Extracting Software Product Lines: A Case Study Using Conditional Compilation

Marcus Vinicius Couto
Institute of Informatics, PUC Minas
Belo Horizonte, Brazil
marcusvnac@gmail.com

Marco Tulio Valente, Eduardo Figueiredo
Department of Computer Science, UFMG
Belo Horizonte, Brazil
{*mtov,figueiredo*}*@dcc.ufmg.br*

***Abstract*—Software Product Line (SPL) is a development paradigm that targets the creation of variable software systems. Despite the increasing interest in product lines, research in the area usually relies on small systems implemented in the laboratories of the authors involved in the investigative work. This characteristic hampers broader conclusions about industry-strength product lines. Therefore, in order to address the unavailability of public and realistic product lines, this paper describes an experiment involving the extraction of a SPL for ArgoUML, an open source tool widely used for designing systems in UML. Using conditional compilation we have extracted eight complex and relevant features from ArgoUML, resulting in a product line called ArgoUML-SPL. By making the extracted SPL publicly available, we hope it can be used to evaluate the various flavors of techniques, tools, and languages that have been proposed to implement product lines. Moreover, we have characterized the implementation of the features considered in our experiment relying on a set of product-line specific metrics. Using the results of this characterization, it was possible to shed light on the major challenges involved in extracting features from real-world systems.**

***Keywords*-software product lines; conditional compilation; refactoring.**

I. Introduction

Software Product Line (SPL) is an emerging paradigm to create variable software systems [1], [2]. The ultimate goal is to move from a one-at-a-time software implementation culture to a new scenario where systems are systematically derived from a managed set of software assets. Typically, SPLs are composed by core components – shared by all products – and components responsible for implementing features that are required only in particular domains or market segments.

The basic principles of software product lines have been proposed in the last decade (or even before, since in essence they just reinforce well-known software reuse principles [3]). However, specially from the source code viewpoint, we still miss the application of product-line principles in medium to large software systems. Particularly, research in SPL usually relies on small systems implemented in laboratories by the research group involved in the target investigative work. As an example, we can mention SPLs such as Expression Product Line [4], Graph Product Line [5], and Mobile Media Product Line [6]. Clearly, SPLs synthesized in research labs are useful to promote the basic principles of this software development approach. In addition, they facilitate the investigation of novel techniques, tools, and languages supporting a product-line based view of software development. On the other hand, it is well-known that there are fundamental differences between engineering small systems (programming-in-the-small) and engineering complex systems, composed by hundreds of modules (programming-in-the-large) [7], [8]. Therefore, since SPL targets reuse-in-the-large, it is essential to investigate whether recent research results in this area can be extrapolated to real and complex software systems.

In order to address the unavailability of public and realistic product lines, this paper describes an experiment involving the extraction of a SPL from ArgoUML, a Java-based open source tool widely used for designing systems in UML. Therefore, instead of generating a SPL from a small system fully conceived and implemented in a research lab, we decided to extract features from a real-world, mature, and complex software system. In our experiment, the code responsible for the implementation of eight features from ArgoUML has been annotated using conditional compilation directives, resulting in a product line called ArgoUML-SPL.

The following optional features are part of the extracted ArgoUML-SPL: Activity Diagram, State Diagram, Collaboration Diagram, Sequence Diagram, Use Case Diagram, Deployment Diagram, Logging, and Cognitive Support. Essentially, such features have been selected because they represent relevant functional requirements (as in the case of the extracted UML diagrams) or classical non-functional requirements (as in the case of Logging). Also, some of the selected features, such as Logging and Cognitive Support, have a crosscutting behavior.

ArgoUML's original implementation has about 120 KLOC. From those lines, we have annotated about 37 KLOC as responsible for the implementation of at least one of the aforementioned features. To the best of our knowledge, such numbers make ArgoUML-SPL more complex and larger than any public product line previously reported in the literature. Finally, we present a detailed quantitative evaluation and characterization of the extracted SPL. For example, we provide information about size, crosscutting behavior, granularity, and static location of the features extracted in our study.

To summarize, our contributions are twofold. First, using conditional compilation we have extracted a SPL including eight complex and relevant features from a mature UML modeling tool. By making the extracted SPL

1534-5351/11 $26.00 © 2011 IEEE
DOI 10.1109/CSMR.2011.25

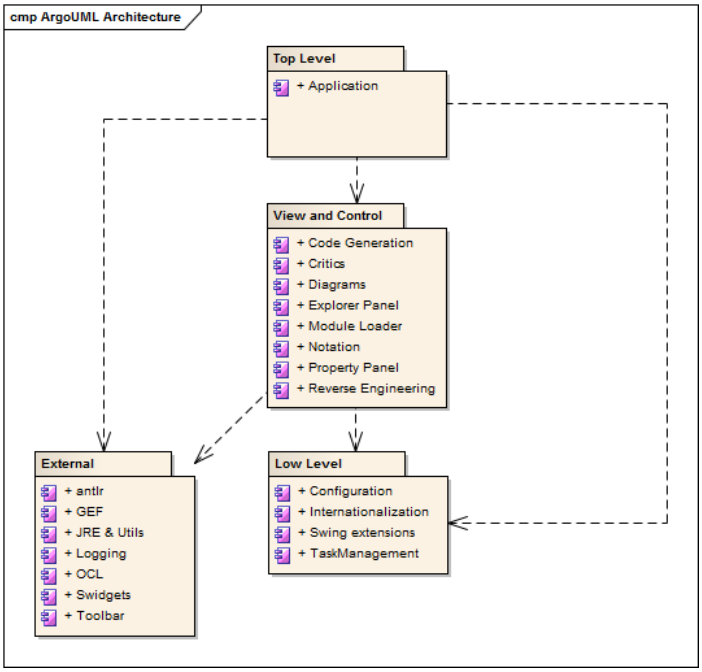

Figure 1. ArgoUML Architecture

publicly available, we hope it can be used to evaluate the various flavors of techniques, tools, and languages that have been proposed to implement product lines. Second, we have characterized the implementation of the features considered in our experiment. Using the results of this characterization, it was possible to shed light on the major challenges involved in extracting features from real-world systems.

The remainder of this paper is organized as follows. The next section shows an overview about the proposed product line. In this section, we show a basic architecture of this system, the implemented feature model and the extraction process that guided the creation of the proposed software product line. Section III presents the results of a quantitative analysis of the extracted features based in a set of metrics. Section IV discusses how the extracted features can be classified according to some patterns usually found in the literature. This section also shows some challenges and problems that could be faced in the possible separation of features with aspect technology. Finally, Section V discusses related work and Section VI concludes the paper and points out directions for future work.

II. ArgoUML-SPL

This section provides an overview about the ArgoUML architecture. It also describes the feature model proposed for ArgoUML-SPL. Finally, we report our effort on manually inserting pre-processor directives in the source code, in order to delimit the ArgoUML-SPL features.

A. Architecture

Figure 1 illustrates the ArgoUML architecture, highlighting the following subsystems [9]:

- External: third-party libraries such as the Graph Editing Framework (GEF) and libraries for handling OCL constraints.
- Low-level: subsystems that are responsible for low-level tasks such as configuration, internationalization, task management etc.
- View and Control: these subsystems are responsible for tasks like diagram management, GUI, code generation, and reverse engineering.
- Top-Level: subsystem that initializes the other subsystems.

B. Feature Model

Figure 2 presents the feature model of the extracted SPL. For creating ArgoUML-SPL, eight features representing functional and nonfunctional requirements have been selected. The first feature – Logging – has been selected as a representative of a non-functional requirement. The other seven features represent functional concerns: Activity Diagram, State Diagram, Collaboration Diagram, Sequence Diagram, Use Case Diagram, Deployment Diagram, and Cognitive Support.

The Cognitive Support feature provides information that helps designers to detect and to solve problems in their models [10]. This feature is implemented by software agents that run continuously in a background thread of control. Such agents analyze the diagrams created by end-users and indicate potential problems. For example, they can recommend well-known design practices, remind about parts of the project that have not been finalized or warn about syntax errors. Finally, Activity Diagram, State Diagram,

COLLABORATION DIAGRAM, SEQUENCE DIAGRAM, USE CASE DIAGRAM, and DEPLOYMENT DIAGRAM provide support to the respective UML diagrams.

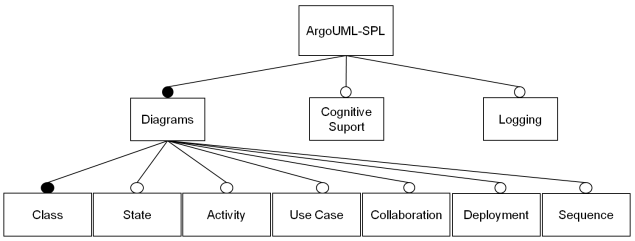

Figure 2. ArgoUML-SPL Feature Model

Rationale: Two main criteria have guided the selection of the mentioned features: relevance and implementation complexity. Regarding relevance, we have decided to extract only features that represent typical functional requirements in the domain of UML modeling tools (as the six UML diagrams), a feature that represent a classical non-functional requirement (LOGGING) and a feature presenting an unquestionable optional behavior (COGNITIVE SUPPORT).

With respect to the complexity criterion, the implementation of each selected feature requires a considerable amount of code not confined in a single class or package. For example, the implementation of LOGGING and COGNITIVE SUPPORT has a crosscutting behavior, impacting several classes of the system. Finally, two situations are rather common in the implementation of the features representing diagrams: tangling (when a block of code, usually a single statement or expression, includes code associated with more than one feature) and nesting (when a block of code associated with a given feature is lexically nested in an external block of code, associated with another feature).

We should mention that ArgoUML-SPL includes only optional and mandatory features (i.e. it does not provide support to alternative features, for example). Furthermore, there are no semantic dependencies between the extracted features. In other words, any possible configuration – with any feature enabled or disabled – is valid. However, these limitations do not represent a threat on the validity of using ArgoUML-SPL to assess SPL implementation techniques. The reason is that eventual dependencies between two features `A` and `B` – including `or`, `xor`, or `requires` dependencies – do not impact in the annotation of the `A` and `B` code (i.e. the code of both features must be annotated anyway). In fact, conditional compilation based approaches consider dependencies between `A` and `B` only at product derivation time in order to prevent the generation of invalid products.

C. Extraction Process

In the extracted SPL, preprocessor directives have been used to delimit the optional code associated to each feature. It is well known that preprocessors pollute the code with annotations, making the program less readable and harder to understand, maintain and evolve [11]–[13]. However, preprocessors have some advantages too, including expressiveness, since they support the annotation of any piece of code. In other words, preprocessors constitute the most primitive and at the same type the most powerful technology for annotating feature code. Therefore, by using preprocessors in the ArgoUML-SPL extraction, our main intention is to provide a product line that can be used to evaluate other modularization and separation of concern technologies, such as aspect-oriented programming [14], feature-oriented programming [15], and disciplined annotations [13].

Since Java does not provide native support to preprocessor directives, we relied on a third-party preprocessor tool – called `javapp`[1] – to annotate the feature code. This tool supports preprocessor directives similar to the ones that exist in C/C++, including `#ifdef`, `#ifndef`, and `#else`. Basically, these directives indicate to the preprocessor whether the code fragment they delimit should be passed to the compiler or not. In this way, it is possible to generate particular products from the SPL without the optional features indicated by SPL developers.

Figure 3 presents an example of code delimited by preprocessor directives. In this figure, the statement in line 3 has been annotated as associated to COGNITIVE SUPPORT. Therefore, the assignment will only be included in products with this feature enabled. On the other hand, the assignment in line 5 will only be included in products with COGNITIVE SUPPORT disabled.

```
1 JPanel todoPanel;
2 //#if COGNITIVE
3 todoPanel = new ToDoPane(splash);
4 //#else
5 todoPanel = new JPanel();
6 //#endif
```

Figure 3. Feature code delimited by conditional compilation directives

To automate the generation of the products that can be derived from ArgoUML-SPL, we have implemented a set of `ant`[2] scripts. Basically, these scripts allow developers to inform the features that should be activated in a particular product. After that, the scripts call the `javapp` preprocessor and the Java compiler, in order to generate the requested product.

III. CHARACTERIZATION OF THE EXTRACTED SPL

In order to reason about the extracted SPL, we have used – and adapted – a recent metrics suite proposed to evaluate SPL implementations based on preprocessors [16]. Basically, four sets of metrics have been collected: size metrics, crosscutting metrics, granularity metrics, and localization metrics. These metrics - and their related values - are detailed in the next subsections.

[1]Available at http://www.slashdev.ca/javapp.
[2]Available at http://ant.apache.org.

A. Size Metrics

These metrics are designed to evaluate the size of the products and features of a SPL implemented using conditional compilation. More specifically, we collected the following size metrics:

- Lines of Code (LOC): counts the number of lines of code – without comments and blank lines – for a given product P generated by the SPL;
- Number of Packages (NOP): counts the number of packages present in a given product P;
- Number of Classes (NOC): counts the number of classes present in a given product P;
- Lines of Feature Code (LOF): proposed by Liebig et al. [16], this metric counts the number of lines of code – without comments and blank lines – responsible for the implementation of a given feature F. Essentially, LOF(F) = LOC(All) − LOC($All - F$), where All denotes the product with all the features from the SPL enabled; $All - F$ denotes the product with all the features enabled, except F.

Tables I and II present the values for the size metrics for products and features, respectively. Table II also presents the percentage of the code dedicated to each feature (regarding the original, non-SPL based version of the system).

Table I
SIZE METRICS FOR PRODUCTS

Product	LOC	NOP	NOC
Original, non-SPL based	120,348	81	1,666
Only COGNITIVE SUPPORT disabled	104,029	73	1,451
Only ACTIVITY DIAGRAM disabled	118,066	79	1,648
Only STATE DIAGRAM disabled	116,431	81	1,631
Only COLLAB. DIAGRAM disabled	118,769	79	1,647
Only SEQUENCE DIAGRAM disabled	114,969	77	1,608
Only USE CASE DIAGRAM disabled	117,636	78	1,625
Only DEPLOY. DIAGRAM disabled	117,201	79	1,633
Only LOGGING disabled	118,189	81	1,666
All the features disabled	82,924	55	1,243

Table II
SIZE METRIC FOR FEATURES

Feature	LOF	
COGNITIVE SUPPORT	16,319	13.56%
ACTIVITY DIAGRAM	2,282	1.90%
STATE DIAGRAM	3,917	3.25%
COLLABORATION DIAGRAM	1,579	1.31%
SEQUENCE DIAGRAM	5,379	4.47%
USE CASE DIAGRAM	2,712	2.25%
DEPLOYMENT DIAGRAM	3,147	2.61%
LOGGING	2,159	1.79%
Total	37,424	31.10%

Analysis of Size: As shown in Table I, the original version of the system – with all the features enabled – has 120,348 LOC. On the other hand, the minimal product that can be derived from the extracted SPL – with all the features disabled – has 82,924 LOC (therefore, 31% smaller). This reduction shows the benefits in terms of customization that can be achieved when moving to a SPL-based approach to software development. That is, instead of a one-size-fits-all approach, the extracted product line allows users to tailor the system size to their real needs. Table I also shows that LOGGING is the only feature whose removal does not affect the number of classes and packages of the system. This property is due to the fact that ArgoUML relies on an external library called Log4J[3] for LOGGING.

As shown in Table II, COGNITIVE SUPPORT is the feature with the highest LOF (13.56% of the lines of the system are dedicated to implement this feature). The simplest diagram – at least in terms of size – is COLLABORATION DIAGRAM (1,579 LOC). On the other hand, SEQUENCE DIAGRAM is the diagram with the highest LOF (5,379 LOC). Table II also shows that the number of lines of code dedicated to LOGGING (2,159 LOC) is close to the lines dedicated to diagrams like ACTIVITY DIAGRAM (2,282 LOC) and USE CASE DIAGRAM (2,712 LOC). Such numbers suggest the importance of considering LOGGING as a feature in ArgoUML-SPL.

The features extracted in our experiment represent a total of 37,424 LOC (31.1% of the original version of ArgoUML). In other words, the core of the extracted SPL corresponds to around 69% of the system size, in terms of lines of code. Basically, the classes in the core are responsible for user interface and for rendering diagrams in the screen (around 38% of the system size) and for several other features, such as persistence, internationalization, code generation, reverse engineering etc.

B. Crosscutting Metrics

These metrics are designed to measure the crosscutting behavior of the features extracted in preprocessor based SPLs. More specifically, we collected the following metrics:

- Scattering Degree (SD): proposed by Liebig et al. [16], this metric counts the number of occurrences of the `#ifdef` constants that define a given feature F. That is, given a constant that defines a particular feature, SD counts the number of occurrences of this constant in `#ifdef` expressions. To illustrate, suppose the code fragment presented on Figure 4. In this example, the constant `STATEDIAGRAM` designates the code associated to the STATE DIAGRAM feature. Therefore, SD(STATE DIAGRAM)= 4, since the mentioned constant appears in four `#ifdef` expressions (lines 1, 4, 8, and 22).

- Tangling Degree (TD): for each pair of features F_1 and F_2, our original definition for this metric counts the number of `#ifdef` expressions where F_1 and

[3]Available at http://logging.apache.org/log4j/

F_2 are combined by AND or OR operators. In the example of Figure 4, TD(ACTIVITY DIAGRAM, STATE DIAGRAM) = 3 because there is an `#ifdef` in which the two features are combined by an AND operator (line 8) and two `#ifdefs` in which the features are combined by an OR operator (lines 1 and 22). The goal of this metric is to measure interactions and dependencies between the features extracted in a preprocessor-based SPL.

```
1  //#if STATEDIAGRAM or ACTIVITYDIAGRAM
2  if ((
3
4    //#if STATEDIAGRAM
5    type == DiagramType.State
6    //#endif
7
8    //#if STATEDIAGRAM and ACTIVITYDIAGRAM
9    ||
10   //#endif
11
12   //#if ACTIVITYDIAGRAM
13   type == DiagramType.Activity
14   //#endif
15
16   ) && machine == null) {
17   diagram = createDiagram(...);
18 } else {
19
20 //#endif
21 diagram = createDiagram(...);
22 //#if STATEDIAGRAM or ACTIVITYDIAGRAM
23 }
24 //#endif
```

Figure 4. Example of feature scattering and tangling

Tables III and IV show the values collected for the SD and TD metrics, respectively.

Table III
SCATTERING DEGREE (SD)

Feature	SD	LOF/SD
COGNITIVE SUPPORT	319	51.16
ACTIVITY DIAGRAM	136	16.78
STATE DIAGRAM	167	23.46
COLLABORATION DIAGRAM	89	17.74
SEQUENCE DIAGRAM	109	49.35
USE CASE DIAGRAM	74	36.65
DEPLOYMENT DIAGRAM	64	49.17
LOGGING	1287	1.68

Table IV
TANGLING DEGREE (TD)

Pairs of Features	TD
(STATE DIAGRAM, ACTIVITY DIAGRAM)	66
(SEQUENCE DIAGRAM, COLLABORATION DIAGRAM)	25
(COGNITIVE SUPPORT, SEQUENCE DIAGRAM)	1
(COGNITIVE SUPPORT, DEPLOYMENT DIAGRAM)	13

Crosscutting Behavior Analysis: Table III shows that LOGGING presents the highest SD value. According to the data presented on this table, there are 1,287 static locations in the system where LOGGING is required. Therefore, this value confirms the recurrent claims in the literature about the crosscutting behavior of LOGGING. However, as presented in Table II, LOGGING has the second lowest value for LOF among the features considered in this study. That is, LOGGING is used in 1,287 parts of the system, but it requires few lines of code in each of these points (on overage, 1.68 lines of code on each location where it is demanded). On the other hand, there are features like DEPLOYMENT DIAGRAM that are required in only 64 locations of the system, but that demand much more code at each location (on average, 49.17 lines of code on each location).

Regarding the TD metric, there are tangling relations between four pairs of features, as presented in Table IV (for the combination of features not presented in this table, TD=0). Tangling relations have been detected between similar UML diagrams, particularly between STATE DIAGRAM and ACTIVITY DIAGRAM and between SEQUENCE DIAGRAM and COLLABORATION DIAGRAM. Since such diagrams are similar, tangling denotes code fragments that must be included when both diagrams are enabled (AND operator) or when at least one the diagrams is enabled (OR operator). There is also tangling relations between COGNITIVE SUPPORT and two diagrams: SEQUENCE DIAGRAM and DEPLOYMENT DIAGRAM. In this case, tangling is used to denote classes that provide cognitive support specifically to the mentioned diagrams.

Among the features annotated in the study, LOGGING is the one that appears most often lexically nested in blocks of code associated to other features. Indeed, we have found that LOGGING appears nested to all the other features considered in the study. This property of LOGGING can be explained due to its crosscutting behavior (as measured by the SD metric). In other words, LOGGING is a non-functional requirement whose implementation crosscuts most of the functional requirements of a system. Finally, we found eleven static locations where ACTIVITY DIAGRAM is lexically nested in STATE DIAGRAM. By inspecting the code, we have observed that this nesting is due to the fact that ACTIVITY DIAGRAM is a specialization of STATE DIAGRAMS, as defined by the UML specification [17].

C. Granularity Metrics

Granularity metrics quantify the hierarchical level of the program elements annotated for a specific feature. These metrics are able to identify granularity levels ranging from coarse to fine-grained features. By our definition, a feature has coarse granularity when its implementation occurs mainly in syntactic units with a higher hierarchical level according to the Java grammar, such as packages, classes and interfaces. On the other hand, a feature is fine-grained when its code is composed by lower level syntactic units, such as statements and expressions [18]. The metrics used in our study to measure granularity are:

Table V
COARSE-GRAINED GRANULARITY METRICS

Feature	Package	Class	InterfaceMethod	Method	MethodBody
COGNITIVE SUPPORT	11	9	1	10	5
ACTIVITY DIAGRAM	2	31	0	6	6
STATE DIAGRAM	0	48	0	15	2
COLLABORATION DIAGRAM	2	8	0	5	3
SEQUENCE DIAGRAM	4	5	0	1	3
USE CASE DIAGRAM	3	1	0	1	0
DEPLOYMENT DIAGRAM	2	14	0	0	0
LOGGING	0	0	0	3	15

Table VI
FINE-GRAINED GRANULARITY METRICS

Feature	ClassSignature	Statement	Attribute	Expression
COGNITIVE SUPPORT	2	49	3	2
ACTIVITY DIAGRAM	0	59	2	6
STATE DIAGRAM	0	22	2	5
COLLABORATION DIAGRAM	0	40	1	1
SEQUENCE DIAGRAM	0	31	2	3
USE CASE DIAGRAM	0	22	1	0
DEPLOYMENT DIAGRAM	2	13	1	3
LOGGING	0	789	241	1

- `Package`: quantifies packages entirely annotated (all classes and interfaces) as implementing a feature.
- `Class`: quantifies classes or interfaces entirely annotated as implementing a feature.
- `ClassSignature`: counts pieces of annotated code associated to class signatures, i.e. `extends` and `implements` clauses.
- `InterfaceMethod`: counts method signatures in a Java interface annotated to implement a feature.
- `Method`: counts methods entirely annotated to implement a feature (i.e. signature and body).
- `MethodBody`: counts methods whose body (but not the signature) has been annotated to implement a feature.
- `Attribute`: counts the number of class and instance attributes annotated to implement a particular feature.
- `Statement`: counts the number of statements annotated to implement a feature, including method calls, assignments, conditional statements, loop statements etc.
- `Expression`: counts the number of expressions annotated to implement a particular feature (e.g. the expression of an `if` or loop statement).

The granularity metrics do not take into account elements that have already been counted in another metric. For instance, the `Class` metric does not consider a class or interface belonging to a fully annotated package (i.e. a package counted by the `Package` metric). We make this decision to avoid double counting a specific element.

In this paper, we assume the following categories for coarse and fine-grained syntactic elements:

- Coarse-grained: `Package`, `Class`, `MethodBody`, `Method` and `InterfaceMethod`.
- Fine-grained: `ClassSignature`, `Attribute`, `Statement`, and `Expression`.

Granularity Analysis: Tables V and VI show the values for the proposed coarse and fine-grained granularity metrics, respectively. Two main findings can be taken from them:

- The implementation of COGNITIVE SUPPORT, ACTIVITY DIAGRAM and STATE DIAGRAM has more coarse-grained elements than the other features. Basically, part of the code of such features is properly modularized by elements such as packages, classes, and interfaces. However, such coarse-grained components are referenced by other elements of smaller granularity. In summary, the implementation of the aforementioned features is well modularized, but its use is spread in the rest of the code.

- LOGGING is a fine-grained feature that demands 789 statements and 241 attributes. However, the implementation of LOGGING is not modularized at all. In fact, the mentioned statements are scattered all over the ArgoUML-SPL code.

D. Localization Metrics

These metrics provide information on the static location of the syntactic elements annotated for the ArgoUML-SPL features. These metrics have been calculated only for elements of `Statement` granularity. The goal is to show the localization of the statements that have been associated to each feature, e.g. whether they occur statically at the beginning or at the end of a method body. We have

Table VII
LOCALIZATION METRICS

Feature	StartMethod	EndMethod	BeforeReturn	NestedStatement
COGNITIVE SUPPORT	3	5	0	10
ACTIVITY DIAGRAM	2	20	2	19
STATE DIAGRAM	2	19	3	12
COLLABORATION DIAGRAM	1	10	3	3
SEQUENCE DIAGRAM	0	9	3	7
USE CASE DIAGRAM	0	2	0	1
DEPLOYMENT DIAGRAM	0	0	0	3
LOGGING	127	21	89	336

defined these metrics inspired by the join point model found in aspect-oriented programming languages, such as AspectJ [19].

The localization metrics are defined as follows:

- `StartMethod`: counts the statements annotated for a particular feature that appear at the beginning of a method.
- `EndMethod`: counts the statements annotated for a particular feature that appear at the end of a method.
- `BeforeReturn`: counts the statements annotated for a particular feature that appear immediately before a `return` statement.
- `NestedStatememt`: counts the statements annotated for a particular feature that appear nested in the scope of a non-annotated and more external statement.

Table VII shows the values measured for the localization metrics.

Analysis of Localization: Among the four considered localizations, those that can be directly implemented by the join point model of AspectJ are: `StartMethod`, `EndMethod`, and `BeforeReturn`. However, for LOGGING – the typical crosscutting feature analyzed in this study – these three types of locations occurred less frequently for example than `NestedCommand`. In fact, `NestedCommand` in LOGGING is at least twice more frequent than the other locations. This result is interesting because crosscutting features may naturally emerge as candidate for aspectization. However, the physical modularization of code nested in external statements is more challenging, requiring, for instance, refactoring of the object-oriented code [20]–[22].

IV. DISCUSSION

This section presents and discusses some results that can be drawn from our study and compares these results to other studies published in the related literature. In particular, the following subsections discuss: (IV-A) some crosscutting patterns found in ArgoUML-SPL and how harmful those patterns seem to be and (IV-B) the feasibility of using aspect-oriented development techniques to extract the eight optional features of ArgoUML-SPL.

A. Recurrent Forms of Features

Recent work has shown that not all manifestations of crosscutting features are harmful to quality attributes of the system, such as modularity and stability [23]–[25]. It is therefore important to study and identify the features that present the most harmful crosscutting patterns. The results presented in Section III help us to identify three recurrent patterns of crosscutting features already documented in the literature [23], [25]: God Concern, Black Sheep and Octopus.

God Concern occurs when a feature requires a significant fraction of the system code in its implementation [25]. This type of feature was inspired by the definition of God Class by Fowler [25]. That is, as in God Class, elements that implement a God Concern feature hold too much responsibility of the software system. According to data presented by Tables II, V and VI we can observe that COGNITIVE SUPPORT has the pattern defined by God Concern. That is, this feature implementation involves a lot of the system functionality such as, for instance, 11 entire packages and 9 classes in other packages (Table V). Moreover, 16,319 lines of the system code (13.56%) are used to implement COGNITIVE SUPPORT (Table II). Fortunately, recent studies have shown little correlation between God Concern features with issues of modularity and stability [25].

Black Sheep is a crosscutting pattern which has the opposite definition of God Concern. This pattern defines a crosscutting feature that is implemented by some pieces of code spread across different classes of the system system [23]. The Black Sheep feature does not hold much responsibility and, therefore, it can be removed without drastic impact the system core functionality. For example, Black Sheep occurs when no class has the only purpose of implementing a feature. The feature is instead scattered in small parts across the system structure, such as in statements and expressions. The LOGGING feature follows the Black Sheep pattern in ArgoUML-SPL because no class is completely dedicated to the implementation of this feature (Table V). This feature also spreads over certain methods and method bodies (Table V). However, most of the feature code implements fine-grained elements, such as statements, attributes, and expressions (Table VI). Software engineers should be aware of a feature that manifests itself as Black Sheep because studies have shown a moderate to high correlation between this type

of feature and system stability [25].

Unlike God Concern and Black Sheep, the Octopus crosscutting pattern is defined by a feature that is partially modularized into one or more classes [23]. Although it is modularized in some classes, the Octopus feature also spreads over several other classes of the system. Therefore, we can identify two types of classes in this crosscutting pattern: (i) classes that are completely dedicated to the implementation of the feature representing the *octopus body* and (ii) classes that use only small parts of its code to implement the feature, representing the *tentacles* of the octopus. Based on this definition and on the data presented by the metrics discussed in Section III, we observe that the implementation of all six diagrams – State, Activity, Collaboration, Sequence, Use Case, and Deployment – follows the Octopus shape. That is, for all diagrams, we can identify classes either completely dedicated to the feature implementation (body) or using just a few of their methods to implement the feature (tentacles). Similar to Black Sheep, features that fit the Octopus pattern show moderate to high correlation with design stability [25]. Therefore, these features must be carefully evaluated by the software engineers and refactoring techniques may be applied to improve their modularity.

B. Is it Feasible to Modularize into Aspects?

The set of metrics presented in Section III provides us with important data to assess the possibility of refactoring selected features using aspect technology. That is, a detailed analysis of the information provided by the tables presented in that section allows us to evaluate which features are good candidates to this kind of refactoring.

For instance, according to Table V, COGNITIVE SUPPORT is the best modularized feature in ArgoUML-SPL. That is, its implementation is focused on some packages and specific classes of the system. However, according to the data in Table VI, this feature also uses many fine-grained elements in its implementation, only behind LOGGING and ACTIVITY DIAGRAM. Therefore, although COGNITIVE SUPPORT is partially modularized, it also offers opportunity for the use of aspects.

Compared to COGNITIVE SUPPORT, the STATE DIAGRAM, ACTIVITY DIAGRAM, SEQUENCE DIAGRAM, and COLLABORATION DIAGRAM features have an extra complexity to be refactored into aspects due to the tangled nature of their implementation, as can be seen in Table IV (TD metric).

LOGGING is often cited in the literature as a feature that has characteristics that make it a good candidate for refactoring by means of aspects [19], [26]. In our work as can be seen in Table VII, this feature showed the highest absolute number of code fragments which favors the use of aspects join points. However, in relative terms, this number is still small in comparison to the total number of statements that implement this feature (30%). Moreover, in a manual investigation, we found out that messages that are recorded by logging calls are different from each other in 84.5% of the cases. Therefore, it would require an extensive use of different aspects for each different message. In other words, if implemented using aspects, the feature certainly would require a low degree of quantification [27], [28].

Given this scenario, we can preliminarily state that it may not be worth to refactor any of the features using a technology for physical modularization of crosscutting concerns, such as aspects.

V. RELATED WORK

The extraction of ArgoUML-SPL was motivated by the fact that research in the area of product lines is usually based on demonstration systems built in laboratories. For example, three product lines are often used in such research: Expression Product Line (EPL) [4], Graph Product Line (GPL) [5], and Mobile Media Product Line (MMPL) [6]. The EPL consists of an expression grammar with the following variabilities: data types (literals), operators (negation, addition etc.) and operations (print and evaluation). The EPL version implemented in Java has about 2 KLOC. GPL is a product line in the area of graph whose variabilities include edges (directed or undirected, weighted or not etc), search methods (DFS or BFS) and some classical graph algorithms (cycle checking, shortest path, minimum spanning tree etc). Similar to EPL, the GPL has also only 2 KLOC. Finally, MMPL is a family of products for mobile devices, whose goal is to implement applications for management of multimedia files, including photos, video, and music. Besides the type of media, MMPL includes other variabilities, such as transferring documents via SMS and file management operations (creation, removal, viewing etc). MMPL has about 3.5 KLOC.

Aiming to evaluate the application of AspectJ to implement variability in product lines, Kastner, Apel and Batory extracted several features of a database manager called Berkeley DB (a system of 84 KLOC) [29]. Initially, they identified a total of 38 features in that system, including features related to persistence, transactions, caching, logging, statistics, thread synchronization etc. However, due to technical limitations of AspectJ, they failed to refactor some features with coarse-grained granularity, such as the database persistence system. The extracted features correspond to about 10% of the size of the system. Thus, since it involves a real, medium to large sized system, Berkeley DB is considered to be closest of our goals when we decide to refactor the ArgoUML.

However, the experiments with Berkeley DB and ArgoUML have some important differences: (a) in our experience, we could annotated all features originally proposed, including those with higher granularity (such as UML diagrams), (b) in our experience, the total amont of refactored code was much higher (about 37 KLOC, or 31.1% of the total system size, against 8 KLOC, or 10% of the system's size, in the case of Berkeley DB), and (c) the ArgoUML-SPL is publicly available to facilitate and encourage its use by the community of researchers in product lines. Last but not least, the two product lines

(ArgoUML-SPL and Berkley DB) are complementary because they offer researchers the opportunity to explore two approaches in two heterogeneous systems and thus allow the generalization of experimental results.

Recently, Liebig et al. conducted an extensive study aimed at evaluating how preprocessor directives are effectively used to implement variabilities [16]. Their work involved the analysis of forty systems (with sizes ranging from 10 KLOC to a 1 MLOC), all implemented in C. However, such systems have not been refactored. Therefore, the variabilities being considered include basically very low features, usually selected through command line parameters (e.g. debugging, optimization, or portability options in the case of compilers). Moreover, in our work, instead of analyzing low-level features of several legacy systems, we chose to analyze higher level features of a single up-to-date system (ArgoUML). This system was properly refactored to allow the generation of various products. Finally, we use several metrics to evaluate the extraction of ArgoUML-SPL - including metrics such as LOF and SD used in the Liebig's work.

Annually, the International Software Product Line Conference (SPLC) chooses examples of successful practical implementation of the product line principles. These examples become part of the Product-Line Hall of Fame [30]. However, these cases do not necessarily imply the physical separation or the annotation of the code responsible for implementing features. That is, in some product lines listed in the Hall of Fame, the reuse occurs only in terms of software architecture, development process, execution platforms, and common components [31]. Furthermore, the award-winning product lines are usually commercial systems which are not publicly available for conducting research.

VI. Conclusions

We have described an experience involving the extraction of eight complex features from a real software system (ArgoUML) in order to generate a software product line (that we called ArgoUML-SPL). The main contributions of our work are the following:

- The extraction of a SPL from a real and complex system. The version of ArgoUML used in this work has about 120 KLOC and we have annotated around 37 KLOC using conditional compilation directives. Clearly, such numbers distinguish ArgoUML from systems normally used to evaluate SPL-based techniques, languages, and properties. In fact, to the best of our knowledge, the amount of code refactored to create ArgoUML-SPL is the largest among all other systems evaluated on product line research.

- We have made ArgoUML-SPL source code publicly available as a subproject at the main ArgoUML web site: `http://argouml-spl.tigris.org`. Our intention is to promote the use of ArgoUML-SPL among researchers and practitioners interested on product line related topics.

- We have proposed a framework for the evaluation and characterization of pre-processor-based product lines. This framework has been inspired by a set of metrics originally proposed by Liebig et al. [16]. However, we have extended this framework with new metrics, such as those related to scattering and tangling. The extended framework supports the characterization of features according to different perspectives, including size, crosscutting behavior, granularity, and static location in the code. The combined analysis of these different perspectives allowed us to understand and to characterize the features extracted in ArgoUML-SPL. This knowledge is documented in Section III and IV to allow further replications of our study by different research groups.

As future work, we have plans to refactor other features. By making the current version of the system publicly available, we aim to motivate researchers to contribute to its extension. We also have plans to investigate the refactoring of ArgoUML-SPL features to other programming paradigms, such aspect-oriented programming, feature-orientated programming, and disciplined annotations. Actually, as stated in Section II, preprocessor directives have been selected as the first technology to extract features in our product line exactly with this objective, i.e., to provide a baseline for comparison with other modularization techniques.

Finally, the manual annotation of features, using `#ifdef` and `#endif`, has demonstrated to be a tedious and repetitive task. Therefore, we intend to investigate the use of automatic techniques and tools to extract features.

Acknowledgment

This research has been supported by grants from FAPEMIG and CNPq.

References

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

[2] V. Sugumaran, S. Park, and K. C. Kang, "Introduction to the special issue on software product line engineering," *Communications ACM*, vol. 49, no. 12, pp. 28–32, 2006.

[3] D. L. Parnas, "On the design and development of program families," *IEEE Transactions on Software Engineering*, vol. 2, no. 1, pp. 1–9, 1976.

[4] R. Lopez-Herrejon, D. Batory, and W. R. Cook, "Evaluating support for features in advanced modularization technologies," in *19th European Conference on Object-Oriented Programming (ECOOP)*, ser. LNCS, vol. 3586. Springer-Verlag, 2005, pp. 169–194.

[5] R. E. Lopez-Herrejon and D. S. Batory, "A standard problem for evaluating product-line methodologies," in *Third International Conference on Generative and Component-Based Software Engineering (GPCE)*, ser. LNCS, vol. 2186. Springer-Verlag, 2001, pp. 10–24.

[6] E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. C. Ferrari, S. S. Khan, F. C. Filho, and F. Dantas, "Evolving software product lines with aspects: an empirical study on design stability," in *30th International Conference on Software Engineering (ICSE)*, 2008, pp. 261–270.

[7] F. DeRemer and H. Kron, "Programming-in-the large versus programming-in-the-small," in *International Conference on Reliable software*, 1975, pp. 114–121.

[8] F. P. Brooks, *The mythical man-month (anniversary ed.)*. Addison-Wesley, 1995.

[9] L. Tolke, M. Klink, and M. van der Wulp, "ArgoUML cookbook," 2010, http://argouml.tigris.org/wiki/Cookbook.

[10] J. E. Robbins and D. F. Redmiles, "Cognitive support, UML adherence, and XMI interchange in ArgoUML," *Information & Software Technology*, vol. 42, no. 2, pp. 79–89, 2000.

[11] H. Spencer, "#ifdef considered harmful, or portability experience with C News," in *USENIX Conference*, 1992, pp. 185–197.

[12] B. Adams, W. D. Meuter, H. Tromp, and A. E. Hassan, "Can we refactor conditional compilation into aspects?" in *8th International Conference on Aspect-Oriented Software Development (AOSD)*, 2009, pp. 243–254.

[13] S. Apel and C. Kästner, "Virtual separation of concerns - a second chance for preprocessors," *Journal of Object Technology*, vol. 8, no. 6, pp. 59–78, 2009.

[14] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *11th European Conference on Object-Oriented Programming (ECOOP)*, ser. LNCS, vol. 1241. Springer Verlag, 1997, pp. 220–242.

[15] J. Liu, D. Batory, and C. Lengauer, "Feature oriented refactoring of legacy applications," in *28th International Conference on Software Engineering (ICSE)*, 2006, pp. 112–121.

[16] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze, "An analysis of the variability in forty preprocessor-based software product lines," in *32nd International Conference on Software Engineering (ICSE)*, 2010.

[17] M. Fowler and K. Scott, *UML distilled*. Addison-Wesley, 2000.

[18] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in software product lines," in *ICSE '08: Proceedings of the 30th international conference on Software engineering*. New York, NY, USA: ACM, 2008, pp. 311–320.

[19] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of AspectJ," in *15th European Conference on Object-Oriented Programming (ECOOP)*, ser. LNCS, vol. 2072. Springer Verlag, 2001, pp. 327–355.

[20] M. Nassau and M. T. Valente, "Object-oriented transformations for extracting aspects," *Information and Software Technology*, vol. 51, no. 1, pp. 138–149, 2009.

[21] M. Nassau, S. Oliveira, and M. T. Valente, "Guidelines for enabling the extraction of aspects from existing object-oriented code," *Journal of Object Technology*, vol. 8, no. 3, pp. 1–19, 2009.

[22] D. Binkley, M. Ceccato, M. Harman, F. Ricca, and P. Tonella, "Tool-supported refactoring of existing object-oriented code into aspects." *IEEE Transactions Software Engineering*, vol. 32, no. 9, pp. 698–717, 2006.

[23] S. Ducasse, T. Gîrba, and A. Kuhn, "Distribution map," in *International Conference on Software Maintenance (ICSM)*, 2006, pp. 203–212.

[24] M. Eaddy, T. Zimmermann, K. D. Sherwood, V. Garg, G. C. Murphy, N. Nagappan, and A. V. Aho, "Do crosscutting concerns cause defects?" *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 497–515, 2008.

[25] E. Figueiredo, B. C. da Silva, C. Sant'Anna, A. F. Garcia, J. Whittle, and D. J. Nunes, "Crosscutting patterns and design stability: An exploratory analysis," in *International Conference on Program Comprehension (ICPC)*, 2009, pp. 138–147.

[26] M. Mezini and K. Ostermann, "Conquering aspects with Caesar," in *2nd International conference on Aspect-oriented Software Development (AOSD)*, 2003, pp. 90–99.

[27] M. T. Valente, C. Couto, J. Faria, and S. Soares, "On the benefits of quantification in AspectJ systems," *Journal of the Brazilian Computer Society*, vol. 16, no. 2, pp. 133–146, 2010.

[28] R. E. Filman and D. P. Friedman, "Aspect-oriented programming is quantification and obliviousness," in *OOSPLA Workshop on Advanced Separation of Concerns*, Oct. 2000.

[29] C. Kästner, S. Apel, and D. Batory, "A case study implementing features using AspectJ," in *11th International Software Product Line Conference (SPLC)*, 2007, pp. 223–232.

[30] Software Product Line Conference, "Product line hall of fame," 2010, http://splc.net.

[31] P. Mohagheghi and R. Conradi, "An empirical investigation of software reuse benefits in a large telecom product," *ACM Transactions on Software Engineering Methodology*, vol. 17, no. 3, pp. 1–31, 2008.