Emergent Feature Modularization

Márcio Ribeiro Humberto Pacheco Leopoldo Teixeira Paulo Borba

Informatics Center, Federal University of Pernambuco, 50740-540, Recife – PE – Brazil

{mmr3, hsp, lmt, phmb}@cin.ufpe.br

Abstract

Virtual Separation of Concerns reduces the drawbacks of implementing product line variability with preprocessors. Developers can focus on certain features and hide others of no interest. However, these features eventually share elements between them, which might break feature modularization, since modifications in a feature result in problems for another. We present the concept of emergent feature modularization, which aims to establish contracts between features, to prevent developers from breaking other features when performing a maintenance task. These interfaces are product-line-aware, in the sense that it only considers valid feature combinations. We also present a prototype tool that implements the concept.

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

General Terms Design

Keywords Product Lines, Modularity, Preprocessors

1. Introduction

A Software Product Line (SPL) is a family of intensive systems developed from reusable assets. These systems share a common set of features that satisfy the specific needs of a particular market segment [6]. By reusing assets, it is possible to construct products through specific features defined according to customers' requirements [17]. Features are the semantic units by which different programs within a SPL can be differentiated and defined [21]. The set of possible products of a SPL is usually represented through feature models.

Features are often implemented using preprocessors [3, 12]. Conditional compilation directives such as `#ifdef` and `#endif` encompass code associated with features. Despite their widespread use, preprocessors have some drawbacks, including no support for separation of concerns [18]. Virtual Separation of Concerns (VSoC) [12] allow developers to hide feature code not relevant to the current task, being important to reduce some of the preprocessors drawbacks. The idea is to provide developers a way to focus on a feature without the distraction brought by other features [10].

Although this approach is helpful to visualize a feature individually, it does not modularize features to the extent of supporting independent feature maintenance and development [16], since developers know nothing about hidden features. In fact, by visualizing and trying to maintain a feature individually, a developer might introduce errors into the hidden features, since these features eventually share elements — such as variables and methods — with the maintained feature. For instance, the new value of a variable might be correct to the maintained feature, but incorrect to another one that uses this variable. Thus, we have a problem due to the lack of feature modularization: the modification of a feature leads to errors in another one. Moreover, this problem may be worse because this error is only noticed when running the product built with the problematic feature [10].

In this work, we propose the concept of emergent feature modularization, which consists of establishing contracts among feature implementations. We call our approach emergent because the components and interfaces here are neither predefined nor have a rigid structure. Instead, they emerge on demand to give support for specific feature development or maintenance tasks. For example, using the emergent concept, the developer firstly selects feature code to maintain. We associate this selection with a feature, or a combination of features, which we denote as a feature expression. Then, information with respect to the other features and their combinations emerge through an interface.

We also achieve the hiding benefits towards feature comprehensibility. But, while still hiding the feature code, emergent interfaces abstract its details. At the same time, they provide valuable information to maintain the selected code and keep other features and their combinations safe.

This paper makes the following contributions:

- We present the concept of emergent feature modularization to help developers when maintaining SPL implemented with preprocessors-like mechanisms (Section 3);

Onward! 2010, October 17–21, 2010, Reno/Tahoe, Nevada, USA.

- A general algorithm to compute emergent components and their emergent interfaces (Sections 3.1, 3.2, and 3.3);
- A prototype tool based on CIDE [12] (a tool that relies on VSoC) to support the concept. It computes and shows emergent interfaces after developers select the code to maintain. Emergent interfaces contain provided/required information to/from other features (Section 4).

2. Motivating Example

Virtual Separation of Concerns (VSoC) reduces some of preprocessors drawbacks, allowing developers to hide feature code not relevant to the current maintenance task [12]. Thus, developers can, to some extent, maintain a feature without the distraction brought by other features [10]. However, we show here that VSoC is not enough to provide feature modularization, which aims at achieving independent feature comprehensibility, changeability, and development [16].

For example, consider **Scenario 1**, where a developer has to maintain the `Music` feature of the Mobile Media SPL [7]. It implements this functionality using the J2ME standard media API, so that users can play music in formats like *MIDI* and *Mp3*. Basically, the implementation contains a controller (`MMController`), responsible for handling the play and stop events, and a screen (`MMScreen`), responsible for painting the buttons and encapsulating the media API. Our scenario consists of adding a new format to play music: the *Ogg* open format. Since the *Ogg* API supports not only play and stop but also pause, rewind and forward events, we need a new controller and screen for it. Figure 1 shows the changes in part of the `Music` feature code in order to fit this new format.

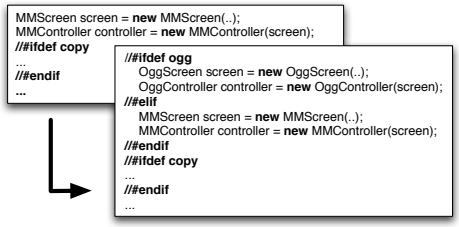

Figure 1. Adding Ogg format to the Mobile Media SPL.

Notice that there is an `#ifdef` directive encompassing the code of the `Copy` optional feature, which allows us to remove that part of the code. Therefore, in the original version of the product line, we have at least two products (possible feature selections): (i) `Music` without `Copy`; and (ii) `Music` with `Copy`. The compiler raises no errors when compiling the (i) variant in the resulting version of the product line. However, there is a compilation error when we take the (ii) variant into consideration. The `Copy` feature uses a method (`controller.setMediaName(..)`) that is only defined in `MMController`, so that it does not exist in our new `OggController` class due to a programmer failure. This shows that we have no proper feature modularization: the modification of a feature (`Music`) breaks another feature (`Copy`). In addition, this situation gets worse because this error is only noticed when the developer eventually builds the product line with the problematic feature combination (using `Copy`).

We do not provide the `Copy` feature code on purpose in order to simulate a developer that is using VSoC, so that he is not concerned about other features (such as `Copy`). Consequently, he is neither maintaining nor visualizing the code surrounded by the `Copy` `#ifdef` directive. To some extent, this support for hiding features is worthwhile to the independent *feature comprehensibility* benefit, since it may help developers to comprehend a feature individually. Despite this advantage of visualizing features individually, VSoC does not provide enough support for understanding and modifying features in separate. For example, because there is no information about the hidden feature (`Copy`) when maintaining `Music`, problems may occur in it. So, the independent *changeability* benefit is not achieved.

In this context, sharing information about two or more features may be a confusing point for two developers, so that achieving the *parallel development* is difficult. For example, consider **Scenario 2**, where a developer is responsible for evolving the `Copy` feature and another one is responsible for the `Music` feature. The first developer might decide to use the `screen` variable for implementing a progress bar for showing the copy progress. Meanwhile, because only one place in the original version of the product line uses the `screen` variable, the second developer might decide to change `MMController(screen)` for `MMController(new MMScreen(..))` and delete the `screen` declaration. Since `screen` is now undeclared, a compilation error will happen in the `Copy` feature. It happens because there is no "*mutual agreement between the creator and accessor*" [22]. Since this contract does not exist, developers of a feature might actually break another one.

Scenario 1 and **Scenario 2** basically show that conditional compilation, even with VSoC support, does not provide adequate modularization support. Bug reports[1] of systems like Linux Kernel and Mozilla present similar problems (undeclared variables, missing methods, ...) involving different features. Besides these kinds of syntactic errors, as discussed in the remaining of the paper, we also consider semantic errors such as when changing the value of a variable that another feature uses. This way, the new value might be right for the maintained feature but wrong for others. Our approach aims to address these cases as well.

3. Emergent Feature Modularization

To solve the problems discussed previously, we propose the concept of emergent feature modularization, which basically consists of establishing, on demand and according to a given development task, interfaces for feature implementations. It is an uncommon way to think about components and

[1] See http://bugzilla.kernel.org/ and https://bugzilla.mozilla.org/

interfaces: they are not predefined by developers, nor have a rigid structure. Instead, we compute them on demand to give support for specific feature development tasks.

For example, in a maintenance task, we consider the feature code to be maintained a component, named *Selection*. The backward/forward paths of the code surrounding it are components too. Paths consider the different feature combinations by the feature model. We name them *dataflows*, since features exchange data among them. Interfaces basically capture data dependencies between these components, and give support to maintaining *Selection* without having to understand the details of code associated to the *dataflows*.

Still using the Mobile Media SPL [7], we now illustrate our approach using a scenario with two optional features, `Sorting` and `Favorites`, and a mandatory one, `Management`, which is the feature to maintain — the *Selection* component. Figure 2 illustrates forward dataflows with arrows. Since `Sorting` and `Favorites` are optional features, there are four different feature expressions, each one associated with a *dataflow* component: **d1:** `Management` $\wedge$ ($\neg$ `Favorites`) $\wedge$ ($\neg$ `Sorting`); **d2:** `Management` $\wedge$ `Favorites` $\wedge$ `Sorting`; **d3:** `Management` $\wedge$ `Favorites` $\wedge$ ($\neg$ `Sorting`); and **d4:** `Management` $\wedge$ ($\neg$ `Favorites`) $\wedge$ `Sorting`.

For each component (*Selection* and *Dataflow*), we compute associated interfaces expressing dependencies between them. For example, Figure 2 shows that the dataflow *d3* (associated with `Favorites`) requires the `media` variable provided by the *Selection*. These interfaces allow us to change *Selection* abstracting details of the surrounding feature code. At the same time, they provide information to the developer, so that he might avoid implementations that cause problems to other features, like removing `media`, for instance.

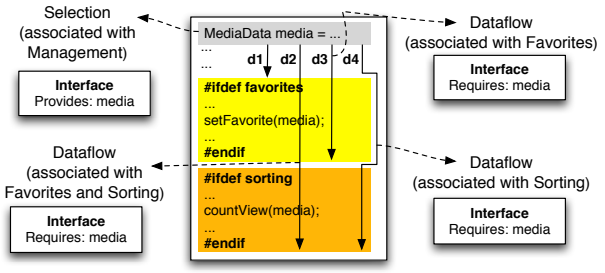

Figure 2. Components and their respective interfaces.

An advantage of using the feature model information is that we can filter which dataflows to take into account. For example, suppose that we associate the *Selection* component with feature *A*, that cannot be present in the same product with feature *B*. This might be due to a constraint in the feature model or by them being alternative features. Thus, we discard dataflows containing both features.

The conceptual model of our approach, depicted in Figure 3, summarizes these ideas. As explained, there are two kinds of components: *Selection*, which corresponds to the code to maintain (selected by the developer); and *Dataflow*, representing backward (from the beginning of the method to the selection) and forward dataflows derived from the *Selection* component. The dataflows are useful to navigate through the code, being important to retrieve data dependencies among features with respect to the *Selection*. For example, through dataflow *d4*, we learn that the *Sorting* feature uses `media`, which is a variable declared in the *Selection*. Notice that we discard dataflows in which all code is only associated with the *Selection* feature expression. In this way, *d1* (Figure 2) is not taken into account. So, as mentioned, we associate each component with a feature expression and an interface, which states that components may provide/require elements such as variables to/from other components. These interfaces emerge from the components, establishing contracts between features.

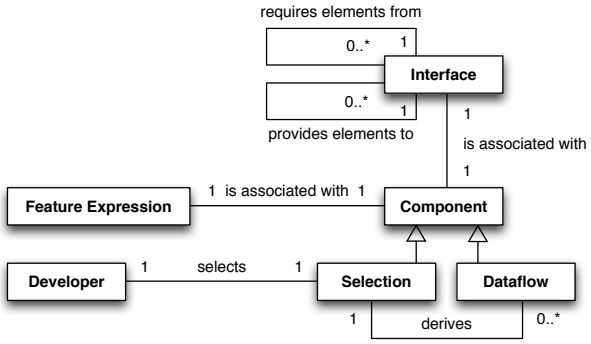

Figure 3. Conceptual Model of our approach.

We now illustrate how our approach might be useful to avoid the problems of the scenarios showed in Section 2.

Scenario 1

In order to add the *Ogg* format, we place both `MMScreen` and `MMController` declarations into an `else` statement. Since the maintenance involves these declarations, developers select them before proceeding with the task, as illustrated in **Step 1** of Figure 4. This is an example of *Selection* component. Additionally, this component is associated with a feature expression, which, in this case, is *Music*. Now, this developer needs information to proceed with the maintenance without damaging other features. For example, which information *Selection* provides (like declared variables and their values) that other features require.

By using the *Selection* component, we derive dataflow components. In Figure 4, arrows from *Selection* towards other parts of the code represent dataflows (*d1* and *d2*). Since `Copy` is an optional feature, we have the following dataflows with their respective feature expressions: **d1**: `Music` $\wedge$ `Copy` and **d2**: `Music` $\wedge$ ($\neg$ `Copy`).

After defining the *Selection* and *Dataflow* components, their interfaces emerge in order to establish the contracts. Dataflow *d1* is associated with the `Copy` feature, which uses the `controller` variable from the `Selection` component

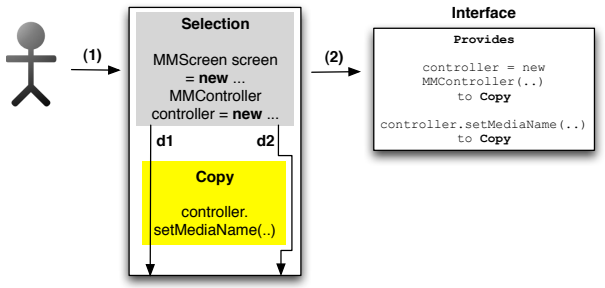

Figure 4. Emergent Interface for Scenario 1.

(see Figure 4). This way, the interface of dataflow *d1* states that `Copy` requires `controller`. Because the developer is maintaining the *Selection*, he must be aware that there is another feature (and, maybe, another developer) depending on `controller`. In this way, the *Selection* interface emerges (**Step 2** of Figure 4), stating that the `Selection` component should provide to the `Copy` feature a variable `controller` assigned to a `MMController` object and this object must have the `setMediaName(..)` method. This interface might be useful to avoid problems like the one reported in **Scenario 1** of Section 2. Now, looking at the interface makes the developer think twice before assigning the same `controller` variable to an `OggController` object.

Scenario 2

Figure 5 illustrates two developers working on two different features. Each developer is responsible for a feature, as explained in **Scenario 2** of Section 2: developer *A* maintains the `Music` feature, whereas developer *B* works on the `Copy` feature. So, developer *B* selects the `Copy` code snippet, as illustrated in **Step 1** of Figure 5. Because there is code before and after the selection, forward and backward dataflows are computed. They are, respectively, *d3* and *d4*. Again, these dataflows are helpful to establish the contracts. We can use them to discover other features that share information contained in the *Selection* component of developer *B*.

Now, suppose that the contracts have already emerged (*Selection* interfaces illustrated in **Steps 2** and **Step 3**) and when maintaining `Copy`, developer *B* decides to use the `screen` variable (from the `Music` feature) to implement a progress bar for the `Copy` feature (**Step 4**). Developer *B* now requires a variable provided by another feature (`Music`). Thus, we update both *Selection* and dataflow interfaces. **Step 5** shows the updated *Selection* interface for developer *B*. This way, when selecting the `Music` feature before a maintenance, the analogous components/interfaces computations occur for developer *A*, and the *Selection* interface shows information to him (**Step 6**). Now, when looking at this interface, developer *A* would think twice before refactoring `new MMController(screen)` to `new MMController(new MMScreen())`.

Therefore, our emerging interfaces should help developers to make some changes in one feature without breaking others, even when they are working on parallel, mitigating the problem illustrated in **Scenario 2**.

Now we present the details on how we implement our emergent approach. Mainly, we defined a general algorithm that consists of three major steps: (i) Compute *Selection* and dataflows components; (ii) Compute their interfaces; and (iii) Match these interfaces.

3.1 Selection/Dataflows Components

The first step consists of computing the *Selection* component and the dataflow components. We associate the *Selection* component with a feature expression and its computation is straightforward: it is the Abstract Syntax Tree (AST) representing the code selected by the developer (see **Step 1** and **Step 2**) of Figure 6. From the *Selection* component, we compute the backward and forward dataflow components. As discussed earlier, we compute dataflows in accordance to the feature expression associated with the *Selection* component and the feature model.

Figure 6 shows a simplified feature model of the Mobile Media product line [7]. It contains the *Management* mandatory feature (filled circle) and the *Copy* and *SMS* optional features (open circles). When maintaining the *Selection* component (associated with *Management*), we compute the following dataflows: **d1:** `Management` $\wedge$ ($\neg$ `Copy`) $\wedge$ ($\neg$ `SMS`); **d2:** `Management` $\wedge$ `Copy` $\wedge$ `SMS`; **d3:** `Management` $\wedge$ `Copy` $\wedge$ ($\neg$ `SMS`); and **d4:** `Management` $\wedge$ ($\neg$ `Copy`) $\wedge$ `SMS`.

Although all these feature expressions are valid according to the feature model, there is a particularity within the code of Figure 6: dataflows *d2* and *d4* are the same because of the two conditional directives: `copy || sms` and `sms`. When considering dataflow *d2*, `Copy` and `SMS` are present, so that both directives evaluate to *true*, which means that *d2* starts from the *Selection* towards the second `#endif` directive. The same happens to dataflow *d4*. Because `SMS` is present, both conditional compilation directives evaluate to *true* as well.

These feature expression combinations are important to alert developers about the impact of their maintenance. By using them, we are able to inform the exact product configurations impacted by a determined maintenance task. For example, if the maintenance assigned a new value to `canv` (Figure 6), we can inform the developer which SPL products this change may affect. In this case, it affects all possible feature combinations. If the number of possible affected products is high, and if the `canv` maintenance point is avoidable, developers may opt for another strategy or algorithm to maintain the desired code without causing potential problems in those products.

3.2 Selection/Dataflows Interfaces

Now that the *Selection* and the dataflow components are already defined, we should compute their interfaces. First, we consider the *Selection* interface, calculated by using not

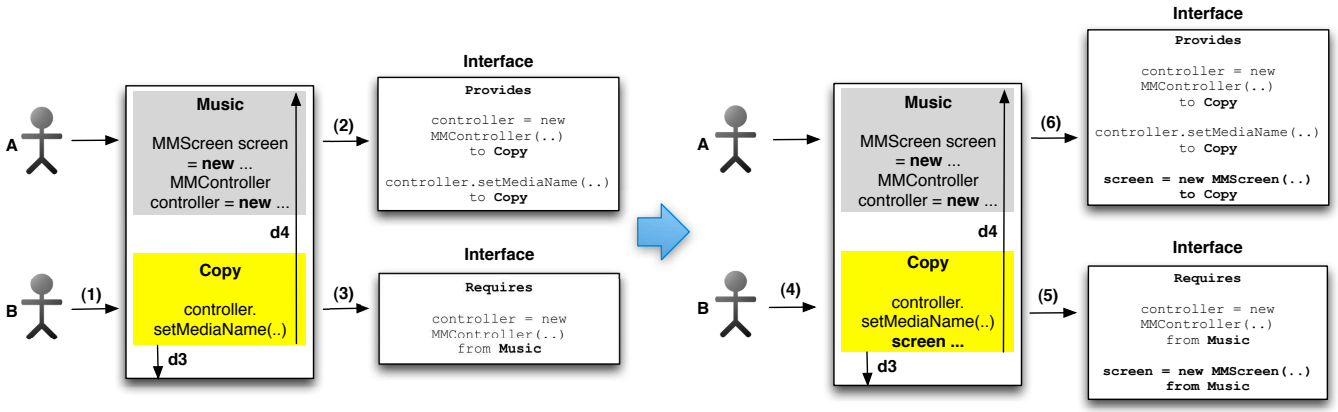

Figure 5. Emergent Interfaces for Scenario 2.

only the *Selection* component, but also the backward and forward dataflow components.

When the developer selects the code associated with the *Management* mandatory feature, we compute the AST of the *Selection* component and navigate throughout it to define the *Selection Elements List (SEL)*. This list plays an important role to define the *Selection* interface. It represents all declarations, assignments and variable uses within the *Selection* component. We illustrate SEL in **Step 3** of Figure 6. For each dataflow component, we navigate through its code searching for elements of SEL. We use some algorithms in this search. Now, we detail two of them:

- **Does any other feature need variable declarations?** Taking `canv` variable from SEL as an example, we search in the forward dataflows components for uses of this variable within features other than *Management*. Because this variable appears in three dataflow components (*d2*, *d3*, and *d4*) within other features (*Copy* and *SMS*), we add it to the *Selection* interface. Thus, developers must be careful when dealing with this variable, since there are other features needing it. This way, the *Selection* interface states the following:
  - **Provides** `canv` to: `Copy` $\wedge$ `SMS`; `Copy` $\wedge$ ($\neg$ `SMS`); and ($\neg$ `Copy`) $\wedge$ `SMS`
- **Does any other feature need a specific assignment?** Now, we consider the assignments present in SEL that reach other features. If there are two assignments to a variable within *Selection*, we only consider the last one. There is only one assignment in SEL: `nextcontroller = this`. This algorithm verifies in the forward dataflows components if this assignment reaches other features. As we can see, it reaches the feature expressions associated with *d2*, *d3*, and *d4*, since each of these dataflows (bold line in Figure 6) uses `nextcontroller` assigned to `this`. Although we reach all feature expressions, the other use of `nextcontroller` (within `#ifdef sms`) is not reached because `nextcontroller` gets reassigned (italic line in Figure 6). We show the updated *Selection* interface in what follows.
  - **Provides** `canv` and `nextcontroller = this` to: `Copy` $\wedge$ `SMS`; `Copy` $\wedge$ ($\neg$ `SMS`); and ($\neg$ `Copy`) $\wedge$ `SMS`

Now, the developer of the *Selection* component knows what he should provide in order to keep the other features safe. In addition, this emergent interface abstracts details of these surrounding features, keeping the developer focused on the maintenance task as well as on the elements that may cause problems to these features.

Notice that there is an element of SEL that is not used in any of the dataflows: `storedImage`. This way, removing such a variable or changing its value does not affect other features, but only the maintained feature. Therefore, this variable is not considered in the *Selection* interface.

We consider the dataflow interfaces as follows. We need to compute the interfaces for the dataflow components of Figure 6. Instead of using the SEL as an input, we use the *Selection* interface in order to avoid useless elements, such as `storedImage`. Thus, two elements must be considered: `canv` and `nextcontroller = this`. Now, we search for these elements within each dataflow component.

- *d2*, *d3*, and *d4* **Require**: `canv` and `nextcontroller = this` from `Management`

3.3 Matching Interfaces

Given that we have the *Selection* and *Dataflow* interfaces, we check if they match. That is, if everything that is required by the interfaces is provided. Using the example from Figure 6, suppose that the developer responsible for the *Management* feature now decides to change the value assigned to `nextcontroller`. Thus, `Selection` does not provide `nextcontroller = this` anymore, which interfaces of dataflows *d2*, *d3*, and *d4* require. The contract is now broken, and this should be reported to the developer.

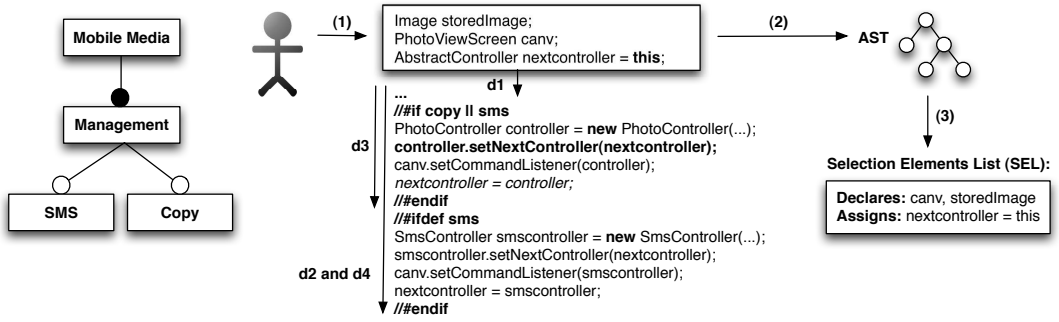

Figure 6. Copy and SMS features of Mobile Media.

4. Supporting Developers

To support developers in charge of maintaining annotative-based SPLs, we present xCIDE (eXtended Colored IDE): a prototype tool that implements the concept of emergent interfaces[2]. We implemented xCIDE as an extension of CIDE [12], a tool that allows developers to use preprocessors in a disciplined manner. Instead of textual comments, it uses background colors to represent features. Thus, given that we already have a SPL implemented with CIDE, our tool can automatically compute the emergent interfaces on demand to support developers that need to maintain a SPL.

As mentioned, the developer must first select code that he wishes to maintain. Based on this selection, the tool computes interfaces for the *Selection* and dataflow components. It presents the *Selection* interface to the developer, in order to prevent him of breaking feature modularity — that is, breaking a feature that he is not concerned while performing the maintenance task. Figure 7 illustrates our tool showing an emergent interface to the developer. As Figure 7 points out, there is hidden feature code, associated with *Sorting* and *Copy* features, respectively. After the developer indicates the code snippet to maintain, the tool shows the emergent interface related to that code — stating that `controller` is provided with a certain value to *Copy*.

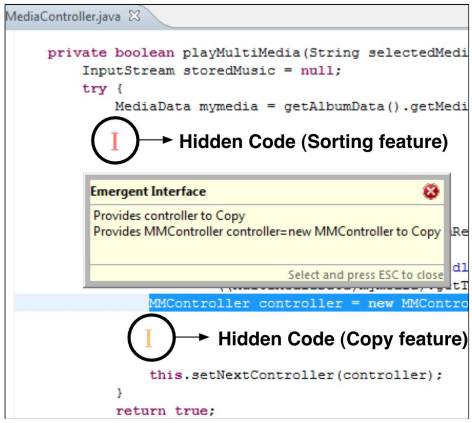

Figure 7. xCIDE screenshot.

4.1 Implementation

Our implementation relies on the three steps showed in Section 3 plus another one to show the emergent interface. The first one uses the Eclipse Java Development Tool (JDT) [1] to retrieve the AST of the selected code and compute the *Selection* component. Based on this selection, we then proceed to retrieve the dataflow components with the aid of Soot [2], a Java optimization framework for analyzing and transforming Java bytecode. This way, we have computed the *Selection* as well as the dataflow components.

The second step consists of computing the interfaces. The AST retrieved is important to compute the Selection Elements List (SEL). We use this list as an input to Soot, which is also used to compute the interfaces, where we analyze the dataflow components of our interest. This analysis relies on a variety of algorithms like the ones presented in Section 3.2. Roughly, since CIDE uses background colors to represent feature expressions, these algorithms search for elements of SEL which are in different colors of the feature expression associated with the *Selection* component. If found, we consider them in the *Selection* component interface. Then, we take this interface as an input for Soot to compute each dataflow interface. We match them (third step) and show the *Selection* interface to the user (fourth step).

4.2 Limitations and Ongoing work

Our tool currently implements the general algorithm to emerge interfaces. The main limitation when computing interfaces happens when we have mutually exclusive features. The tool searches SEL elements in all feature expressions (all colors), whether mutually exclusive or not. Improving this computation is an ongoing work.

We are also working on other algorithms, using both intra and interprocedural analysis. An example is the chain of assignments. In this case, changing the value of a variable `x` in the *Selection* component may produce a chain of other changes that reaches another feature. For instance, we might use the new value of `x` to define the value of `y` which, in

[2] The tool is available at `http://www.cin.ufpe.br/~mmr3/onward10`

its turn, defines the value of `z`. If another feature uses `z`, changing the value of `x` can cause problems to this feature.

5. Related work

Interfaces for non-annotative approaches. This work focuses on interfaces for techniques that annotate code to define feature boundaries, such as conditional compilation. Since this leads to scattering and tangling, researchers evaluated the use of Aspect-Oriented Programming [3, 4, 15] to solve these problems. However, because of problems like fragile pointcuts [19], researchers proposed interfaces between classes and aspects to achieve modularity.

Griswold et al. [8] proposed Crosscutting Programming Interfaces (XPIs) aiming at decoupling the aspects from details of classes, providing better modularity during parallel evolution. Also, there is a notion of provides and requires that XPIs may check. For example, we might define a contract in which aspects cannot change the state of some object. We can write XPIs using AspectJ language constructs. Thus, components and interfaces have a rigid structure: classes, aspects, and XPIs. Unlike XPIs, our approach does not predefine components and interfaces. They emerge on demand, according to a maintenance task. In addition, since emergent interfaces are not written, they do not need language constructs but tools responsible for generating them. Like XPIs, we abstract details of features, being important to make developers focus on the maintenance task.

The AspectScope tool [9] realizes the idea of aspect-aware interfaces in AspectJ [14]. It performs whole-program analysis of AspectJ programs and displays module interfaces according to current deployment of aspects. It aims to help developers understand program behavior with local reasoning. Their concept of presenting interfaces to the developer is similar to what we propose in this work, aiming to facilitate modular reasoning through tool support. However, AspectScope module interfaces are not product-line-aware. As we do, AspectScope provides a visualization of interfaces.

Separation of Concerns. Some approaches aim to provide separation of concerns by hiding information. Mylyn [13] is a task-focused approach to reduce information overload, so that only artifacts (like packages, classes, and methods) relevant to a current task are visible. A task context, created during a programming activity, filters this information. This way, Mylyn monitors tasks aiming at storing information about what developers are doing to complete the task. If the task is not completed, developers can continue them afterwards. When opening the IDE to complete that task, instead of showing thousands of artifacts, developers may select the task and Mylyn provides only the artifacts related to it, improving productivity (developers do not spend time searching for the artifacts of that task) and reducing the information overload. Like Mylyn, our approach also needs a selection. Developers select the snippet in order to maintain it, whereas when using Mylyn they select tasks. Our interfaces and the task context of Mylyn emerge during maintenance. Finally, we also provide information reduction, since we only show elements shared with other features to the developer through the *Selection* interface.

Colored IDE (CIDE) is a tool for decomposing legacy applications into features [12]. Although CIDE uses the preprocessors semantics (based on the same annotative approach), it avoids pollution of code, which means that *#ifdef* directives are no longer needed. Instead, it relies on the Eclipse editor to define the features boundaries through background colors. CIDE relies on VSoC, so that it is possible to hide code of features not interesting to the current maintenance task. We presented an extension of this tool to improve feature modularization. Our intent is to make developers aware about other features *before* initiating their maintenance tasks. Also, emergent interfaces show the exact product configurations that a maintenance may affect.

Conceptual Module [5] is an approach to support developers on maintenance tasks. They set lines of code to be part of a conceptual module and use queries to capture other lines that should be part of it and to compute dependencies among other conceptual modules. We also catch dependencies, but we go beyond since we consider features relationships. Both approaches abstract details from developers so that they concentrate on relationships among features or conceptual modules rather than on code of no interest, being important for comprehensibility. However, our interest lies not only on providing dependencies, but also information that may be useful during maintenance. For example, interfaces may indicate that hidden features have statements like `continue`, `break`, `throws`, and `return`. Now, developers are aware about possible control flow changes during maintenance.

Safe composition. Safe composition relates to safe generation and verification of properties for SPL assets: i.e., providing guarantees that the product derivation process generates products with properties that are obeyed [11, 20]. Generating all SPL products to check safe composition turns out to be impractical as the SPL becomes larger.

Thaker et al. present techniques for verifying type safety properties of product lines using FMs and SAT solvers [20]. They extract properties from feature modules and verify that they hold for all SPL members. Safe composition is also proposed for the Color Featherweight Java (CFJ) calculus [11]. This calculus establishes type rules to ensure that CFJ code only generates well-typed programs. CIDE — the tool we extended — implements this formalization.

These works check for type errors on SPL products, being similar to the matching interfaces step of our algorithm, where we catch some of these errors. However, our intent is to use emergent interfaces to prevent errors when maintaining features. Moreover, some elements in our emergent interfaces deal with the system behaviour (value assignment), rather than only with static type information.

6. Concluding Remarks

This paper introduced the emergent feature modularization concept, which might be applied to maintain features in product lines. We call our approach emergent since we do not rely on components and interfaces with a rigid structure, meaning that they are not predefined. Instead, they emerge on demand to support developers when maintaining features.

Our interfaces abstract details from features that are not relevant to the current task (the hidden ones), but at the same time provide valuable information to maintain a feature and keep these hidden ones safe. Because of this abstraction, developers still have the benefits provided by VSoC, in the sense that the feature code of no interest continues hidden.

We also presented a three-step algorithm to compute emergent components and interfaces, implemented in xCIDE. The tool uses the emergent feature modularization concept, so that after a selection, it shows an emergent interface to the developer, keeping him informed about the contracts between the selected feature and the other ones.

As future work, we intend to improve our tool with more robust emergent interfaces. Also, we should conduct an experiment to evaluate our proposal to verify its advantages/disadvantages in terms of developer's productivity.

Acknowledgments

This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES[3]), funded by CNPq and FACEPE, grants 573964/2008-4 and APQ-1037-1.03/08

References

[1] Eclipse Java Development Tools, January 2008. http://www.eclipse.org/jdt/.

[2] Soot: a Java Optimization Framework, April 2010. http://www.sable.mcgill.ca/soot/.

[3] 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.

[4] M. Anastasopoulos and C. Gacek. Implementing Product Line Variabilities. In *Proceedings of the 2001 Symposium on Software Reusability (SSR'01)*, pages 109–117, New York, NY, USA, 2001. ACM Press.

[5] E. L. A. Baniassad and G. C. Murphy. Conceptual module querying for software reengineering. In *Proceedings of the 20th International Conference on Software Engineering (ICSE'98)*, pages 64–73, Washington, DC, USA, 1998. IEEE Computer Society.

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

[7] E. F. et. Al. Evolving software product lines with aspects: an empirical study on design stability. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, pages 261–270, New York, NY, USA, 2008. ACM.

[8] W. G. Griswold, K. Sullivan, Y. Song, M. Shonle, N. Tewari, Y. Cai, and H. Rajan. Modular Software Design with Crosscutting Interfaces. *IEEE Software*, 23(1):51–60, 2006.

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

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

[11] 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)*, pages 258–267. IEEE Computer Society, September 2008.

[12] 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.

[13] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *Proceedings of the 14th International Symposium on Foundations of Software Engineering (FSE'06)*, pages 1–11, New York, NY, USA, 2006. ACM.

[14] G. Kiczales and M. Mezini. Aspect-Oriented Programming and Modular Reasoning. In *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, pages 49–58. ACM Press, 2005.

[15] G. C. Murphy, A. Lai, R. J. Walker, and M. P. Robillard. Separating features in source code: an exploratory study. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE'01)*, pages 275–284, Washington, DC, USA, 2001. IEEE Computer Society.

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

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

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

[20] S. Thaker, D. S. Batory, D. Kitchin, and W. R. Cook. Safe composition of product lines. In *Proceedings of the 6th International Conference Generative Programming and Component Engineering, (GPCE'07)*, pages 95–104. ACM, 2007.

[21] 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.

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

[3] http://www.ines.org.br