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

Rafael S. Durelli[1,3], **Márcio E. Delamaro**[1] **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.

[3]RMoD Team, INRIA, Lille, France

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

Abstract. *Legacy systems mainly consist of two kinds of artifacts: source code and databases. Usually, the maintenance of those artifacts is carried out through refactoring processes in isolated manners and without following any kind of standardization. In order to provide a more effective maintenance of the whole system both artifacts should be analyzed, evolved jointly and standardized way. As is known refactoring is the most suitable mechanism to deal with the software aging problem since it can improve maintainability and reusability of these system. However, recent researches have been shifted from the typical refactoring processes to the so-called Architecture-Driven Modernization (ADM), which is an approach that follows the all the principles of Model-Driven Development (MDD). As one of the challenges of Software Engineering focuses on mechanisms to support the automation of software refactoring process in this paper we put forward an infrastructure to assist the modernization of a legacy system based on ADM, which uses the Knowledge Discovery Metamodel (KDM) standard. This infrastructure analyses Structured Query Language (SQL) queries embedded in a legacy source code in order to restructure and re-organize the system by using design patterns, such as Data Access Object (DAO) and Service-Oriented Architecture (SOA).*

1. Introduction

Companies have a vast number of operational legacy systems and these systems are not immune to software aging problems. However, they can not be discarded because they incorporate a lot of embodied knowledge due to years of maintenance and this constitutes a significant corporate asset. Furthermore, these kind of systems mainly consist of two artifacts: source code and databases. Usually, the maintenance of those artifacts is carried out through restructuring processes in isolated manners [**?**]. Nevertheless, we claim that for a more effective maintenance of the whole system both should be analyzed and evolved jointly.

Refactoring can be used in order to deal with the maintenance of these systems. Martin Fowler [**?**] defines refactoring as “A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior”. In the literature there is a lot of studies that make refactoring at the level of source code. Empirical studies of refactoring have shown that it can improve maintainability [?] and reusability [?] of legacy system. Not only does existing work suggest that refactoring is useful, but it also suggests that refactoring is a frequent practice [?]. Cherubini and colleagues‘ survey indicates that developers rate the importance of refactoring as equal to or greater than that of understanding code and producing documentation [?].

In a parallel research line, researchers have been shifted from the typical refactoring process to the so-called Architecture-Driven Modernization (ADM) [?]. ADM has been proposed by OMG (Object Management Group) with the concept of modernizing legacy systems with a focus on all aspects of the current system architecture and the ability to transform current architectures to target architectures. In addition, ADM advocates conducting reengineering processes following the principles of Model-Driven Development (MDD) [?], i.e., it treats all the software artifacts involved in the legacy system as models and can establish transformations among them. Firstly a reverse engineering is performed starting from the source code and a model instance (**P**lataform **S**pecific **M**odel - PSM) is created. Next successive refinements (transformations) are applied to this model up to reach a good abstraction level (PSM) in model called KDM (**K**nowledge **D**iscovery **M**etamodel). Upon this model, several refactorings, optimizations and modifications can be performed in order to solve problems found in the legacy system. Secondly a forward engineering is carried out and the source code of the modernized target system is generated again. According to the OMG the most important artifact provided by ADM is the KDM metamodel, which is a multipurpose standard metamodel that represents all aspects of the existing information technology architectures. The KDM is divided into four layers representing both physical and logical software assets of information systems at several abstraction levels. Each layer is further organized into packages. Each package defines a set of meta-model elements whose purpose is to represent a certain independent facet of knowledge related to existing legacy systems. However, in this paper we are only interested in both Program Elements layer and Runtime Resources layer, used to represent a language-independent intermediate representation for programming languages and useful to represent information such as database, respectively.

Accordingly, one of the challenges of Software Engineering focuses on specifications, architectures and mechanisms to support the automation of software reengineering process, aiming to reduce time and effort spent in this process. Furthermore, legacy systems, often with high maintenance costs due to scarcity or absence of documentation, may be restructured and ported to ADM, providing greater quality in development and maintenance of these systems. Motivated by these ideas, we put forward an infrastructure based on the metamodel KDM which support legacy system reengineering from databases queries and source code, with the objectives of reducing the time and effort in this process and providing migration of these systems to ADM. More specifically, this infrastructure analyses Structured Query Language (SQL) queries embedded in a legacy source code in order to restructure and re-organize the system by using design patterns, such as Data Access Object (DAO) and Service-Oriented Architecture (SOA). This paper is organized as followed: Section 2 provides information related to the infrastructure - Section 3 the architecture of the infrastructure is depicted - in Section 4 there are related works and in Section 5 we conclude the paper with some remarks and future directions.

2. NOME DA INSFRA

In this section our infrastructure is presented. The theory behind it is based on the horseshoe modernization model, which is depicts in Figure 1. As can be seen, this model consist of three steps, **Reverse Engineering**, (*ii*) **Restructuring**, and **Forward Engineering**. The first step is represented by the left side of the horseshoe, it analyzes the legacy system in order to identify the components of the system and their interrelationships. Usually the reverse engineering step builds one or more representations of the legacy system at a higher level of abstraction (PSM). The second one is represented by the curve of the horseshoe since this steps takes the previous system's representation (PSM) and transforms it into another one (**P**lataform **I**ndependent **M**odel - PIM) at the same abstraction level. Finally, the last step is represented by the right side of the horseshoe because it generates physical implementations (source-code) of the target system at a low abstraction level from the previously restructured representation of the system.

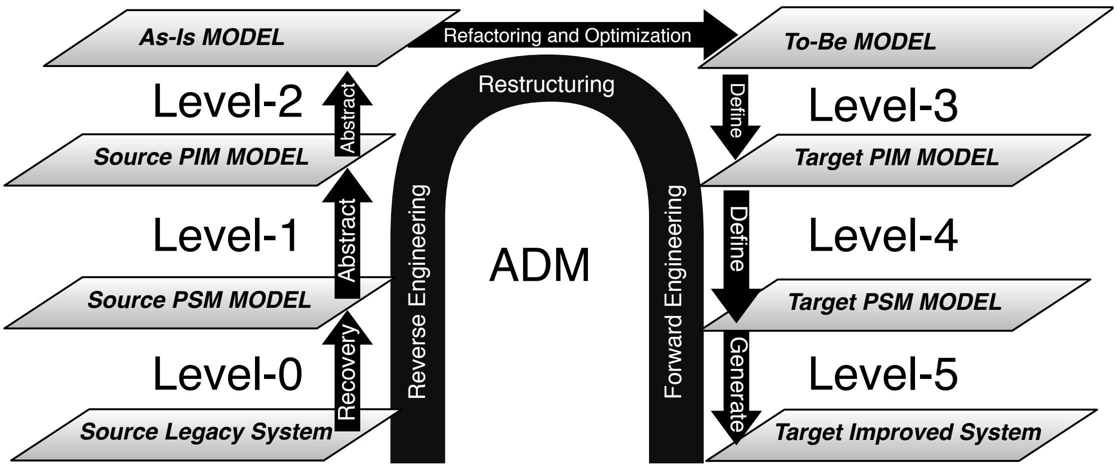

Figure 1. Horseshoe modernization model.

It worth noticing that KDM is the most essential part of our infrastructure. KDM supplies the representation and management of knowledge extracted by means of reverse engineering from all the different software artifacts of the legacy system. Therefore, the legacy knowledge obtained is then modernized into a target improved system by using the concept of MDD. As the infrastructure herein follows the three steps depicts in Figure 1 then it has to go through five abstraction levels with four transformations among them. In Figure 2 is depicted the infrastructure and also illustrates all abstract levels and the transformations (see the letters 'A'- 'F'). The Section 2.1 describes the abstract levels and the Section 2.2 explains the transformations.

2.1. Abstract Levels

The **Level-0** represents all artifacts of the legacy systems in the real world, such as source code and database. The infrastructure uses this artifacts as input to start the ADM process.

The **Level-1** consist of two PSM. The former model is a source code model, which represents all abstraction of the legacy system's source code, Figure 2(B) illustrates it. More specifically, this model represents an **A**bstract **S**yntax **T**ree (AST), which is a tree representation of the abstract syntactic structure of Java source code. The latter is a database model that illustrates informations related to database used in the legacy system, see Figure 2(A). Similarly, this model is also represented by an AST, the only difference is that it represents syntactic structure of database instead of Java source code. This database model is based on the SQL-92 Data Manipulation Language which can represent the SQL operations such as *Select*, *Delete*, *Update* and *Insert*. In order to transform the legacy system in these models it is necessary carry out a syntactical analyzer. As for the first model we used a parser that provides a complete self-describing representation of Java source code, i.e., this representation reflects the structure of the software artifact directly in the nesting of elements. Regarding the second model we devised a SQL parser. It consists of a extension of the previously parser but it intends to identify SQL embedded in the legacy system's source code. This parser focus on identifying SQL operations such as *Select*, *Delete*, *Update* and *Insert*; the tables, the columns, primary keys, etc, that appear in these SQL operations are then represented into a AST. Both models, the source code model and the data base model are represented in the XMI (**XM**L Metadata **I**nterchange)-based syntax as can be seen in Figure 2 (B) and Figure 2 (A), respectively.

The **Level-2** consist of a combination of the two PSMs (see **Level-1**) for creating a single PIM which is based on the KDM, Figure 2(C) depicts it. This single model makes possible to represent the artifacts of the legacy systems in an integrated and technological-independent way. The first PSM (source-code) it is transformed to the KDM Code Package, which represents common program elements supported by various programming languages, such as data types, classes, procedures, and templates. In the same way, the second PSM (database model) it is transformed to the KDM Data Package that represents complex data repositories, such as relational databases. Afterwards, in the **Level-3**, by using this PIM the infrastructure allows the software engineer to realize a set of refactoring, refinement and optimizations in such model in order to meet new requirements and business rules. For instance, the software engineer can pick out if he would like to refactor the legacy system to meet either Data Access Object (DAO) or Service-Oriented Architecture (SOA) or both refactorings. As result, new meta-classes may be added to this PIM model, as well as new meta-attributes and meta-relationship in order to meet these requirements.

A set of refactoring and optimizations is carried out by using this model.

This model is based on the KDM specification.

2.2. Transformation

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[1]. 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 per-

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

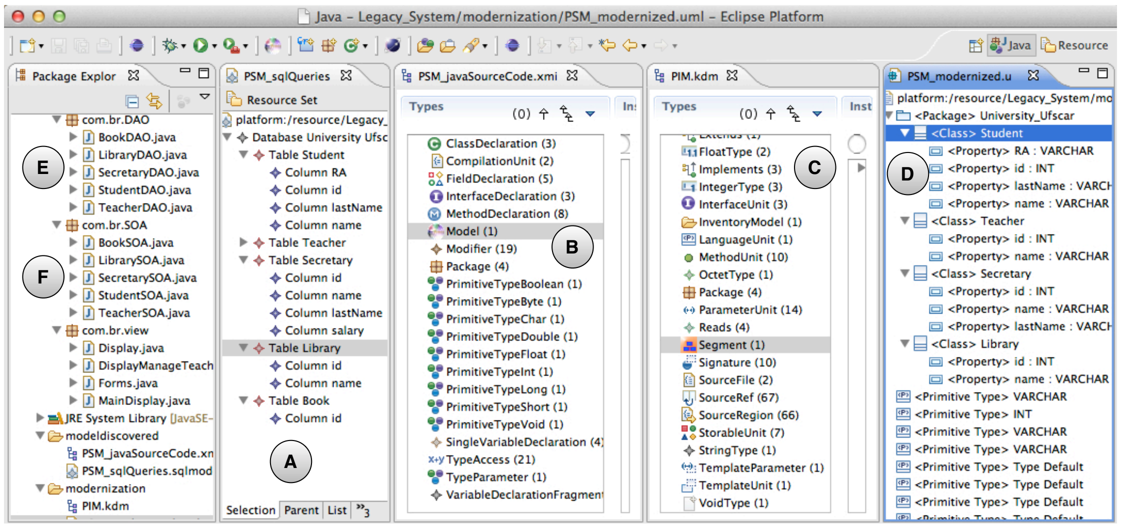

Figure 2. Screenshot of the Infrastructure

sisted in a remote server, available to be reused in the AE phase. This remote server is a 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.

4. Related Work

The approach proposed by Cechticky *et al.* [**?**] 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.* [**?**]. 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

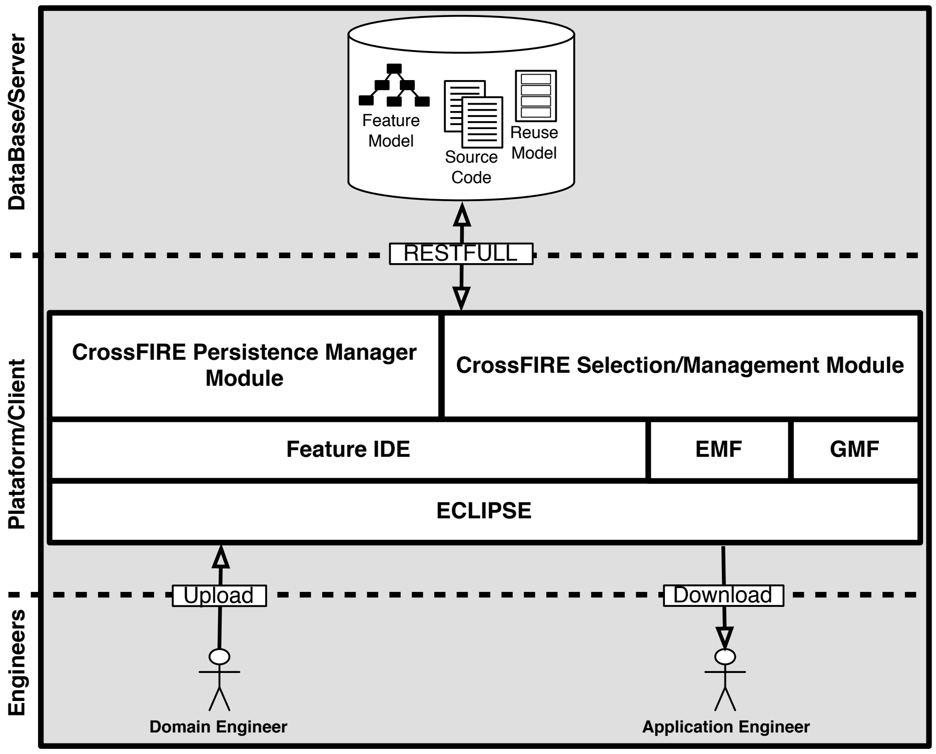

Figure 3. Architecture of CrossFIRE

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

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.

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.