriented characterization of exception flows in software product lines," in *Software Engineering (SBES), 2012 26th Brazilian Symposium on*, 2012, pp. 121–130.

[29] S. Thaker, D. Batory, D. Kitchin, and W. Cook, "Safe composition of product lines," in *Proceedings of the 6th international conference on Generative programming and component engineering*, ser. GPCE '07. New York, NY, USA: ACM, 2007, pp. 95–104. [Online]. Available: http://doi.acm.org/10.1145/1289971.1289989