Emergo: A Tool for Improving Maintainability of Preprocessor-based Product Lines

Márcio Ribeiro[1,2] Társis Tolêdo[1] Johnni Winther[4] Claus Brabrand[1,3] Paulo Borba[1]

[1] Federal University of Pernambuco, Av. Prof. Luis Freire, 50.740-540, Recife, Brazil
[2] Federal University of Alagoas, Av. Lourival de Melo Mota, 57.072-970, Maceió, Brazil
[3] IT University of Copenhagen, Rued Langgaards Vej 7, DK-2300, Copenhagen, Denmark
[4] Aarhus University, Nordre Ringgade 1, DK-8000, Aarhus, Denmark

{mmr3, twt, phmb}@cin.ufpe.br, jw@cs.au.dk, brabrand@itu.dk

Abstract

When maintaining a feature in preprocessor-based Software Product Lines (SPLs), developers are susceptible to introduce problems into other features. This is possible because features eventually share elements (like variables and methods) with the maintained one. This scenario might be even worse when hiding features by using techniques like Virtual Separation of Concerns (VSoC), since developers cannot see the feature dependencies and, consequently, they become unaware of them. Emergent Interfaces was proposed to minimize this problem by capturing feature dependencies and then providing information about other features that can be impacted during a maintenance task. In this paper, we present Emergo, a tool capable of computing emergent interfaces between the feature we are maintaining and the others. Emergo relies on feature-sensitive dataflow analyses in the sense it takes features and the SPL feature model into consideration when computing the interfaces.

Categories and Subject Descriptors D.2.3 [*Software Engineering*]: Coding Tools and Techniques; D.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages —*Program analysis*

General Terms Design

Keywords Software Modularity, Software Product Lines, Dataflow Analysis.

AOSD'12, March 25?30, 2012, Potsdam, Germany.

1. Introduction

Features in Software Product Lines (SPLs) are often implemented using preprocessors [5]. Conditional compilation directives such as `#ifdef` and `#endif` encompass code associated with features, mixing common, optional and even alternative behavior in the same code asset. Despite their widespread use, preprocessors have drawbacks, including no support for separation of concerns [10]. Virtual Separation of Concerns (VSoC) [4] allows developers to hide feature code not relevant to the current task, reducing some of the preprocessors drawbacks. The idea is to provide developers a way to focus on a feature without being distracted by others.

However, VSoC is not enough to provide feature modularization, which aims at achieving independent feature comprehensibility, changeability, and development [7]. In fact, by visualizing and trying to maintain a feature individually, a developer might introduce errors in other features since they possibly share elements—such as variables and methods—with the maintained feature, henceforth denominated feature dependencies. Thus, we face the lack of feature modularization: because of feature dependencies, the new value of a variable, for example, might be correct to the maintained feature, but incorrect to another one that uses the same variable. So, there is no "*mutual agreement between the creator and accessor*" [12]. In fact, these code feature dependencies are quite common in practice [9].

To minimize these problems, we proposed the concept of emergent interfaces [8]. The idea consists of capturing dependencies between the feature a programmer is maintaining and the others. These interfaces emerge and provide information about other features that might be affected by maintenance tasks. Developers then become aware of the feature dependencies and, consequently, might avoid the introduction of errors to the SPL, thus being important for feature modularity. Notice that developers still have the VSoC benefits. Emergent interfaces complement VSoC in that in addi-

tion to hiding feature code, they provide dependencies information.

In this context, we present Emergo[1], an Eclipse plugin for computing emergent interfaces. Given a product line implemented in Java that uses tools like Antenna [1] to annotate features using `#ifdefs` and a set of maintenance points by using the Eclipse Java editor, Emergo is capable of computing emergent interfaces with respect to those points by using dataflow analyses. However, the analyses we consider are feature-sensitive [3], in the sense that they take preprocessor directives and the product line feature model into consideration. Emergo also implements the VSoC concept. To do so, it provides projections where developers can collapse and expand feature code. To display the information, Emergo provides Eclipse views based on tables and graphs.

2. Emergent Interfaces with Emergo

Emergo can compute emergent interfaces based on feature dependencies between methods or within a single method, by using interprocedural or intraprocedural dataflow analysis, respectively. To generate emergent interfaces, the developer firstly selects the maintenance points and then invokes the tool for the interfaces. Figure 1 illustrates Emergo. The developer is maintaining the mandatory feature (with no `#ifdefs`). She selected `AbstractController nextcontroller = this` as the maintenance point. In this example, there are two optional features involved: `COPY` and `SMS`. The *Emergo Table View* presents the emergent interface as a table (see the bottom of Figure 1) and the *Emergo Graph View* presents the interface as a graph (see the right-hand side of Figure 1). Both views present all product configurations that might be impacted by the maintenance points. This way, developers can analyze the interface and then proceed with the maintenance task, but now aware of the dependencies. Developers can navigate throughout the code by using both views by clicking on any interface element. When this happens, the editor moves the cursor to the class and source line that represents such an element.

In this particular example, changing the `nextcontroller` value would impact two product line configurations: `(!COPY & SMS)` and `COPY`. Both features use `nextcontroller`. However, since `COPY` assigns a new value to such a variable, changing `nextcontroller` would only impact `SMS` in cases where `COPY` is not present.

3. Architecture

Figure 2 depicts the architecture of our tool. In this section, we detail each element of the architecture.

In order to compute feature dependencies and then generate emergent interfaces, we can actually use conventional dataflow analysis. In this way, we generate all possible product configurations and analyze them individually. However, the cost to generate the interfaces by using this idea is high, especially in case of many features within the SPL. To minimize the costs, we make dataflow analysis feature-sensitive [3], so that our dataflow analyses are capable of analyzing all configurations simultaneously: there is no need to generate and compile all possible product configurations.

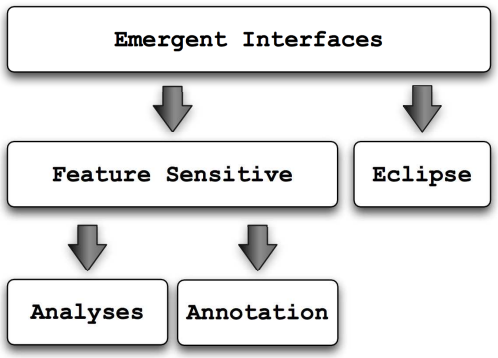

Figure 2. Emergo architecture.

The *Emergent Interfaces* component depends on the *Feature Sensitive* component to execute dataflow analyses in a feature-sensitive way. The *Feature Sensitive* component then executes feature-sensitive *Analyses*. However, such a component needs to identify which code elements are encompassed by which features. We read this information by using the *Annotation* component. Internally, such component instruments the nodes of the AST with feature information. This information from the instrumented AST is then retrieved to feed the *Feature Sensitive* component to proceed with the analyses [3, 11].

After computing dataflow analysis for each product configuration, we need to generate the emergent interfaces. The *Emergent Interfaces* component is responsible for this task. First, it reads the information computed by the Feature Sensitive component to obtain dataflow information for each possible configuration. Then, it computes the emergent interface by crossing the obtained information with the maintenance points. The result consists of a graph of dependencies that associates the maintenance points with points of the program that might be impacted by the formers. Finally, it displays the interface to the developer in the Eclipse views.

4. Related work

Colored IDE (CIDE) is a tool for decomposing legacy applications into features [5]. Although CIDE uses the preprocessors semantics (based on the same annotative approach), it avoids pollution of code, which means that `#ifdefs` 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. Emergo extends CIDE in the sense we improve fea-

[1] `http://www.cin.ufpe.br/~emergo`

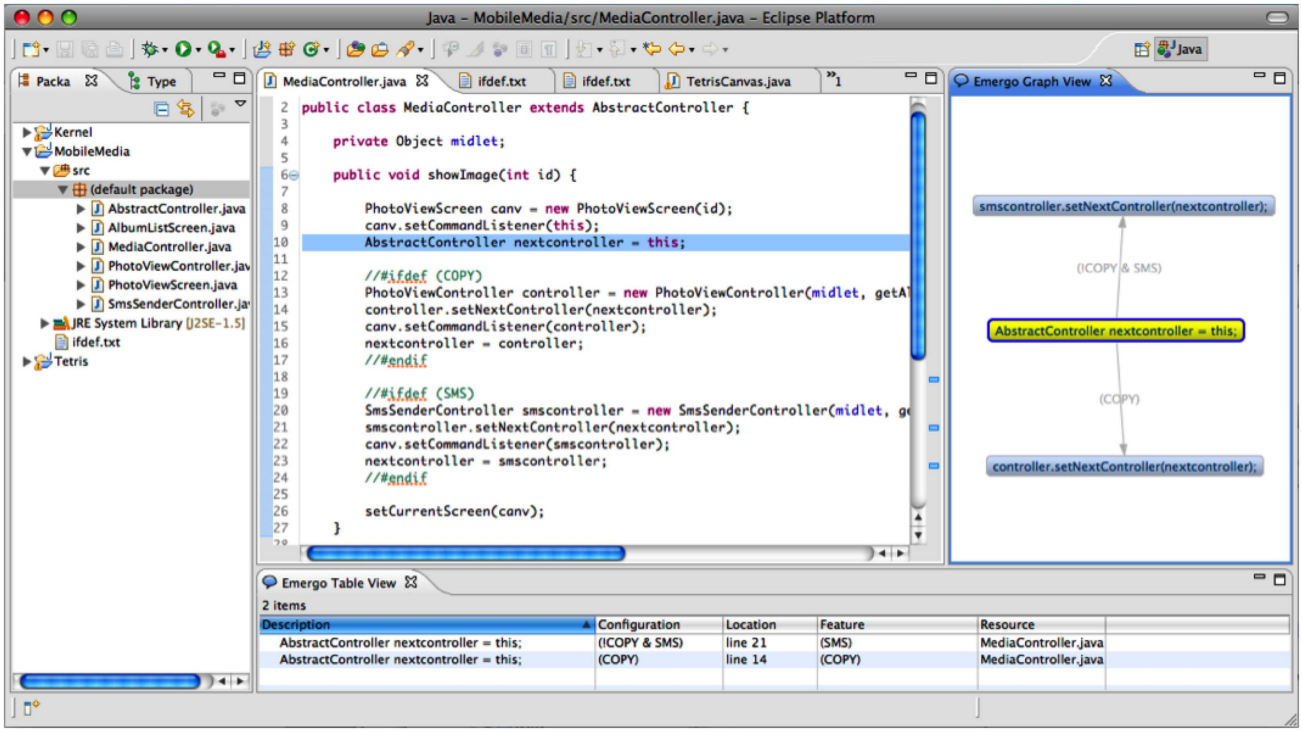

Figure 1. Emergo.

ture modularization, since it computes emergent interfaces, which is important to make developers aware of feature dependencies. Like CIDE, we also provide VSoC by using code projections, which helps on improving feature comprehensibility. We firstly used CIDE to implement our feature-sensitive dataflow analyses [3]. Therefore, we used this idea in product lines based on `#ifdefs` and colors. So, this can suggest we can apply our approach regardless the preprocessor-based annotation mechanism.

Mylyn [6] is a task-focused approach to reduce information overload through information hiding, so that only assets (like packages, classes and methods) relevant to a current task are visible. This information is filtered by using a task context that is created during a programming activity. This way, tasks are monitored by Mylyn aiming at storing information about what developers are doing to complete the task. Developers can select a task and Mylyn provides only the assets related to it, improving productivity. Like Mylyn, when using Emergo, developers also need to select code. The developer selects the snippet to maintain it, whereas when using Mylyn developers select tasks. Like Mylyn, Emergo also provides information reduction, since it focuses on the potential impacted features. So, developers do not need to open and analyze the others.

Conceptual Module [2] is an approach to support developers on maintenance tasks. Developers can 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. Our tool also captures dependencies, but goes further, since it takes features into consideration. Both approaches abstract details from developers so they concentrate on relationships among features or conceptual modules rather than on code of no interest, which is important for comprehensibility.

5. Concluding remarks

This paper presented Emergo, an Eclipse-based plugin for computing emergent interfaces in preprocessor-based SPLs. To do so, Emergo uses feature-sensitive dataflow analyses. When considering them, we analyze all product configurations at once, instead of generating all possible configurations and analyzing them individually.

After executing these analyses, it captures the dependencies between the feature a developer is maintaining and the others, being essential to build the emergent interface. Then, it shows such an interface in terms of tables and graphs.

Acknowledgments

We would like to thank CNPq, FACEPE, and National Institute of Science and Technology for Software Engineering (INES), for partially supporting this work. Also, we thank SPG[2] members for the fruitful discussions about this paper.

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

References

[1] Antenna preprocessor, January 2012. http://antenna.sourceforge.net/.

[2] 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.

[3] C. Brabrand, M. Ribeiro, T. Tolêdo, and P. Borba. Intraprocedural dataflow analysis for software product lines. In *Proceedings of the 11th International Conference on Aspect-Oriented Software Development (AOSD'12)*, Potsdam, Germany, 2012. ACM. To appear.

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

[5] 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.

[6] 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.

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

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

[9] 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 (GPCE'11)*, pages 23–32, Portland, Oregon, USA, 2011. ACM.

[10] 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.

[11] J. Winther. Experimental java compiler, January 2012. http://users-cs.au.dk/jwbrics/java/.

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