Identifying Subdomains of Multiple-Domain Frameworks

Victor Hugo Santiago C. Pinto[1], Daniel G. San Martín Santibáñez[1] and Valter V. de Camargo[1]

[1] Advanced Research Group on Software Engineering (AdvanSE) – Computing Department, Federal University of São Carlos (UFSCar), Washington Luís highway, km 235 – 13.565-905, São Carlos - SP, Brazil
{victor.santiago, daniel.santibanez, valter}@dc.ufscar.br

Abstract. Multiple-Domain Frameworks (MDF) are frameworks that provide variabilities to address several domains/subdomains. MDFs present difficulties such as (i) the presence of useless variabilities in the final releases and (ii) architectural inflexibility. The former affects the productivity of application engineers as they need to live together with variabilities which are useless to their domain. The latter prevents framework engineers from composing different framework configurations to attend more constrained domains. One alternative for solving this problem is to convert them into Framework Product Lines (FPL). FPL is a Software Product Line which members are frameworks, rather than complete applications, allowing that their members being created just with the variabilities required for a specific domain. Although this conversion process seems straightforward, the most challenging activity is the identification of the MDF subdomains and their mapping to the source code. This must be done because these subdomains will turn into the main features of the resulting FPL. In this paper we present an approach to assist this activity, which is schematically represented as an algorithm. The approach was evaluated by means of its application in an MDF called GRENJ. It was restructured into an FPL and a comparative study was conducted between the two versions. The results were promising regarding the number of useless variabilities and also in terms of the composability of the resulting architecture.

Keywords: Framework, framework product lines, reusability.

1 Introduction

Frameworks are reusable tools that support application development through a process known as instantiation, which consists basically in choosing variabilities to address application requirements [11][15]. They have been extensively used for decades to support application development; some of them support the development of complete applications, such as GRENJ [9] and the ERP (Enterprise Resource Planning) developed by SAP, while others assist in the development of specific parts of applications such as Hibernate [14], Spring, JSF, etc.

Regardless the way a framework is instantiated (black, white or grey box), all of its variabilities/modules are usually carried along with the application code in the final release. For instance, if an application is developed using the Hibernate, the final release includes the object code of both the application and the whole framework, regardless of the amount of variabilities that is used. That is to say, the whole framework is kept along with the application, even if few variabilities are actually used. However, here resides an important point; since there exist the possibility of using the other non-used variabilities when the application evolves; this is not a problem, because this is a framework characteristic.

Multiple-Domain Frameworks (MDFs) is a term we have used to characterize frameworks whose boundaries go beyond just one domain, that is, they provide variabilities from several domains. In general, conventional frameworks become MDFs when they are submitted to a non-controlled and unmanaged evolution process. Since an MDF covers more than one domain, when applications are created with its support, the final release involves variabilities that will never be used by these applications. So, one inherent problem of MDFs is the presence of useless variabilities in specific sets of applications, that is, variabilities that are not likely to be used in the future. As a result, MDFs present problems for Application Engineers and Framework Engineers. Application Engineers need to live together with a vast set of variabilities and parts of them are useless to some specific domains, impacting negatively into the productivity. Framework Engineers do not manage to build smaller framework versions thanks to the architectural inflexibility of MDFs.

Framework Product Line (FPLs) is a term we have presented in a previous work [17] to represent a product line whose members are frameworks instead of complete applications. Members can be built involving just the variabilities of one domain, avoiding unnecessary features in final release of the applications developed with these members (frameworks). An important characteristic of FPLs is their flexible architecture, allowing building frameworks containing just the variabilities required for very specific domains. Therefore, FPLs fit perfectly to solve the aforementioned problems.

However, although FPLs present the ideal solution for the MDF problems, turn them into FPLs is far from trivial. The first and most challenging activity in this conversion process is the identification of all the subdomains covered by the MDF and their consequent mapping to the source code. This is an imperative task in order to build a suitable feature model and proceed to a successful restructuration. Hence, to the best of our knowledge, there is no in the literature clear guidelines to perform this activity. Most of the framework restructuration processes aim at remodularizing frameworks using other criteria, such as: crosscutting concerns, layers, lower level functionalities, etc. But, none is focused in break them down into subdomains and consider these subdomains as building blocks (features) of a product line.

In this paper, we present general guidelines, represented as an algorithm, to assist domain engineer in the first activity for turning an MDF into an FPL – we call this activity: Subdomains Identification. In order to illustrate the application of our algorithm, we apply it in the framework GRENJ. Additionally, we conducted the modularization of GRENJ and, as a consequence, we achieved an FPL called GRENJ-FPL. For evaluating our proposal, under the application engineers' perspective, we conducted a comparative study using two versions of the GRENJ-FPL (coarse-grained and fine-grained features) and the original version of GRENJ. Three applications were then instantiated using each of these reuse infrastructure and comparisons were conducted using source-code quantitative metrics. The aim was to compare the amount of useless variabilities/features in the final releases and the simplification of the source-code. Bearing that in mind, the improvements in terms of reuse, maintenance and composition using FPLs become clear.

In Section 2, the FPL concept is revisited and explained. In Section 3, we present the activities that may contribute to identifying subdomains of supposed MDF. In Section 4, we present the case study, the identification of the possible subdomains covered by framework GRENJ. In Section 5, we show the result of the modularization of GRENJ toward GRENJ-FPL and we present a comparative study among some members from GRENJ-FPL with the previous MDF in terms of reuse through the use of quantitative metrics. In Section 6, we present the related work. In Section 7 contains the conclusions and future perspectives are proposed.

2 Framework Product Lines

Framework Product Line (FPL) is a Software Product Line (SPL) [7] whose members are frameworks, rather than concrete software applications. The composition of features on an FPL results in frameworks that still need to be instantiated or coupled to concrete applications to work properly [6][17]. Figure 1 illustrates the main idea of an FPL organized in three parts: a, b and c. In the part (a) there are two FPLs with different feature granularity. In the part (b) there are some frameworks that can be generated from these FPLs. In the part (c), one can see the applications that can be generated from the frameworks.

On the left (a), there is an FPL with coarse-grained features. The common part is called *Core* and the others are subdomains *S1*, *S2* and *S3*. The reason for creating this FPL is to allow the reuse of *Core*, since there are common variabilities that may be used in both subdomains; if it were not the case, the ideal case would be three entirely independent frameworks. Note that the relationship “xor” among subdomains indicates they are separated, in other words, there is not a combination that admits more than one subdomain. On the right (b) there is the same FPL, but with fine-grained features. The difference between the first and second FPL is that in the last, some variabilities in *Core* and in the subdomains are optional. Features *C1* and *C2* and their sub features were included in *Core*, now there is the possibility of choosing which features are really important for the framework, and consequently for an application. Regarding to the constraint among features *S1*, *S2* and *S3* is still valid. One should notice that this kind of constraint is applicable for most of FPLs. However, there may be some different cases: applications that will use a few variabilities of one restricted subdomain and others that will use variabilities of more subdomains, but not from all them. For treating these different cases, it is possible to create constraints among features. For instance, in the case of “*S1* can be composed with *S2*, but not with *S3*”, if the relationship among features is “or”, it does not address the case, because *S1* and *S3* can be together. If the relationship among features is “xor”, *S1* never will be together with *S2*, then “*S1* excludes *S3*” constraint could be applied, keeping the relationship “or”. This case illustrates the constraint among subdomains, but there may be constraints of a subdomain with part of other subdomains.

On the left side (part b), there are three framework members that were derived from the first FPL. Note that there are just three valid combinations: *Core* + *S1*, *Core* + *S2* and *Core* + *S3*. On the right side (part b), there are other three framework members derived from the second FPL. Frameworks from the right side are more restricted than those from the left side. Note that this is a refined level of feature selection [2][3][8]. The aim is to create frameworks containing only features that address the requirements of an application [4] instead of using the complex structure with all available variabilities covering several subdomains.

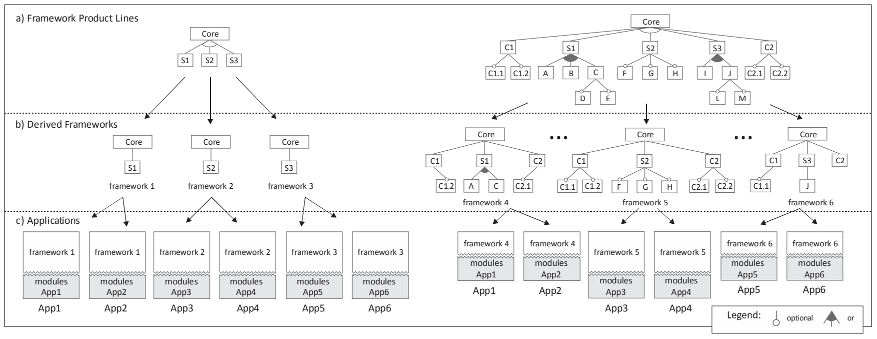

Fig. 1. Framework Product Line

Notice that in the second FPL there are other possible feature combinations which are not in the chart. The frameworks 1, 2 and 3 seem small, but each one has all variabilities of their subdomains. The frameworks 4 and 6 are more restricted than frameworks 1 and 3, since they do not have all variabilities from subdomains *S1* and *S3* and all common features from *Core*. The "framework 5" is equal to "framework 2" in terms of available variabilities: both have *Core*, *C1* and *C2* with all their sub features and *S2* with features *F*, *G* and *H*.

In part (c), it is possible to notice six applications developed with the support of derived frameworks. Note that on the left side (part c), the applications were developed with support of framework 1, 2 and 3, while on the right side (part c), the same applications were developed with support of framework 4, 5 and 6. This possibility indicates that the applications may be developed with one restrict part of subdomains. Moreover, for developing App1, features *C1.1*, *C2.2*, *B*, *D* and *E* were not necessary, even so, the framework used in that application could have *S1*+*Core* in their full form, as illustrated on the left side (part c). In that case, the framework of the final release App1 will have features not used, but since App1 may evolve over time, the remaining features might be used, as long as these features are from the same subdomain/domain of application. To clarify, if framework 4 were used to develop App1, it would be possible that the evolution of this application would require features from FPL and to add to this framework.

Based on the overview previously discussed, FPLs may be developed to address two different usage scenarios. Usage Scenario 1 whose FPL has coarse-grained features in subdomains level – this FPL enables generation of few, but large frameworks from small set of features. Usage Scenario 2 whose FPL has fine-grained features – this FPL allows both the creation of large and small frameworks.

The main point in both usage scenarios is that the FPL should be modularized in a way to not allow the generation of frameworks with features that will never be used for certain sets of applications (domain). We believe that the most common situation is the existence of FPLs with a few features for the generation of large frameworks, as indicated in Usage Scenario 1. However, if necessary, someone can build FPLs with a larger number of more fine-grained features for the provision of various frameworks with slightly different characteristics, as indicated in Usage Scenario 2.

One important point to be highlighted is that in Usage Scenario 2, when applications evolve, they may ask for features that do not exist in the restricted version of the framework. Thus, in this scenario it is important to have an "on-demand feature selection and composition" strategy. Therefore, there must be a mechanism for searching new features in the FPL, check them out and compose them to the framework. This is important but it is out of the scope of this paper.

The main motivation for Usage Scenario 1 is to identify if a framework comprehends more than one subdomain, that is, when using it to develop applications that are specific to the framework subdomain. A set of features may not be used during instantiation; nonetheless, since they belong to the same subdomain of the developed application, they are likely to be used in the future. As the granularity of the FPL features is coarse, one may create wider frameworks. This means that applications can evolve without having to seek features that are not in the framework. The number of features that are carried along with the application is much higher, though; it is a trade-off.

3 Identifying subdomains of MDFs

Identifying subdomains covered by frameworks is not always an easy task. Also, different software designers could get to different subdomains. A way to diminish this difference is through an analysis of all available information about the framework, such as documentation, usage history and applications developed with framework support and previous knowledge. If the analysis of subdomains is performed only taking into account the documentation, there may be a risk of not achieving the proper subdomains. In this paper, we present a strategy to identify the subdomains, it consists in domain analysis including existing applications. In order to clarify the subdomains identification from existing applications, we suggest an algorithm to assist the understanding of this part. Figure 2 presents the algorithm and Figure 3 schematically shows how this identification process happens.

It is important to note that this strategy is a manual process and it may be considered as a preliminary indicative for subdomain identification, that is, it does not consist in a definitive alternative, because it cannot assure that applications will not evolve and use the remaining variabilities of other subdomains. As a result, the final decision must be based on domain knowledge of framework. In this analysis we suggest to admit stable applications, i.e., applications that have already evolved over time, because if the applications are very recent, we cannot be sure that their variabilities may be used in a restrictive way by certain subsets of these applications.

In the part (I) of Figure 3 there are two sets, referred to as "F" and "A". The set "F" contains all 9 variabilities of a supposed MDF and the set "A" represents a repository containing 17 applications developed with support of this framework. The variabilities are represented by "puzzle pieces" with prefix "v" followed by a numerical value, while applications are represented by squares with prefix "a" also followed by a numerical value. Let us suppose that there is a mapping "M" that allows one to identify which variabilities from "F" are used in each application of "A". These three sets are presented as input for the algorithm depicted in Figure 2.

In Figure 2, the first step is to select each application a[n] from "A" and to search the used variabilities v[0]…v[n] on set "F" in each application. The found variabilities will form the subsets L[0]…L[n]. Then, for each application a[n], should create a subset L[n] containing all variabilities used by a[n]. The part (II) of Figure 3 shows some sets L[n] formed in this step. For instance, for application a1, the L[1] has the variabilities v1, v4, v6 and v9, because a1 was developed using them.

In the second step, the frequency on which the variabilities on subsets L[0]…L[n] are used simultaneously in the applications should be analyzed. In the third step, from this analysis and domain knowledge, the variabilities should be grouped in subsets S[0]…S[n]. For that to be done, it is also necessary to investigate if the variabilities contained in each subset are sufficient not only to support the development, but also to the evolution of these applications.

```
      Input: A is a set of applications developed with F support;
             F is a set of available variabilities from framework F;
             M is a set of relationships between F and A;
      Output: S is a set of identified subdomains;

      Begin
         for all applications (a[0]...a[n]) of the set A  do
1st         - Searching for the variabilities (v[0] ...v[n]) used in each application;
            - Creating a subset (L[0] ...L[n]) from F with the variabilities found;

2nd      search on subsets (L[0]...L[n]) the frequency which certain variabilities (v[0]...v[n])
            are being used simultaneously in the applications (a[0]...a[n])

           If it is possible to group the variabilities (v[0]...v[n]) into subsets (S[0]...S[n])
              such that most of the variabilities belong to only one of these subsets   then
3rd             - The framework has characteristics of an MDF;
                - From the subsets (S[0]...S[n]) the subdomains can be abstracted;
           else
                - The framework is not an MDF;

           Concluded Analysis;
      end
```

Fig. 2. Algorithm to identify subdomains from applications developed with support of an MDF

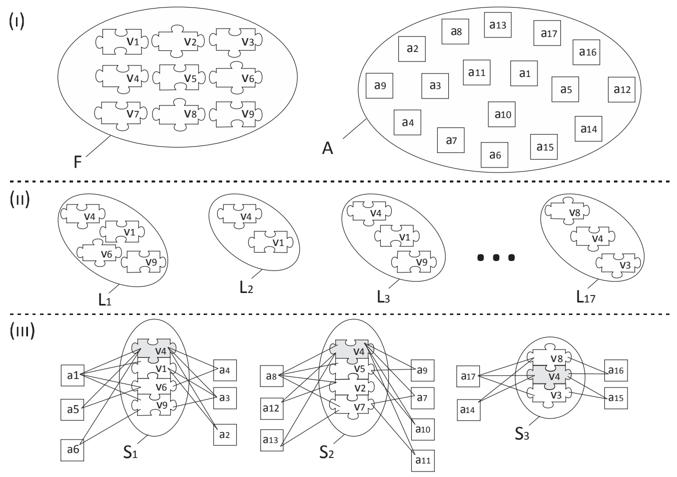

Fig. 3. Steps for subdomain identification using proposed strategy

If the grouping is possible, one may conclude the considered framework has characteristics of an MDF and from the subsets S[0]…S[n], the subdomains may be identified. Otherwise, it is not an MDF, thus all its variabilities may be used for all applications. In the part (III) of Figure 3 there are three subsets: S_1, S_2 and S_3. The application a4 only uses variabilities v4 and v6; in the future a4 may be evolved and also use variabilities v1 and v9, but rarely a4 will use variabilities from subsets S_2 and S_3. One ought to realize that there may be applications covering more than one of these subsets. However, the main idea is to separate the variabilities in subsets, based on the analysis of applications that use (or will use) only variabilities concentrated in these subsets. Notice that there may be common variabilities in the formed subsets, as the variability v4 that is used for all applications.

If we were to devise a feature model [19] to represent a possible FPL from the analysis of the framework "F", it could contain only four features: *Core*, *S1*, *S2* and *S3*. In this way, the variability v4 would be associated with the feature *Core* and the other variabilities with features that represent the subdomains. Based on available applications, the features *S1*, *S2* and *S3* could be organized, in the feature model, through the relationship of mutual exclusion, because there are not applications which use variabilities from more than one subdomain (other than the variability v4).

3.1 Mapping between subdomains and source code

In order to consolidate the subdomains identification we suggest that, in addition to the feature model to represent them, be created an artifact to map them to the source code. This artifact may assist the modularization process of the pieces of code for features that will form the resultant FPL. For this, the mapping must provide enough information in which from a certain subdomain we can know what are its variabilities and corresponding units that collaborate with its implementation. Besides that, some kind of description for pieces of code may be added to the mapping to indicate attributes, full methods or small refinements belonging to methods that deal with a different concern.

Table 1 partially shows how this mapping could be created in case of framework "F", which was discussed earlier. From this mapping we can identify the variabilities, implementation units and descriptions that indicate the pieces of code that implement each feature. The column named “Index” is used to identify the classes that contribute to the implementation of more than one variability, for instance, the classes ClassB and ClassX. In case of feature *S1*, its variabilities are v1, v6 and v9; the classes that implement these variabilities are ClassX, ClassB, ClassJ, ClassI and other classes that are not being shown; the methods methodX1(), methodsX2() and methodB2() belong to these classes and they cooperate with the implementation of the variabilities v1, v6 and v9. One should realize that there are classes that totally collaborate with the implementation of certain variabilities, such as the classes ClassJ and ClassI that implement v9.

Table 1. Mapping between subdomains and source code

Feature	Variability	Index	Implementation unit	Indicative for piece of code
Core	v4	1	ClassA (implements v4)	Full Class
		2	ClassB (implements v4)	attibuteB1, attributeB2 and metthodB1()
		3	ClassY (implements v4)	Full Class
S1	v1	**4**	ClassX (implements v1 and v6)	methodX1() and methodX2()
	v6	**2**	ClassB (implements v1, v6 and v9)	methodB2()
	v9	5	ClassJ (implements v9)	Full Class
		6	ClassI(implements v9)	Full Class
S2	v5	7	ClassH (implements v5 and v7)	Full Class
	v2	**4**	ClassX (implements v2)	attributeX1, attributeX2, methodX3(), methodX4() and methodX5()
	v7	**2**	ClassB (implements v5, v2 and v7)	attributeB3 and methodB3()

4 Case Study: GRENJ

GRENJ [9] was developed based on pattern language called GRN [5] and it covers Business Resource Management domain. Applications commonly developed with its support involve rental, trade and maintenance of business resources. A resource may be something like cars, CDs, hotel bookings, books, electronic equipment etc. In that way, we consider as subdomains: *Rental*, *Trade* and *Maintenance*. GRENJ contains 18.5 KLOC divided into three packages, 71 implementation units (classes/interfaces) and 1117 methods.

In order to make sure that the subdomains have been properly identified, we consider a set of applications that had evolved many times. Furthermore, based on the documentation we created an artifact to describe the available variabilities in the GRENJ. From this artifact and the applications, we applied the algorithm discussed in Section 3. Thus, the variabilities that were being used in applications were separated into four groups: *Core*, *Rental*, *Trade* and *Maintenance*. The variabilities associated with *Core* group can be applied to all subdomains, while the other variabilities are specific to them. Table 2 partially shows this separation of variabilities for GRENJ. It is important to highlight that this separation is a result of the use of algorithm already presented.

Table 2. Separation of variabilities into specific groups

Core	Rental	Trade	Maintenance
Resource	Fine Rate	Sale	Task
Transaction Itemized	Associated Sale	Purchase	Part
Destination Party	Reservation	Delivery	Quotation

After the analysis of developed applications with its support, documentation, pattern language, usage history and source-code, we decided to create a feature model [19] to represent GRENJ subdomains and for that we first figured out that the feature model would contain only three features, with the separation of two subdomains, as illustrated in part (a) of Figure 4. So, applications of *Rental* subdomain could evolve and use the variabilities of *Trade* and vice-versa. In this case, *Rental* and *Trade* could stay together, but separated from *Maintenance*. This separation was performed because applications of *Rental* subdomain will not use the variabilities of *Maintenance* subdomain. Nevertheless, the applications of *Trade* subdomain could evolve adding *Maintenance*, since the relationship is mutually exclusive, it would not be possible. Therefore, according to part (b) of Figure 4, the relationship among features became "or", but it does not address the case, because *Rental* and *Maintenance* can be together. In relation to it, we created a constraint [1] "Rental excludes Maintenance" and keeping the relationship "or", as presented in part (c) of Figure 4. This last feature diagram represents organization of GRENJ subdomains.

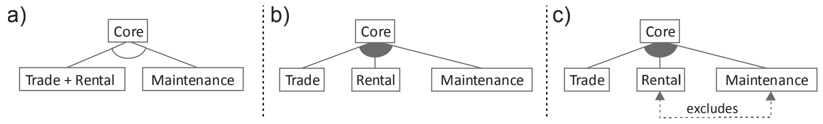

Fig. 4. Feature Model for subdomains of GRENJ

Consequently, applications belonging to *Trade* subdomain, for instance, might use a framework that contains only features of this subdomain and discard the remaining that belong to *Maintenance* subdomain, the same holds true for the opposite. Furthermore, the separation of *Trade* and *Rental* subdomains in order to make them also alternative may be justified by the need to develop applications that will use only variabilities of one of these subdomains. If the motivation is to obtain an FPL with fine-grained features, the feature diagram obtained in this activity that has only the subdomains separation may be refined. From the analysis of common and specific variabilities of subdomains and their usage history in the applications, we are able to decide which ones may be new features. Figure 5 shows the fine-grained features for an FPL from GRENJ. Due to space limitations we separate the feature model in the indicated parts: "A", "B", "C", "D" and "E". Note that there are more possibilities of choice that involve common features to the subdomains and the specific subdomain features. The two feature models depicted in the part (c) of Figure 4 and Figure 5 represent two possible versions of FPLs from GRENJ which we call GRENJ-FPL, considering the usage scenarios discussed earlier.

After creating a feature model to represent the FPL according to usage scenario 1 or 2, we may identify the units that collaborate with the implementation of the features. This activity can be supported by concern mining tools, as proposed by [18], in which it is possible to create a library containing keywords, imports and dependencies to investigate a specific concern. In GRENJ's case, we analyzed its source-code and documentation to identify the units which get together to cooperate with features belonging to the feature models. Based on this investigation we created a mapping for GRENJ following the idea discussed in Section 3.1. Table 3 partly shows the mapping of the subdomain (feature) "Maintenance", specifically the variability "Task" with implementation units and some descriptions that indicate the pieces of code that must be modularized in terms of features.

To modularize the identified units, several techniques can be employed, such as Aspect-Oriented Programming [20], Collaboration-based design [2], Model-Driven Development [10] etc. However, in order to provide a better separation of concerns and to obtain an FPL which is flexible in architectural terms, proper techniques ought to be used. To modularize the GRENJ we used Acceleo templates [16]. We have chosen Acceleo because it allows associating pieces of code with their respective features and generating source-code of the members from models.

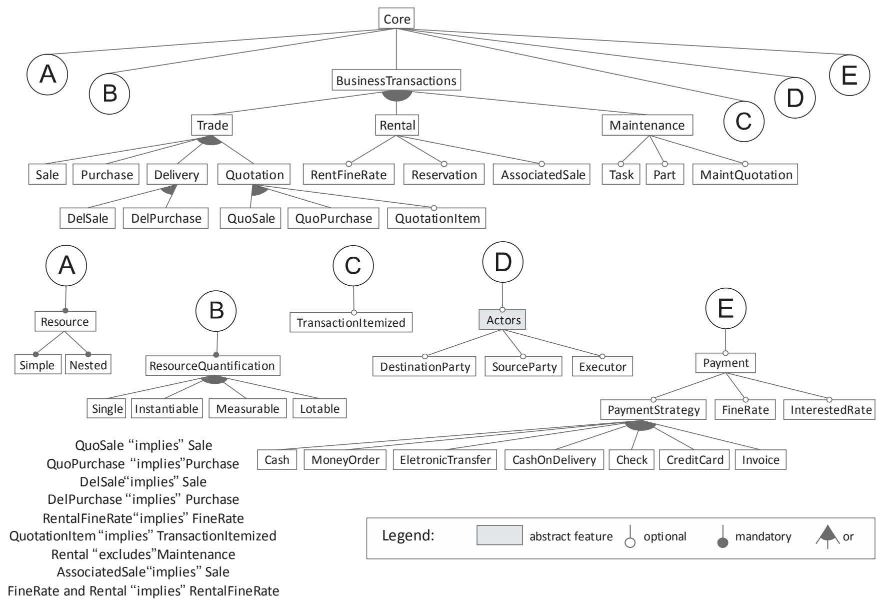

Fig. 5. Feature Model with fine-grained features for GRENJ

Table 3. Mapping between GRENJ subdomains and source code

Feature	Variability	Index	Implementation unit	Indicative for piece of code
Maintenance	Task	**3**	BasicMaintenance	BasicMaintenance(), setTasks(), getTasksAsString(), getTasks(), getTotalTasks(), getTotalTasksAsDouble(), setTotalTasks(),saveTasks(), getTotalTasksFromItemTransaction(), getTasksAsList() and addTask()
		32	MaintenanceTask	Full Class
		54	ResourceMaintenance	Full Class

5 Evaluation

GRENJ-FPL has two versions, one for each Usage Scenario: one with coarse-grained features (Usage Scenario 1) and another with fine-grained features (Usage Scenario 2), they are represented by feature models, in the part (c) of Figure 4 and Figure 5, respectively. In order to demonstrate the improvements in terms of use of these FPLs against previous MDF, we conducted a comparative study using source-code quantitative metrics. Thus, we developed three hypothetical applications with support of the FPLs and MDF. These applications have the following purposes: Rental of Vehicles (App.1), Sale of domestic equipment (App.2) and Maintenance of Motorcycles (App.3).

First of all, the applications were developed with GRENJ support, and then with a derived member from the FPL obtained concerning to Usage Scenario 1 (FPL1) and finally, with a derived member from the FPL obtained according with Usage Scenario 2 (FPL2).

The members from FPL1 are as follow: "*Core + Rental*" (C+R) for App.1, "*Core + Trade*" (C+T) for App.2 and "*Core + Maintenance*" (C+M) for App.3. The derived frameworks from the FPL2 aim to address the applications requirements. For App.1, the features were: *Core, Resource, Simple, Nested, ResourceQuantification, Instantiable, BusinessTransactions, Rental, TransactionItemized* and *DestinationParty*. For App.2: *Core, Resource, Simple, Nested, ResourceQuantification, Single, BusinessTransactions, Trade, TransactionItemized, DestinationParty, SourceParty, Executor, Payment, PaymentStrategy, Cash, MoneyOrder, EletronicTransfer, CashOnDelivery, Check, CreditCard* and *Invoice*. Finally, for App.3: *Core, Resource, Simple, Nested, ResourceQuantification, Single, BusinessTransactions, Maintenance, Task, Part, MaintQuotation, DestinationParty, Executor, Payment, PaymentStrategy, Cash, MoneyOrder, EletronicTransfer, CashOnDelivery, Check, CreditCard* and *Invoice*.

The comparative study was conducted with support of some metrics, as follows: Total number of lines of framework code (KLOC); Total number of abstract methods (hot spots); Quantity of implemented methods essential to make functional application; Total of lines of implemented methods; Quantity of parameters of abstract methods; Quantity of available features in the framework; Quantity of used features in the application; Total of variabilities/features that can be used in the future considering the application domain; and Quantity of variabilities/features that will rarely be used.

Table 4 shows the obtained data in the three versions of the frameworks for each application. The rows contain values for each metric and the columns contain values for each framework used in the applications. The total number of lines of code framework was calculated using the plug-in Eclipse Metrics and other metrics by direct observation of the code. In the rows related to metrics "Total number of lines of framework code (KLOC)" and "Total number of abstract methods (hot spots)", with the use of FPLs members, there is a reduction of quantity of code and available hot spots. For instance, considering GRENJ with 8.5 KLOC, FPL1 member used for developing App.1 had 4.2 KLOC and FPL2 member for the same application had 3.4 KLOC, and so on. This means a reduction of 50,58% and 60% of framework code for App.1.

In relation to the available hot spots in GRENJ, the reduction was 77,22% for FPL1 member and 89,88% for FPL2 member used for developing App.1. This also happens in the frameworks used by the other applications. In the development of the three applications with GRENJ support was needed to concretize methods not necessarily related to the applications requirements. For instance, 20 methods for App.1, 48 methods for App.2 and 55 methods for App.3 were concretized, as described in the row related to metric "Quantity of implemented methods essential to make functional application". From FPL2 members, it was possible to concretize only methods related to applications requirements. It represents a reduction of 20% of methods in App.1, 41,66% for App.2 and 47,27% for App.3. The obtained data with metrics "Total of lines of implemented methods" and "Quantity of parameters of abstract methods" indicates that the number of lines of implemented methods and their parameters continued steady in the use of MDF and FPL1 members. However, when the applications were developed with FPL2 members, there was a significant reduction in number of code lines. For App.1, the reduction was from 52 to 40 lines (it represents 23%), for App.2 was 187 to 132 (29,41%) and for App.3 was 89 to 63 lines (29,21%).

In order to compute the four last metrics, we admitted an equivalence level. As total quantity of features, we adopted those provided by FPL2, i.e., 43 features. So, for FPL1 members, we considered the including features in: *Core*, *Rental*, *Trade* and *Maintenance*. For MDF, we considered it providing 43 variabilities. For example, the derived member of FPL1 used by App.2 provides 35 features included in only two: *Core* and *Trade*. There are 24 features in *Core* and 9 in *Trade*, resulting in 35 features instead of 33, because *Core* and *Trade* are not abstract features. For App.2, the values related to the metric "Quantity of available features in the framework" contain C=25 and T=10, whereas the derived member of FPL2 for App.2 provides only 21 features based on the requirements of this application. This analysis is also performed for App.1 and App.3.

Table 4. Collected data in the evaluation

Metric	Rental of Vehicles - App.1			Sale of domestic equipment - App. 2			Maintenance of Motorcycles - App. 3		
	MDF (GRENJ)	FPL1 member	FPL2 member	MDF (GRENJ)	FPL1 member	FPL2 member	MDF (GRENJ)	FPL1 member	FPL2 member
Total number of lines of framework code (KLOC)	8.5	4.2	3.4	8.5	5.6	4.1	8.5	3.5	2.9
Total number of abstract methods (hot spots)	158	36	16	158	56	28	158	62	29
Quantity of implemented methods essential to make functional application	20	20	16	48	48	28	55	55	29
Total of lines of implemented methods	52	52	40	187	187	132	89	89	63
Quantity of parameters of abstract methods	12	12	8	19	19	16	18	18	14
Quantity of available features in the framework	43	C+R C=25 R=4 29	10	43	C+T C=25 T=10 35	21	43	C+M C=25 M=4 29	22
Quantity of used features in the application	43/10	29/10	10/10	43/21	35/21	21/21	43/22	29/22	22/22
Total of variabilities/features that can be used in the future considering the application domain	18	C=16 R=2 18	0	14	C=5 T=9 14	0	7	C=7 M=0 7	0
Quantity of variabilities/features that will rarely be used	15	0	0	8	0	0	14	0	0

In order to obtain the values for metric “Quantity of used features in the application”, it was necessary to take into account the obtained values in the previous metric. To illustrate, although MDF provides 43 variabilities, App.1 required only 10 (43/10). The FPL1 member provided 29 (29/10) and FPL2 member only 10 (10/10) features and so on.

The values for metric “Total of variabilities/features that can be used in the future considering the application domain” show the remaining features which are likely to be used in the future. As to emphasize, for computing we consider that evolution of the three applications do not imply an addiction of another subdomain. For instance, an application of *Trade* subdomain evolves aggregating *Rental*. Therefore, by using GRENJ for developing App.1, 18 features that were not used, may be used in the future. For FPL1 member used in this application, 18 features can be decomposed as follows: 16 belong to *Core* and only 2 to *Rental* (C=16, R=2). Regarding the FPL2 members, there are not remaining features for three applications.

The last line of Table 4 contains values related to metric “Quantity of variabilities/features that will rarely be used”, i.e., the quantity of features provided by the framework that rarely will be used in the applications, even if they evolve. In case of use of MDF to develop App.1, 15 of 43 features rarely will be used and even so they continue in the application. It represents 34,88%

of unnecessary features. The quantity of unnecessary features for App.2 was 18,6% and App.3 was 32,55%. Note that those unnecessary features do not exist in FPL1 and FPL2 members.

In general, the instantiation of a smaller framework and targeted to the domain of an application, may be more productive and lower error-prone, when compared with the use of the complete version of the same framework. The higher the quantity of available variabilities, mainly if they have not been implemented with proper management, there are more chances for AE to make mistakes, for instance, to select erroneously variabilities and to concretize methods incorrectly.

6 RELATED WORK

The approach proposed by [12] aims at modularizing crosscutting concerns found in frameworks. This research is based on the concept of Horizontal Decomposition (HD) and the HD principles were evaluated based on the results of a restructured persistence framework. The main concern of this study was to improve the framework modularity, in order to reduce the effort when performing maintenance. Most of the literature that deal with framework modularization make use of the existing crosscutting concerns as main criterion. That means the objective is to have a new framework version where the crosscutting concerns are well modularized. Our goal, however, differs in the fact that for us what guide the modularization are the subdomains. So, we could use, for example, HD as our strategy. Having this pointed out, this work is complementary to ours.

Another approach was proposed by [21]. The researchers present a methodology for restructuring of frameworks in cascade. They consider that a framework can be specified by a set of models and, through these, a set of modularizations may be sequentially applied. The modularization starts in the feature model, then in the use case model and, after that, in the architectural model up to the time it achieves the source code. In order to preserve the framework's behavior after each change, trace maps are used among the models. As a result of this process, decision records regarding the transformations are analyzed to document and to completely restructure the framework, with improvements in terms of modularity, which reflect more effective levels of maintenance. Considering this methodology, the trace maps can assist the organized modularization of MDFs in FPLs, as it enables the traceability among the features, including composition rules and their relationships with the framework code. This work presents just a strategy, but does not concern about a modularization criteria. So, it is possible to use their modularization strategy to restructure MDFs into FPLs.

7 CONCLUSIONS

This paper defines the FPL concept, a strategy to identify subdomains of an MDF as an essential step to modularize it into FPL and also shows a comparative study demonstrating the benefits of developing applications with support of FPL members, instead of using the complex structure of an MDF with all available variabilities covering several subdomains.

An FPL provides a flexible architecture that enables the creation of members that have a subset of the features. These members are frameworks directed to the domain of these applications, so they need to be instantiated in order to obtain concrete applications. When modularizing MDFs in FPLs, one can achieve greater flexibility in the composition of features, which provide smaller frameworks and also better productivity levels by reducing error-prone in the instantiation process of these frameworks. The flexibility of composing features of an FPL enables to address the specific demands of applications. It is worth mentioning that these frameworks can be formed by a subset of features that must be used now or in the future by a certain set of applications that have a domain in common, i.e. one can avoid unnecessary variabilities that support different domains of these applications.

Considering the concepts presented, one of the future perspectives of the work is to explore the possibility of developing an SPL from an FPL that was obtained from an application framework, including the classes that instantiate it and, then, compose and test the applications derived from this line. Moreover, it is possible to explore the fact of an FPL to be maintained and used under the concepts of Software Ecosystems (SECOS) [13], since an FPL may consist of several developers on a distributed and open-source platform, that is, a collaborative network. Therefore, the creation of FPLs that are flexible and targeted to the business areas is also possible. An FPL may become available so that other developers can add features or improve the existing ones. Also, many application frameworks restricted as proprietary software, and consequently hard to find, can be provided and improved with the use of SECOS.

A limitation of the FPL concept is that there is not a comprehensive tool that supports the process of framework modularization. With a complete tool, it would also be possible to investigate the impact of new features in the architecture of an FPL, by analyzing the interferences they can cause in the existing ones. It is also believed that the creation of a plugin to visualize the mapping between features and classes can help FEs modularize frameworks, by showing which classes implement a particular feature and which are affected in case a feature is selected. Apart from this plugin, another tool to enable the creation of FPL members at various levels can be developed. At first, the FPL Engineer would create members with the features of a given domain and, then, if necessary, they would select a more specific set of features from this subdomain.

Acknowledgements

The authors would like to thank Capes, CNPq (Process 560241/2010-0) and Fapesp (Process 2011/04064-8) for financial support.

References

1. Barreiros, J., Moreira, A.: Soft Constraints in Feature Models. In: 6th International Conference on Software Engineering Advances (ICSEA), pp. 136-141, Barcelona (2011)
2. Batory, D., Cardone, R., Smaragdakis, Y.: Object Oriented Frameworks and Product Lines. In: 1st Software Product Lines Conference (SPLC1), pp. 227-247, Colorado (2000)
3. Batory, D., Lopez-Herrejon R. E., Martin J.P.: Generating Product-Lines of Product-Families. In: 17th IEEE International Conference on Automated Software Engineering, IEEE Press, pp. 81–92, Edinburgh, UK (2002)
4. Batory, D. Shepherd, C. T.: Product Lines of Product Lines. Technical report, University of Texas-Department of Computer Science (2011)
5. Braga, R. T. V., Germano, F. S. R., Masiero, P. C.: A Pattern Language for Business Resource Management. In: 6th Pattern Language of Programs Conference, pp. 1–33, Monticello, Illinois (1999)
6. Camargo, V. V., Masiero P. C.: An approach to design crosscutting framework families. In: Proceedings of the 2008 AOSD workshop on Aspects, components, and patterns for infrastructure software, pp. 1-6, Brussels, Belgium (2008)

7. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley Professional, 3rd edition (2001)
8. Czarnecki, K., S. Helsen, U. Eisenecker.: Staged configuration through specialization and multilevel configuration of feature models. In: Software Process: Improvement and Practice, pp. 143 – 169 (2005)
9. Durelli, V. H. S., Durelli, R. S., Braga, R. T. V., Borges, S. S.: A Domain Specific Language for Lessening the Effort Needed to Instantiate Applications Using GRENJ Framework. In: Information Systems Brazilian Symposium, pp. 31-40, Pará, Brazil (2010)
10. Gottardi, T., Durelli, R., López, O., Camargo, V. V.: Model-based reuse for crosscutting frameworks: assessing reuse and maintenance effort. In: Journal of Software Engineering Research and Development, v. 1, pp. 4-34 (2013)
11. Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: Design patterns: Elements of reusable object-oriented software. Addison Wesley (1995)
12. Godil, I., Jacobsen, H.: Horizontal decomposition of Prevayler. In: Conference of the Centre for Advanced Studies on Collaborative Research, pp. 83-100. Richmond Hill, Canada (2005)
13. Jansen, S., Cusumano, M.: Defining Software Ecosystems: A Survey of Software Platforms and Business Network Governance. In: 4th Workshop on Software Ecosystem, pp. 41-58, Boston, MA, USA (2012)
14. JBoss Community – Hibernate, http://www.hibernate.org
15. Johnson, R. E.: Reusing Object-Oriented Design. Technical Report - University of Illinois, (1991)
16. Obeo - Model Driven Company, Inc. – Acceleo http://www.eclipse.org/acceleo
17. Oliveira, A.L., Ferrari, F.C., Penteado, R.A.D., Camargo, V. V.: Investigating Framework Product Lines. In: ACM Symposium on Applied Computing, 27th ACM Symposium on Applied Computing, pp. 1177-1182, Trento, Italy (2012)
18. Santibáñez, D. S. M., Durelli, R. S.. Marinho B.. Camargo, V.V.: CCKDM - A Concern Mining Tool for Assisting in the Architecture-Driven Modernization Process. In: CBSoft - The Tools Session, Brasilia, Brazil (2013)
19. Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E, Peterson, A. S.: Feature Oriented Domain Analysis (FODA) Feasibility Study. Technical Report, (1990)
20. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., Irving, J.: Aspect Oriented Programming. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP). Springer-Verlag, Finland (1997)
21. Xu L., Butler G.: Cascaded Refactoring for Framework Development and Evolution. In: Proceedings of the Australian Software Engineering Conference (ASWEC'06), pp. 319-330. Sydney, Australia (2006)