CrossFIRE: An Infrastructure for Storing Crosscutting Framework Families and Supporting their Model-Based Reuse

Rafael S. Durelli[1], Thiago Gottardi[2] and Valter V. de Camargo[2]

[1]Computer Systems Department University of São Paulo
São Carlos, SP, Brazil.

[2]Computing Departament
Federal University of São Carlos (UFSCAR)
São Carlos, SP, Brazil.

rdurelli@icmc.usp.br[1], {thiago_gottardi, valter}@dc.ufscar.br[2]

Abstract. *A Crosscutting Framework (CF) is an abstract design of a single Crosscutting Concern (CC) which was designed for being reused. A Crosscutting Framework Family (CFF) is a set of features whose composition results in a CF. As CFs encapsulate just one CC, their usefulness occurs when there are a number of them available for being reused. However, despite the number of CFs proposed in the last years, there is no a suitable repository in which they can be uploaded and made available. Furthermore, most of them employ white-box strategies in their reuse process, requiring significant technical knowledge to reuse them. To cope with these problems, we put forward the CrossFIRE, which is an infrastructure that allows to store, search, view and reuse CFs. Thus, domain engineers can make their CFFs available and the application engineers can search this repository, select the CFF features and reuse them in a model-based fashion.*

1. Introduction

Crosscutting Framework (CF) is a term we have proposed in a previous work to represent an abstract aspect-oriented design of a crosscutting concern (CC), e.g., persistence, security and distribution [Camargo and Masiero, 2005]. The reuse process of CFs has two steps; instantiation and composition. The instantiation involves variability selection and hook implementation, while the composition involves providing composition rules to couple the chosen variabilities to a base code. Another important term in this paper is Crosscutting Framework Families (CFFs) [Camargo and Masiero, 2008]. A CFF is a set of features in which their composition results in a CF, that is, a member of this family is a CF. The difference between CF and CFF is that, in the later, there are several features which can be composed before starting the reuse process, while in the former there is no feature selection - the framework is ready to be used.

Most of the CFs found in literature apply white-box reuse strategies, relying on writing source code to reuse the framework [Bynens et al., 2010; Camargo and Masiero, 2005; Cunha et al., 2006; Kulesza et al., 2006; Sakenou et al., 2006]. White-box strategy makes application engineers to worry about low level implementation details during the reuse process, leading to the following problems: (*i*) the need to know coding details regarding the programming paradigm employed in the framework, making the learning

curve steeper; (*ii*) coding mistakes are more likely to happen when the reuse code is created manually; (*iii*) several lines of code must be written for the definitions of small number of hooks, impacting development productivity and (*iv*) reuse process can only be started during implementation phase, as there is no source code in earlier phases.

To overcome the described problems, we put forward an infrastructure which supports (*i*) the storage of CFFs/CFs, (*ii*) the feature selection and (*iii*) the model-based reuse process of CFs. In our infrastructure, domain engineers can make their CF/CFF available and application engineers can use one or more CFs in their application development in a model-based fashion.

There are two models that support the reuse process: Reuse Requirements Model (RRM) and Reuse Model (RM). The RRM documents all the information needed to perform the composition of a CF. The RM is instantiated along with the framework member, thus, it only contains the composition information related to the selected features. This model is also employed to conclude the reuse process in our model-driven approach by filling the fields of this model, which is also used for code generation. The motivation is the following: (*i*) there is neither a repository, in which the CFs/CFFs can be stored by domain engineer in order to share it, nor an infrastructure where the CFs can be reused as easy as possible in order to disseminate them; (*ii*) most of the CFs found apply white-box reuse strategies - thus it is important provides a way to assist the instantiation of these CFs using models. This paper is organized as followed: Section 2 provides information related to CrossFIRE - Section 3 the architecture of CrossFIRE is depicted - in Section 4 there are related works and in Section 5 we conclude the paper with some remarks and future directions.

2. CrossFIRE

In this section the Crosscutting Framework Integrated Reuse Environment (CrossFIRE) is presented. Figure 1 depicts a screenshot of CrossFIRE.

All artifacts of the CF/CFF must be developed before one uses the CrossFIRE. Thus, it is worth highlighting that CrossFIRE only supports the Application Engineering (AE) phase. Having this in mind, the domain engineer has to use a traditional approach in order to create the artifacts in the Domain Engineering (DE) phase. In the DE phase, three artifacts must be created: (*i*) source code of the whole CF, (*ii*) the RRM and (*iii*) the feature models of the CFF. These feature models have to be devised using the FeatureIDE[1], which is a tool that provides a graphical way to assemble feature models. Afterwards, the engineer can use the CrossFIRE in order to uploaded these artifacts into a repository, as is shown in Figure 1(A).

In the AE phase, an application engineer can browse the repository in order to check whether there are CFs that can be reused in order to implement concerns of the planned application. In order to assist this activity, CrossFIRE provides a visualization, which depicts all CFs/CFFs that have been uploaded by the domain engineers. Figure 1(B) shows the visualization of all CFs/CFFs available. As can be seen, there are six different CFFs - persistence, security, distribution, concurrency, logging and Business, respectively. In addition, CrossFIRE also shows descriptions for each of the selected CFFs by clicking

[1] `http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/`

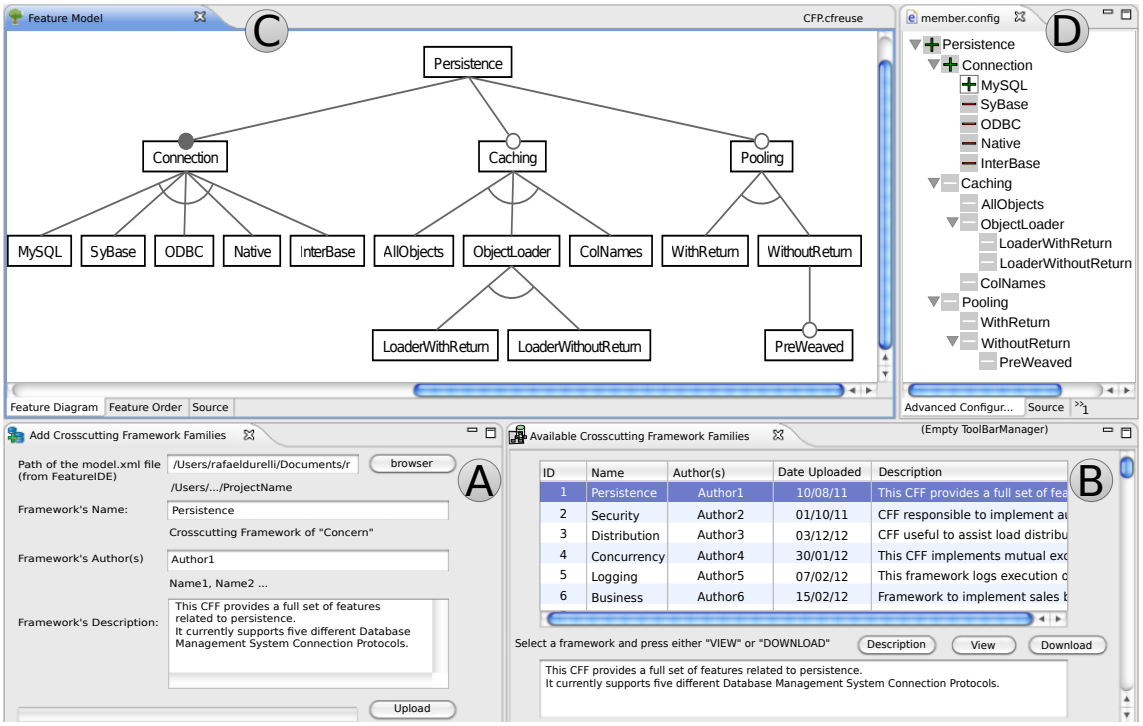

Figure 1. Screenshot of CrossFIRE

on the button “Description”. Nevertheless, if this description is not enough to help the engineer takes a decision on reusing the CF/CFF, CrossFIRE supplies a way to visualize the feature model related to selected CF/CFF by clicking on the button “View”, Figure 1(C). As is shown in Figure 1(B), the “persistence” CF is highlighted to indicate that it has been chosen. Next, the “Download” button has to be clicked to transfer the feature model belonging to the CF chosen from the remote repository to the computer of the application engineer.

Furthermore, to reuse the CFF, its features must be chosen by the engineer aiming at specifying explicitly which features will be reused in the base application. To assist this activity, CrossFIRE uses the “configuration file” of FeatureIDE. By using this “configuration file”, features can be chosen by the application engineer. The graphical notation of the “configuration file” is shown in Figure 1(D), which represents all features of the “persistence” CF. Moreover, FeatureIDE validates if the selected features match a valid combination for the instantiation of a member of the CFF. As shown in Figure 1(D), once the application engineer has chosen the features (represented by “+”), the resulting variant and constraints are generated automatically (represented by “-”).

The application engineer should provide the selected features by using the “configuration file” to repository server, which will carry out an algorithm. This algorithm aims to extract two artifacts. The first extracted artifact only contains code related to the selected features. The second extracted artifact is a RM derived from a RRM, by removing unrelated requirements. The RM is used to support the reuse process of a CF. After that, these artifacts are sent to the application engineer’s computer.

To reuse a member of the CFF or any other CF persisted in the repository, the ap-

plication engineer may use the RM. It is graphically represented as a form which contains fields that should be filled with information regarding the base application. By completing this form, the code needed to couple the CF to the base application can be generated. It is possible to see our model editor in Figure 2. The "Palette" on the right of the figure contains the elements of the RMs. They are: "Group": an element to group any element visible in the models; "Pointcut": represents join-points of the base application; "TypeExtension": represents types found in the base application; "Value": represents any numeric or textual values that must be informed while reusing the CF; "Option": defines a selectable variability of the framework and "OptionGroup": group selectable variabilities of the framework.

Each box contains a name and a description for the required information. The last line should be filled to provide the information regarding the base application. Note that the last line is only used in RMs. For example, to be able to instantiate a persistence CF, the application engineer must specify methods from base application that should be executed after a database connection is opened and before it is closed. Then, the box "Connection Opening" in Figure 2 represents names of methods that need an open database connection. It is also needed to specify methods that represent data base transactions, and the variabilities must be chosen, e.g., the driver which should be used to connect to the database system. After filling the fields of the RM, a model transformation generates the code needed to couple the CF, and then, the reuse process is complete. Our tool also provides validation of the information filled into RM models. At the moment this paper was written, only AspectJ CFs are supported.

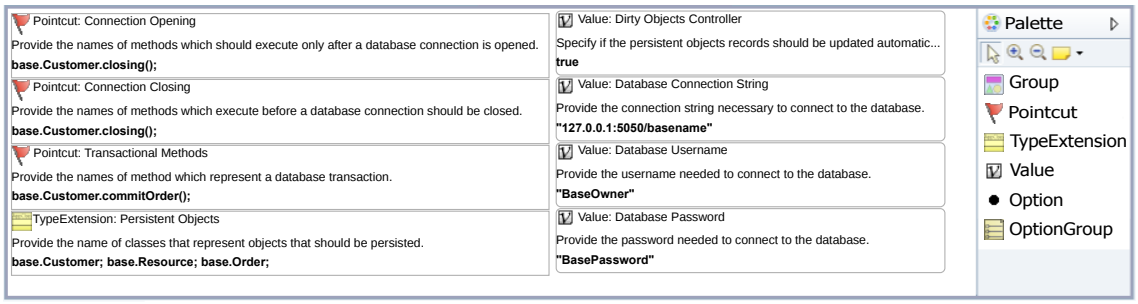

Figure 2. Reuse Model Editor

3. Architecture

In Figure 3 is depicted the architecture of the CrossFIRE. As shown in this figure, we devised the CrossFIRE on the top of the Eclipse Platform. On the other hand, both the RRM and RM have been developed using Eclipse Modeling Framework and Graphical Modeling Framework[2]. Moreover, we have studied FeatureIDE's source code and extended it in order to use for CF/CFF, its feature model editor and its configuration file, which is used to extract member of a CFF.

As can be seen in Figure 3, all artifacts (source code, feature model, RM and RRM) that the CrossFIRE provides are persisted in a database. These artifacts are persisted in a remote server, available to be reused in the AE phase. This remote server is a

[2]http://www.eclipse.org/modeling/gmp/

physical computer, which is dedicated to run the RESTful API. Therefore, to send these artifacts by the server we have used this API as web service to cache the representation of all artifacts. This server receives requests of the CrossFIRE through RESTful, processes database queries and sends a response to the CrossFIRE by using RESTful as well. Furthermore, we have used Java Persistence API (JPA) 2.0 to deal with the way relational data is mapped to Java objects. To implement the database of the server, the MySQL was chosen.

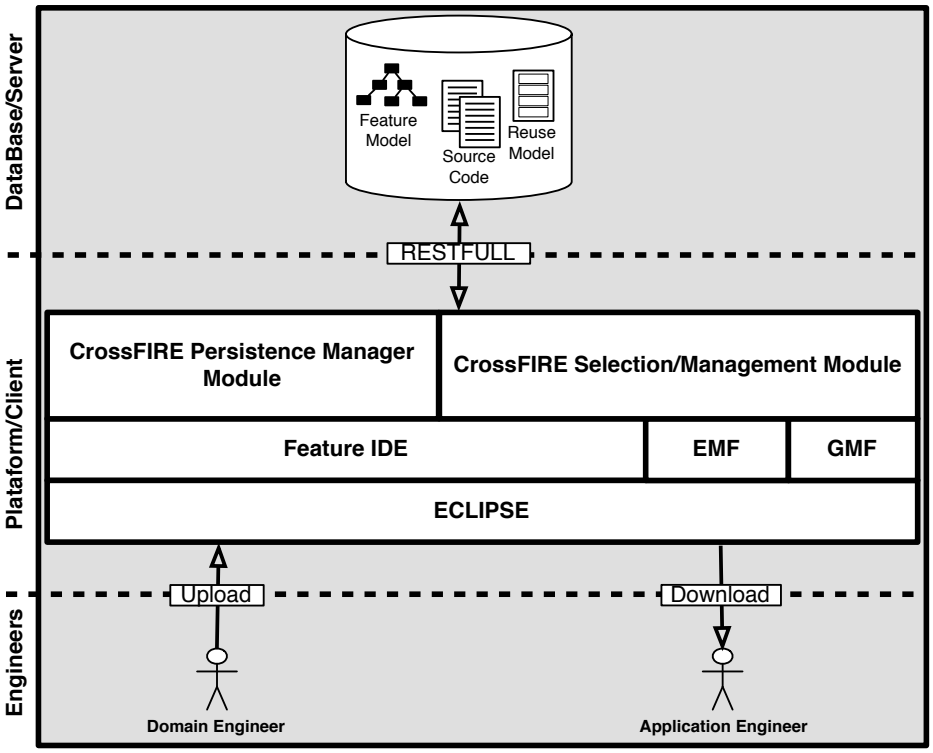

Figure 3. Architecture of CrossFIRE

4. Related Work

The approach proposed by Cechticky *et al.* [Cechticky et al., 2003] allows object-oriented application framework reuse by using a tool called OBS Instantiation Environment. That tool supports graphical models do define the settings of the expected application to be generated. The model to code transformation generates a new application that reuses the framework. The proposal found in this paper differs from their approach on the following topics: (*i*) their approach is restricted to frameworks known during the development of the tool; (*ii*) it does not use aspect orientation; (*iii*) the reuse process is applied on application frameworks, which are used to create new applications.

Another approach was proposed by Oliveira *et al.* [Oliveira et al., 2011]. Their approach can be applied to a greater number of object oriented frameworks. After the framework development, the framework developer may use the approach to ease the reuse by writing the cookbook in a formal language known as Reuse Definition Language (RDL) which also can be used to generate the source code. This process allows to select the variabilities and resources during reuse, as long as the framework engineer specifies the RDL code correctly.

5. Concluding Remarks

In the DE phase, CrossFIRE provides an infrastructure in which all of the CFF artifacts is developed. Afterwards, the CrossFIRE allows the engineers share these artifacts. As a result, these artifacts will be stored in a remote repository to be reused in the AE phase.

In the AE phase the CrossFIRE shows a list of all available CFFs that have been shared. Therefore, the engineer can pick out which CFF(s) can be reused in his base application. Next, the CrossFIRE provides a way to perform the download of the feature models related to each CFF. Through these feature models the engineer can pick out which features his base application really requires. As a consequence, the CrossFIRE downloads two artifacts, the necessaries chunks of code and a RM. Thus, the application engineer fills in the RM with the information needed by an member of a CFF's, and after that, it is possible to generate the final reuse code.

We believe that CrossFIRE increases the degree of reuse and allows the engineer to avoid dead code in his base application. Moreover, this infrastructure aims make the reuse activities easier. Long term future work involves conducting experiments to evaluate the level of reuse provided by CrossFIRE. It is worth highlighting that CrossFIRE is open source and it can be downloaded at *www.dc.ufscar.br/∼valter/crossfire*.

6. Acknowledgements

The authors would like to thank CNPq for Processes 132996/2010-3, 560241/2010-0 and 483106/2009-7 and FAPESP for Process 2011/04064-8.

References

Bynens, M., Landuyt, D., Truyen, E., and Joosen, W. (2010). Towards reusable aspects: The mismatch problem. In *Workshop on Aspect, Components and Patterns for Infrastructure Software (ACP4IS'10)*, pages 17–20.

Camargo, V. and Masiero, P. (2005). Frameworks orientados a aspectos. In *Anais Do 19º Simpósio Brasileiro De Engenharia De Software (SBES'2005), Uberlândia-MG, Brasil, Outubro.*

Camargo, V. V. and Masiero, P. C. (2008). An approach to design crosscutting framework families. In *Proceedings of the 2008 AOSD workshop on Aspects, components, and patterns for infrastructure software*, pages 3:1–3:6, New York, NY, USA. ACM.

Cechticky, V., Chevalley, P., Pasetti, A., and Schaufelberger, W. (2003). A generative approach to framework instantiation. In *Proceedings of the 2nd international conference on Generative programming and component engineering*, GPCE '03, pages 267–286, New York, NY, USA. Springer-Verlag New York, Inc.

Cunha, C., Sobral, J., and Monteiro, M. (2006). Reusable aspect-oriented implementations of concurrency patterns and mechanisms. In *Aspect-Oriented Software Development Conference (AOSD'06)*, Bonn, Germany.

Kulesza, U., Alves, E., Garcia, R., Lucena, C. J. P. D., and Borba, P. (2006). Improving extensibility of object-oriented frameworks with aspect-oriented programming. In *Proc. of the 9th Intl Conf. on Software Reuse (ICSR'06)*, pages 231–245.

Oliveira, T. C., Alencar, P., and Cowan, D. (2011). Reusetool-an extensible tool support for object-oriented framework reuse. *J. Syst. Softw.*, 84(12):2234–2252.

Sakenou, D., Mehner, K., Herrmann, S., and Sudhof, H. (2006). Patterns for re-usable aspects in object teams. In *Net Object Days*, Erfurt.