FLOrIDA: Feature LOcatIon DAshboard for Extracting and Visualizing Feature Traces

Berima Andam
Chalmers University of Technology, Sweden
berima@student.chalmers.se

Andreas Burger
ABB Corporate Research, Germany
andreas.burger@de.abb.com

Thorsten Berger
Chalmers | University of Gothenburg, Sweden
thorsten.berger@chalmers.se

Michel R. V. Chaudron
Chalmers | University of Gothenburg, Sweden
michel.chaudron@cse.gu.se

ABSTRACT

Features are high-level, domain-specific abstractions over implementation artifacts. Developers use them to communicate and reason about a system, in order to maintain and evolve it. These activities, however, require knowing the locations of features—a common challenge when a system has many developers, many (cloned) variants, or a long lifespan. We believe that embedding feature-location information into software artifacts via annotations eases typical feature-related engineering tasks, such as modifying and removing features, or merging cloned features into a product line. However, regardless of where such annotations stem from—whether embedded by developers when writing code, or retroactively recovered using a feature-location technique—tool support is needed for developers to exploit such annotations.

In this tool demonstration, we present a lightweight tool that extracts annotations from software artifacts, aggregates and processes them, and visualizes feature-related information for developers. Views, such as which files implement a specific feature, are presented on different levels of abstraction. Feature metrics, such as feature size, feature scattering, feature tangling, and numbers of feature authors, are also presented. Our tool also incorporates an information-retrieval-based feature-location technique to retroactively recover feature locations.

CCS Concepts

• **Software and its engineering → Maintaining software; Abstraction, modeling and modularity; Software product lines;**

Keywords

features; feature location; visualization; tool support

1. INTRODUCTION

The notion of *feature* is commonly used when engineering software systems. A feature abstracts over concrete software artifacts, such as code, requirements or models. Developers use features for communicating and reasoning about a system, as well as keeping a comprehensive understanding of it. Using features is especially helpful when many variants of a system exist, as features provide an intuitive way of distinguishing individual variants [7, 4, 8].

Many software-engineering activities are centered around features [19], such as extending or removing a feature, propagating a feature across variants, or consolidating cloned features. All these activities require developers to understand the features that exist in a system as well as their exact locations in the artifacts. Unfortunately, maintaining or recovering feature locations can be a daunting task [10]. For instance, externally kept feature-location information may quickly become inaccurate when it is not continuously updated—a laborious and error-prone task.

We rely on feature locations documented as annotations embedded in software artifacts. Such embedded annotations have previously been shown as beneficial for engineering software with many variants [15, 19], especially when variants are not realized as a software product line [3, 11, 2] with an integrated platform, but using clone&own. Adding such annotations to artifacts is cheap, while the maintenance effort is low, as they naturally co-evolve with their artifacts (as opposed to externally kept feature locations), which significantly reduces manual updates [15].

However, tool support is needed to exploit embedded annotations. In large systems with many (cloned) variants, the number of features and annotations may be huge—challenging developers who need to quickly understand how a specific feature is realized. Thus, visualizations and aggregations of the features and their locations are required.

In this tool demonstration, we present the lightweight tool *Feature LOcatIon DAshboard* (FLOrIDA) to support feature-related engineering activities. FLOrIDA extracts embedded annotations from software artifacts (e.g., requirements and code), processes them, and presents various views visualizing features and their locations to the developer. The views comprise a feature tree, visualizations (e.g., which files or folders realize features of interest), and various metrics that help understanding core characteristics of features in a system.

FLOrIDA addresses two main use cases. Its primary use case is to continuously support developers during engineering, encouraging them to add features and annotations. In this light, FLOrIDA's views help in keeping an overview understanding of a system and its variants. FLOrIDA's secondary use case is feature-location recovery. To this end, it incorporates a simple, automated feature-location technique [24],

VaMoS '17 February 01-03, 2017, Eindhoven, Netherlands
© 2017 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-4811-9/17/02.
DOI: http://dx.doi.org/10.1145/3023956.3023967

which proposes annotations that can then be explored by developers in order to be confirmed, removed or adjusted.

Both use cases aim at supporting feature-based development without having established a software product line with an integrated platform. In other words, the embedded annotations represent traceability, but not variability information (e.g., represented using `#IFDEFs`). The latter only represent optional or alternative artifacts (e.g., code), but do not necessarily express the location of software features. In fact, this difference distinguishes FLOrIDA from common variability-management tools, such as pure::variants, Gears or FeatureIDE. Furthermore, FLOrIDA aims at being lightweight instead of depending on a larger tool-chain, specifically to support feature-based development without having established a product line relying on an integrated platform and systematic variability management. Yet, knowing feature locations is essential for merging cloned features to eventually adopt a software product line—introducing variability annotations to account for differences between cloned features or to make features optional.

The tool FLOrIDA, its source code, a user guide, and links to repositories with open-source projects that have features and embedded annotations, are available online.[1]

2. MOTIVATION AND BACKGROUND

Feature Location. Knowing the source artifacts of a software feature is required for many software-engineering tasks. However, feature locations are often not documented explicitly, and the knowledge about a feature's locations quickly deteriorates after development. Consequently, much of a developer's time is spent searching feature locations [27, 21, 10, 24] for performing feature-related tasks, such as extending or removing a feature. Moreover, if a system has many variants, tasks requiring feature-location knowledge also comprise propagating features across variants or merging cloned features when establishing a software product line.

Feature-Location Recovery. Retroactively recovering feature locations is costly and error-prone. Although automated feature-location techniques [24] might help, they usually provide a low accuracy and require substantial effort for their application—still leaving the majority of the work to the developers in large systems. Furthermore, the automatically identified locations are typically relatively coarse-grained, on the file- or function/method level, although features are often more fine-grained, comprising just one or few lines. Consequently, the identified feature locations need to be adjusted, which requires tool support for reasoning about the locations.

To prevent expensive feature-location recovery, we believe that features and feature locations should be continuously recorded during development—when the knowledge about features is still fresh in the minds of developers.

Embedded Feature Annotations. To record feature locations, two storage strategies are possible. Locations can be stored outside the source artifacts (i.e., external storage, such as in a traceability database) or directly within the artifacts. Since the external storage of feature locations (e.g., in a traceability database) is brittle and requires continuous updates, we rely on an internal storage strategy.

We adopt embedded feature annotations as proposed by Ji et al. [15]. Embedding the feature-location information facilitates keeping it updated when the codebase evolves. A case study [15] shows that the cost of creating and maintaining the annotations is low to negligible, while the benefit for feature-related activities is substantial. Specifically, 18 % of the recorded feature locations as annotations saved 90 % of feature-location costs needed for typical feature-related activities. The study also showed that many of the annotations naturally co-evolved with the artifacts (e.g., when code is moved), not requiring active annotation maintenance.

Adoption of Software Product Lines. Software product lines often arise from clone&own-based development [9]. Instead, of conceiving a product line as an integrated platform from the very beginning, with features being modeled in a feature model and an explicit mapping between *optional* features to source artifacts (e.g., using variability annotations, such as `#IFDEFs`), organizations often start with one product in order to clone and adapt it when new requirements emerge. Later, when the need arises to consolidate the cloned products into an integrated platform (a.k.a., software product line), the knowledge about features and their locations needs to be recovered—a process that can take years [14].

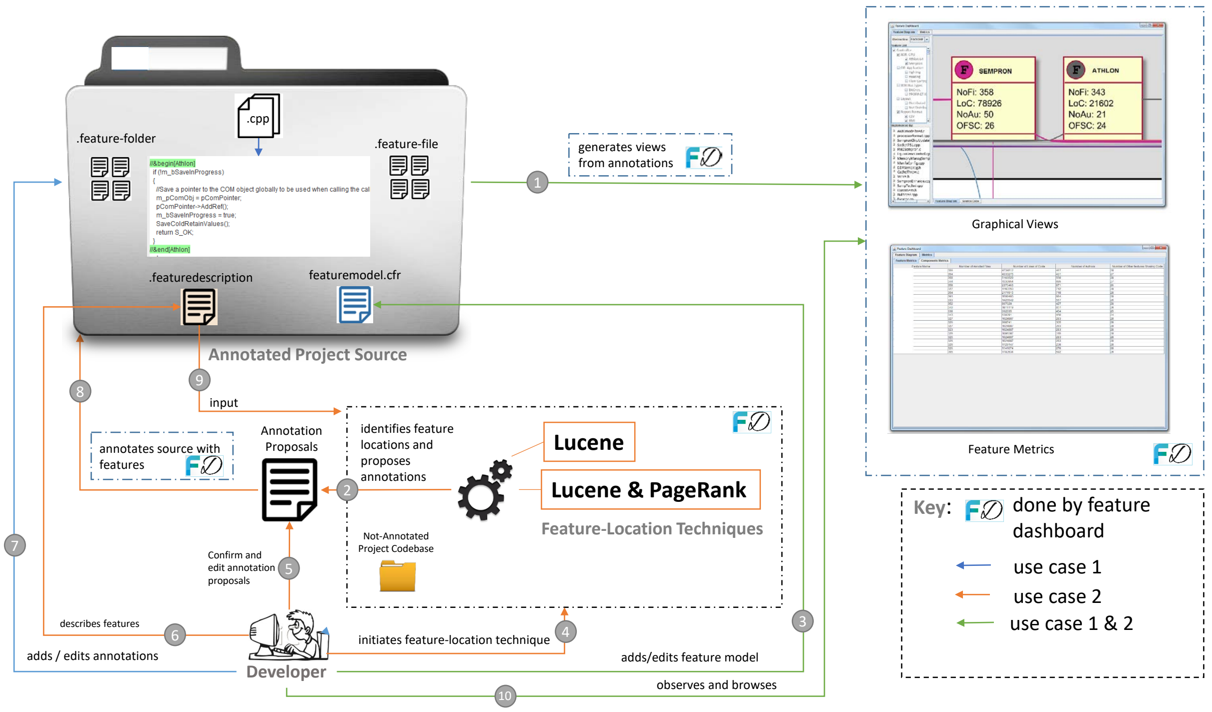
We believe that features and their locations should be recorded early—when they are implemented—to avoid high costs for retroactively identifying features and their locations. Instead of migrating clone-based products in a costly and risky process, we strive to establish a truly incremental product-line adoption process—a.k.a., the virtual-platform approach [2]. This process should support using only a subset of product-line concepts (e.g., features, traceability, configurator, preprocessor or build system), allowing a truly incremental adoption of a product line, where an incremental investment (e.g., introducing features) provides an incremental benefit (e.g., keeping an overview understanding of features across cloned products).

Product-Line Engineering Tools. Many product-line engineering and feature-modeling tools, such as pure::variants, Gears, and FeatureIDE exist. These provide facilities for modeling features and mapping *optional* features to code using variability annotations (e.g., `#IFDEFs`). However, these tools require an already established product line with an integrated platform and focus more on developing variable software artifacts, less on providing feature-traceability support. In fact, variability annotations and traceability annotations are different. For instance, consider two cloned features, denoted using traceability annotations. When merging them into an integrated platform, variability annotations will only be added around their differences. Furthermore, all these tools are full-blown IDEs. Instead, our tools strives to be lightweight, easily usable in addition to existing tools, support traceability annotations, and focuses on single or clone-based systems.

3. TOOL OVERVIEW

The tool FLOrIDA aims at encouraging developers to embed feature-location information into source artifacts and to exploit this information for feature-related engineering activities. We designed FLOrIDA as a lightweight tool that comes within one binary without requiring any installation procedure. It is implemented as a stand-alone Java program, which allows running it on any Java-supported platform. Finally, the embedded annotations are independent of the target programming language, so any kind of artifact can be annotated as belonging to a feature.

[1]https://bitbucket.org/berandam/florida



```
//&begin[Athlon]
 if (!m_bSaveInProgress)
 {
  //Save a pointer to the COM object globally to be used when calling the cal
  m_pComObj = pComPointer;
  pComPointer->AddRef();
  m_bSaveInProgress = true;
  SaveColdRetainValues();
  return S_OK;
 }
//&end[Athlon]
```

```
SEMPRON
NoFi: 358
LoC: 78926
NoAu: 50
OFSC: 26

ATHLON
NoFi: 343
LoC: 21602
NoAu: 21
OFSC: 24
```

Figure 1: Annotating and displaying feature traces

Figure 1 summarizes FLOrIDA's two main use cases. For both, a developer starts the tool and selects the root directory of the project's codebase (large, gray folder on the top-left of Figure 1) that is or should be annotated. It then builds an (internal) model containing the files in the project, the declared feature model, and then associates the files to the features based on the embedded annotations. Using this model, FLOrIDA generates graphical views and metrics (arrow ❶ in Figure 1), which supports developers who observe (arrow ❿ in Figure 1) these for reasoning about the system, keeping an overview understanding of the features, and to browse individual features. In addition, FLOrIDA provides a feature-location recovery technique to suggest feature locations in legacy (i.e., not annotated) code.

3.1 Use Case 1: Immediate and Continuous Recording of Feature Locations

FLOrIDA primarily aims at encouraging and supporting developers to continuously create and maintain traceability information (i.e., features and feature annotations) during development. Immediate and continuous annotation is beneficial as at this point, the knowledge about the feature and its location is still fresh in the developer's mind. In Figure 1, the blue arrows illustrate this use case.

For instance, when adding a feature, the developer records it in the textual feature model at the respective location in the feature hierarchy (arrow ❸ in Figure 1) and adds annotations around the newly added code belonging to the feature (arrow ❼ in Figure 1). Likewise, when removing a feature, the respective declaration in the feature model needs to be removed (arrow ❸ in Figure 1). Code changes might also require changing the feature declaration and the annotations (arrow ❼ in Figure 1). A complete list of patterns on how to create and maintain the annotations is available by Ji et al. [15].

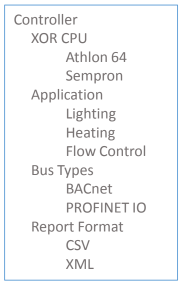
FLOrIDA observes files in the project in order to track annotation changes in the files. To make the process of tracking files efficient, FLOrIDA hooks into the file system and listens for changes to files in the project instead of continuously traversing the files checking for changes. This makes it possible for a developer to interactively change annotations in the project from her favorite IDE, with FLOrIDA's views updating accordingly. Since only files that change are updated, it is scalable enough to be used on systems of considerable size (Evaluated with 3.2 MLoC).

3.2 Use Case 2: Feature-Location Recovery

Alternatively, FLOrIDA can be used to retroactively recover and annotate artifacts with features using a built-in feature-location technique. This use case applies when feature locations were not recorded and the original developers are not (or only limited) available. In Figure 1, this use case is illustrated by orange arrows.

To use this feature, the developer must first create a feature-description file (*.featuredescription*) and place it at the project-root-directory. In this file, she must provide a description for each feature of interest using natural language (arrow ❻ in Figure 1). She can then initiate FLOrIDA's feature location (arrow ❹ in Figure 1), which uses the provided description file (arrow ❾ in Figure 1), to identify and propose feature locations for each feature (arrow ❷ in Figure 1). The developer can then confirm, reject or edit the proposed feature locations (arrow ❺ in Figure 1) before asking FLOrIDA to annotate the project source with the accepted feature annotations (arrow ❽ in Figure 1).

Recall that feature-location techniques can be very inaccurate [24] and normally require manual work. Thus, FLOrIDA can be used for reviewing the automatically proposed annotations. Specifically, when the feature-location technique proposes files belonging to a specific feature, developers can



```
Controller
    XOR CPU
        Athlon 64
        Sempron
    Application
        Lighting
        Heating
        Flow Control
    Bus Types
        BACnet
        PROFINET IO
    Report Format
        CSV
        XML
```

Figure 2: Example feature model in Clafer syntax

confirm the files. If the confirmed files contain in-file annotations, the latter are tagged so that they are distinguishable from manually created or manually confirmed ones.

3.3 Implementation

The implementation of FLOrIDA is divided into four parts.

The first part, the annotations extractor, recursively traverses files and folders of the codebase to gather embedded annotations. The parser creates an internal model of the project that contains nodes for the source artifacts (files and folders) and the features, which are associated based on the annotations. For each file, it also checks for embedded (in-file) annotations (e.g., `//&line[System Monitor]`, see Section 4). If an annotation is found, the file is associated with the respective feature(s), and the specific lines are stored in the internal model.

The second part, the metrics calculator, derives metrics based on the feature model and the annotations. No programming-language- or project-specific information is taken into account, except for the metric **NoAu** (number of authors, see Table 1), which is extracted from author information embedded in comments.

The third part, the visualization module, is responsible for rendering the graphical views. The graphical visualizations are done with the help of PLANTUML[2], which internally uses the DOT [12] graphics library.

The fourth part, the feature-location module, relies on the algorithms PageRank and Vector Space Model implemented by the Lucene[3] search engine. These algorithms have been used in previous work for feature location [5].

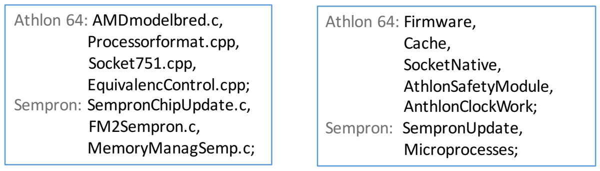
3.4 Example

In the remainder, we will explain the annotation system, present the views that our tool provides, and discuss the feature-location technique. Our running example is a commercial, embedded industrial-automation system, which was used to evaluate FLOrIDA's views and feature-location technique. The system was developed by many different teams over decades, now comprising a codebase with 3.2 MLOC of C and C++ code. All code excerpts, features, and figures are anonymized.

4. EMBEDDED ANNOTATIONS

We now explain the use of the embedded annotation system leveraged by FLOrIDA, illustrated using excerpts from our running example. First, we explain the feature model, then how files and folders are mapped to features, finally how parts of a file are mapped.

[2]http://plantuml.sourceforge.net/index.html
[3]https://lucene.apache.org



```
Athlon 64: AMDmodelbred.c,
           Processorformat.cpp,
           Socket751.cpp,
           EquivalencControl.cpp;
Sempron:   SempronChipUpdate.c,
           FM2Sempron.c,
           MemoryManagSemp.c;
```

(a) .feature-files

```
Athlon 64: Firmware,
           Cache,
           SocketNative,
           AthlonSafetyModule,
           AnthlonClockWork;
Sempron:   SempronUpdate,
           Microprocesses;
```

(b) .feature-folders

Figure 3: Examples of mapping features to files and folders

Feature Model. To initialize the project, the developer first creates a simple, textual feature model in the Clafer syntax [1], saved as featuremodel.cfr and stored in the project's root folder. Any simple text editor suffices to edit the model. The top feature should be the name of the project.

Subsequently, for both use cases, features are added to this model: one feature per line, with the feature hierarchy being expressed by indentation—a Clafer convention. Figure 2 shows an example of such a Clafer feature model.

Although the Clafer language is much richer, we only exploit it for creating a feature hierarchy. Yet, developers could also add domain-specific feature dependencies into the model (e.g., feature groups, such as the XOR group CPU in Figure 2), which could be exploited later when propagating features across cloned variants or merging variants.

To relate parts of an artifact to features, embedded annotations (escaped as comments) are used. To relate a whole artifact (e.g., source file) or a whole folder to one or more features, textual mapping files are added to the folder structure. These traces relate artifacts to features, which are declared in a simple, textual feature model in the Clafer syntax [1].

File and Folder Annotations. To associate whole source artifacts with features in the model, the developer creates special mapping files. Specifically, to associate files in a folder with a feature, a simple text file .feature-file has to be created within the folder. Each line in the file contains a mapping between one or multiple features and a file using the syntax `featureName: fileName(,fileName)*`, as shown in Figure 3a. Mapping whole folders to features is similar. The developer creates a .feature-folder file in the parent folder. Each line in the file maps a feature to one or multiple folders using the syntax `featureName: folderName(,folderName)*`, as shown in Figure 3b.

In-File Annotations. Annotating parts of (non-binary) software artifacts with features is slightly different. To annotate multiple lines, the developer simply surrounds them with a beginning tag `//&begin[featureName]` and an ending tag `//&end[featureName]`, see line 4 and 10 in Listing 1. If only one line should be associated to a feature, the developer can use a single line annotation with the syntax: `//&line[featureName]` on top of the line of code as shown for example in line 1 of Listing 1. Note that in our examples, the comments (`//`) are C/C++ specific. For other languages, the tags should be used within the respective commenting characters.

Feature References for Ambiguous Feature Names. In feature annotations, features are referenced using their least-partially-qualified (LPQ) names. These are usually just the feature names if they are unique within the feature model (which is the case for all features in our example). However, if a name is not unique, then it must be qualified partially—just enough to make the reference unique.

For example, the feature `Sempron` has a unique name in

```
////&line[System Monitor]
 void HEAPUTILModuleOp(tModOp ModOp)
 {
   //&begin[State Visualizer]
   if (ModOp == CloseModOp)
   {
     //#12024-Remove warnings
     // for GNU compiler
   }
   //&end[State Visualizer]
 }
////&line[Report Maker]
```

Listing 1: Annotated source code

the model and can be simply referred to by its name. If this name occurred twice in the model (e.g., if another feature also named `Sempron` occurred under the feature `Application`), then both must be qualified to uniquely identify them from each. Using their LPQ name, features could be referenced as `application::sempron` and `cpu::sempron`. While fully qualified names could also be used (e.g., `::controller::cpu::sempron`), they are much longer and more brittle as compared to the LPQ names when the feature model evolves.

5. FEATURE-ORIENTED VIEWS

We now present the graphical views and metrics that FLOrIDA provides for developers.

5.1 Browse Feature View

A developer can select a feature to show the artifacts annotated with the feature. She can then select a source file in order to explore and to analyze the source code. Any embedded annotations within the source file are highlighted with the assigned color of the feature. This option helps to clearly demarcate to the user where the annotated implementation of a feature begins and ends. Figure 4 shows such a demarcation.

5.2 Trace Views

A developer can select one or multiple features from FLOrIDA's displayed feature tree. Then, it displays the implementing files of the feature(s) and a graphic visualization of this relationship. When more than one feature is selected, the visualization also shows the interaction between features in terms of shared implementing source artifacts. The visualizations can be on the file level or folder level, as shown in Figures 5 and 6.

FLOrIDA's file-level-visualization (Figure 5) shows the relationship between a selected feature(s) and its implementing artifacts. When a feature is selected by a user, a feature node is created by FLOrIDA for that feature, and a color is assigned. For each of its implementing artifacts, a source node is created and an edge between the source and the feature is drawn, using the same color as assigned to the feature. The colors help to highlight the interactions between features when they are implemented by the identical source artifacts.

Developers can also explore a feature implementation on the folder level (Figure 6). When such a request is made, FLOrIDA creates a feature node just as in the case of the file-level visualization. Thereafter, for every folder that is annotated as implementing the selected feature(s) (in .feature-folder), or that contains a file implementing the feature (in .feature-files), a folder node is created. Then just as in the case of the file-level visualization, an edge is created between each feature and its associated folders.

In the case of in-file annotations (i.e., annotations on a lines of code level), FLOrIDA creates an in-file annotation node for the annotated code. An edge is then drawn from the feature to the created node using the feature's assigned color (Figure 7).

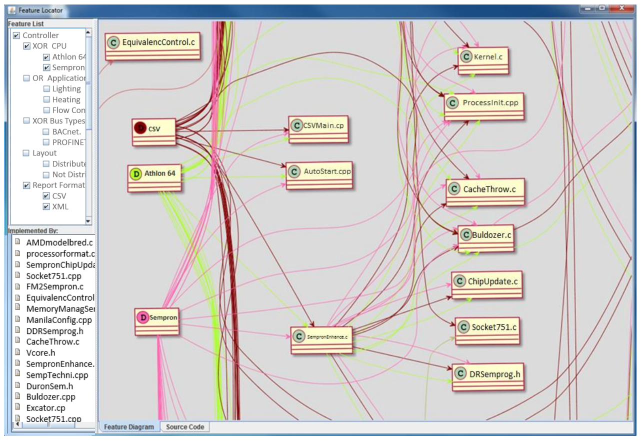
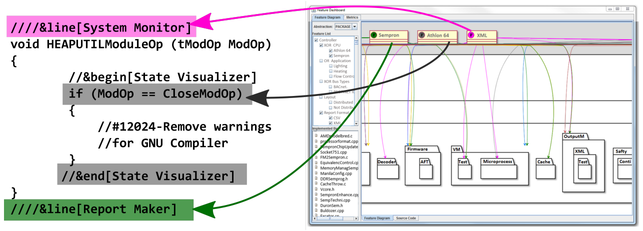


Figure 5: Feature-file trace view

5.3 Metrics Views

To enhance the understanding of feature's, we provide feature, folder, and project metrics. We define all the currently supported metrics in Table 1.

A user can view metrics related to a feature(s) of interest. The user does this by selecting the said feature(s) and then selecting the metrics tab. FLOrIDA then displays several metrics describing the feature and its implementing artifacts



```
////&line[System Monitor]
void HEAPUTILModuleOp (tModOp ModOp)
{
        //&begin[State Visualizer]
        if (ModOp == CloseModOp)
        {
            //#12024-Remove warnings
            //for GNU Compiler
        }
        //&end[State Visualizer]
}
////&line[Report Maker]
```

Figure 4: Demarcation of feature locations in source code

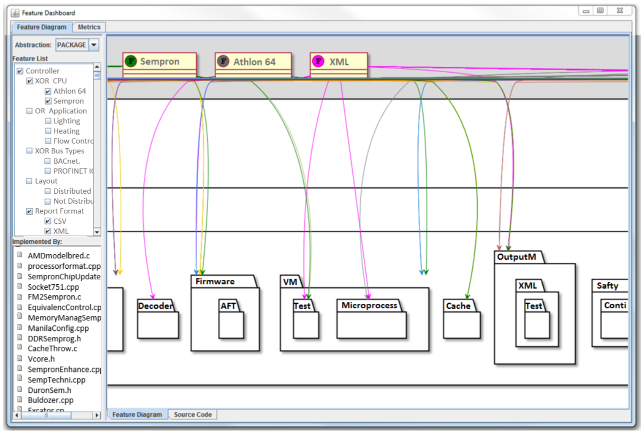


Figure 6: Feature-folder trace view

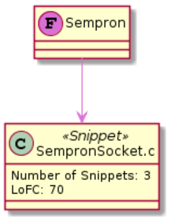


Figure 7: Visualization of an in-file annotation

and their relationship.

Feature metrics describe each feature's relationship with its implementing artifacts, as shown in Figure 8. Some of the metrics are well-known feature-related metrics, such as those provided by Liebig et al. [16] and Berger et al. [6]. We use the established terms, but even though we write "code" as in LoFC (lines of feature code), we actually count the related lines in any kind of non-binary artifact. Feature metrics are also shown directly on each feature node in the trace views, as shown in Figure 9.

Finally, the folder metrics describe each folder and its relationship to the features in the system, and the project metrics provide some aggregate numbers about all features that exist in the whole project.

6. FEATURE-LOCATION RECOVERY

Feature locations can also be retroactively recovered using FLOrIDA's built-in feature-location technique. To use this functionality, the developer has to, besides defining a feature model (featuremodel.cfr) of the system, provide a description of each feature defined in the feature model. To do this, the developer must create a description file .feature-description in the project's root folder. Inside this file, a description for each feature is specified using the syntax *featureName∼featureDescription*. When FLOrIDA's automated feature-location option is selected, it will process the project, locate, and automatically create annotations in the code-base. To distinguish them from manually created ones, a flag [auto] is appended to each line, for instance `featureName:fileName [auto]`.

Two choices of feature-location algorithms are provided from which the developer can select: Lucene and Lucene combined with PageRank. The choice allows exploring the accuracy of two different algorithms.

The Lucene algorithm uses the provided feature description information to automatically retrieve the most semantically similar source artifacts. Lucene implements the Vector Space Model algorithm used to calculate similarity of a body of text to another. Every document (in our case files; support for the method level is planned) is represented as a vector where the contents are the words in the document. Similarity between the feature's description and each document is then calculated by comparing how many unique words in the feature's description appear in the document. Unique words are obtained by using the Term Frequency/Inverse Document Frequency algorithm. Words that appear in all documents are weighted less than words that appear in a few documents. A code-file's similarity to a feature is, thus, the combined score of each unique word in the code file that also appears in the features description.

The Lucene with PageRank algorithm first uses Lucene to calculate each file's similarity with the feature of interest, but then further refines this ranked list of similar artifacts using PageRank. The latter assesses the originality or importance of a document by calculating an importance metric based on how many documents reference a document against how many documents the document itself references. A higher score is given to documents that are referenced by others, but that do not reference others.

Table 1: Feature, folder, and project metrics

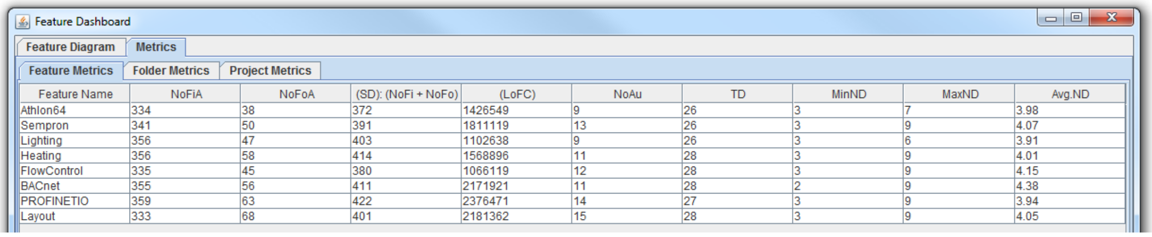
Metric	Description
	Feature Metrics
SD	Scattering Degree: total number of all annotations directly referencing the feature (i.e., in-file, folder, and file annotations referencing it)
NoFiA	Number of File Annotations: total number of file annotations directly referencing the feature
NoFoA	Number of Folder Annotations: total number of folder annotations directly referencing the feature.
TD	Tangling Degree: number of other features that share the same artifacts (or parts of such) with the feature. Two features share (parts of) artifacts when the latter is annotated with both features.
LoFC	Lines of Feature Code: lines of code *belonging* to artifacts, either directly annotated, or indirectly (when a folder is annotated, all descendants are taken into account)
ND	Nesting depths of annotations: Maximum (**MaxND**), Minimum (**MinND**), and Average (**AvgND**) nesting depth the annotations directly referencing the feature. The project's root folder has depth 0 (and so has any file contained in it). Each sub-folder increases the depth by one, a file inherits the depth of its containing folder. The depth of a (top-level, i.e., non-nested) in-file annotation is the depth of the file increased by one. Since in-file annotations can be nested, each nesting increases the depth by one. All nesting-depth metrics are calculated relative to the project root folder.
NoAu	Number of Authors who contributed to a feature's artifact. Author information is automatically extracted from author tags (format: "Author: firstname lastname") in comments wrapped by "/**" and "*/" in the source code if they exist.
	Folder Metrics
NoF	Number of Features: total number of features directly referenced in annotations (folder, file, in-file) of the folder and any of its descendants
LoFC	Lines of Folder Code: total lines in any descendant file of the folder
NoFi	Number of Files: number of all descendant files of the folder
	Project Metrics
NoF	Number of features in project
Total LoFC	Total Lines of Feature Code: sum of LoFC (all features)
Avg. LoFC	Average Feature Lines of Code: sum of LoFC (all features) / NoF
Avg. ND	Average Feature Nesting Depth: sum of ND (all features) / NoF
Avg. SD	Average Feature Scattering Degree: sum of SD (all features) / NoF

The developer can subsequently modify the annotations done by FLOrIDA by removing some or adding additional annotations. If a developer accepts one of the suggestions, FLOrIDA then automatically annotates the newly added code to reduce the effort required by the developer.

7. EVALUATION AND FEEDBACK

We conducted a preliminary evaluation. Specifically, we verified the scalability of the annotation extraction, the visualizations on the codebase, and our feature-location technique based on the original system of our running example (industrial automation system, 3.2 MLOC).

As there were no existing feature models of the system, we created one by analyzing existing user manuals, sales



Feature Name	NoFiA	NoFoA	(SD): (NoFi + NoFo)	(LoFC)	NoAu	TD	MinND	MaxND	Avg.ND
Athlon64	334	38	372	1426549	9	26	3	7	3.98
Sempron	341	50	391	1811119	13	26	3	9	4.07
Lighting	356	47	403	1102638	9	26	3	6	3.91
Heating	356	58	414	1568896	11	28	3	9	4.01
FlowControl	335	45	380	1066119	12	28	3	9	4.15
BACnet	355	56	411	2171921	11	28	2	9	4.38
PROFINETIO	359	63	422	2376471	14	27	3	9	3.94
Layout	333	68	401	2181362	15	28	3	9	4.05

Figure 8: Feature metrics view

brochures, and system documentation. A total of about 43 features was extracted from the documentation. Feature descriptions were then written in natural language for each feature.

Even though the system is considerably large, it took only about four minutes to run the feature location algorithm Lucene+PageRank and to annotate the source with the proposed annotations. Then, it took another 25 seconds to extract all 6110 embedded annotations and to generate the views and calculate the metrics.

We also conducted an interview with two experts who participated in the development of the system. The first expert, currently an architect, has worked on the system as a developer since its inception (decades ago). The second expert, also an architect, has worked on the system for more than ten years. They expressed their opinions about the tool and how it is suited for the particular system after a demonstration of 1.5h.

The experts were positive about the robustness that the embedded annotation approach could give to the documentation of feature locations, which means that a large amount of documentation time could be saved as opposed to keeping documentation externally. They believe this will give a certain amount of robustness to the documentation which they do not have currently.

They also stated that the visualization and navigation that the tool provides is necessary to benefit from the stored knowledge, which could increase exponentially for very large systems, such as the case study. They believe that the additional metrics provided by the tool helps to measure properties of the system that are useful for making future decisions about the features, such as refining the feature.

Finally, the experts were confident that the approach will work well with their currently used agile development method. A thorough and systematic evaluation together with the experts, studying the exact usage, benefits, and costs of using the tool FLOrIDA and the embedded annotations is part of our future work.

8. RELATED WORK

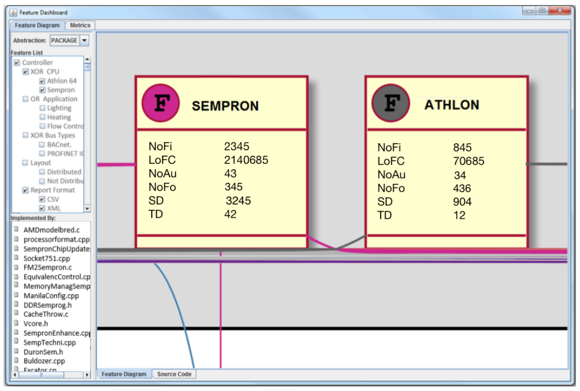
Product-Line Engineering Tools. Although they focus on variability and are heavyweight, existing product-line engineering and feature-modeling tools already provide visualizations of features.

Pure::variants provides some feature-model metrics and various views and filters to explore features and their relation to software artifacts. Gears also provides feature-related metrics to analyze the product line. FeatureIDE provides filters and views, such as visualizations of the feature-to-code mapping (e.g., a "collaboration diagram") and of the classes, methods, and views implemented by a feature.

Pleuss et al. [20] interactively visualize variability expressed in feature models of a product line. Similar to us, they present several abstracted views and filters of the features (which are configuration options) in the model, such as to illustrate cross-tree constraints, and to present consequences (with explanations) of selecting particular features.

Feature-Location Tools. When feature locations are unavailable, these can be recovered with (semi-)automated feature-location techniques. A wide range of techniques is available, requiring different kinds of input. Dynamic techniques require exercising the feature at runtime, in order to analyze call graphs. Static techniques analyze artifacts using information-retrieval and search-based techniques, among others. Many tools use a combination [22, 27, 21, 26, 28, 25, 18, 17, 10]. We made a pragmatic choice and implemented a Lucene- and PageRank-based approach. Our choice was influenced by (i) the lack of inputs required by most tools and (ii) the a lack (or unavailability) of tools for the target programming language of our example system. Dynamic feature-location techniques typically require two sets of test cases for each feature of interest, which is problematic, since most of our test cases exercise more than one feature. Furthermore, we were not able to obtain a static technique that could be applied to our example system, which was realized in C/C++. We therefore re-implemented a known technique.

Concern and Topic Visualization. Concerns and topics can be seen as similar, if not more general, concepts. Concern location has been studied intensively, and various concern visualization techniques exist, such as those by Robillard et al. [23]. Furthermore, the notion of topics is typically used to characterize developer discussions or comments. Likewise, topic visualization techniques exist. For instance, Izquierdo et al. [13] provide graphical visualizations and metrics for Github issues annotated with labels representing issue topics. Features could be seen as topics. Their visualizations are



	SEMPRON
NoFi	2345
LoFC	2140685
NoAu	43
NoFo	345
SD	3245
TD	42

	ATHLON
NoFi	845
LoFC	70685
NoAu	34
NoFo	436
SD	904
TD	12

Figure 9: Metrics shown directly in a trace view

network diagrams illustrating the label usage, label timelines, and user involvements. Our metrics and views also cover some of their information, such as the number of authors of a feature (NoAu); yet, we provide additional metrics, such as feature scattering and tangling degrees.

9. CONCLUSION

We presented a lightweight tool that supports developers in understanding and maintaining features and their feature locations, specifically supporting variant-rich systems that are not consolidated in a software product line with an integrated platform. Our tool relies on a programming-language-independent, embedded feature-annotation system. Feature annotations in a codebase are processed and visualized using different kinds of views and metrics. We used the tool to locate features and to visualize feature annotations in a large industrial system with 3.2M lines of code. We also obtained feedback from experts working on the system.

We plan to extend FLOrIDA with further metrics (e.g., process metrics extracted from an underlying version-control system) and views. Most importantly, we strive to conduct a thorough evaluation with teams of developers who need to maintain and evolve a large industrial-automation system with many variants, obtaining the developers' feedback and further suggestions. Finally, we plan to experiment with recommender systems to encourage developers to add annotations (e.g., suggesting annotations for new code).

Acknowledgment

This work is supported by the ITEA project REVaMP^2 funded by Vinnova Sweden (2016-02804) and the German Federal Ministry of Education and Research (01|S16042B).

10. REFERENCES

[1] M. Antkiewicz, K. Bąk, A. Murashkin, R. Olaechea, J. Liang, and K. Czarnecki. Clafer tools for product line engineering. In *SPLC*, 2013.

[2] M. Antkiewicz, W. Ji, T. Berger, K. Czarnecki, T. Schmorleiz, R. Lämmel, S. Stănciulescu, A. Wąsowski, and I. Schäfer. Flexible product line engineering with a virtual platform. In *ICSE*, 2014.

[3] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines*. Springer, 2013.

[4] S. Apel and C. Kästner. An overview of feature-oriented software development. *Journal of Object Technology (JOT)*, 8(5):49–84, 2009.

[5] A. Armaly, J. Klaczynski, and C. McMillan. A case study of automated feature location techniques for industrial cost estimation. In *ICSME*, 2016.

[6] T. Berger and J. Guo. Towards system analysis with variability model metrics. In *VaMoS*, 2014.

[7] T. Berger, D. Lettner, J. Rubin, P. Grünbacher, A. Silva, M. Becker, M. Chechik, and K. Czarnecki. What is a feature? A qualitative study of features in industrial software product lines. In *SPLC*, 2015.

[8] T. Berger, D. Nair, R. Rublack, J. M. Atlee, K. Czarnecki, and A. Wasowski. Three cases of feature-based variability modeling in industry. In *MODELS*, 2014.

[9] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wąsowski. A survey of variability modeling in industrial practice. In *VaMoS*, 2013.

[10] T. J. Biggerstaff, B. G. Mitbander, and D. Webster. The concept assignment problem in program understanding. In *ICSE*, 1993.

[11] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[12] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Software – Practice and Experience*, 30(11):1203–1233, 2000.

[13] J. L. C. Izquierdo, V. Cosentino, B. Rolandi, A. Bergel, and J. Cabot. Gila: Github label analyzer. In *SANER*, 2015.

[14] H. P. Jepsen and D. Beuche. Running a software product line: standing still is going backwards. In *SPLC*, 2009.

[15] W. Ji, T. Berger, M. Antkiewicz, and K. Czarnecki. Maintaining feature traceability with embedded annotations. In *SPLC*, 2015.

[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 *ICSE*, 2010.

[17] A. Marcus. Semantic driven program analysis. In *ICSM*, 2004.

[18] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic. An information retrieval approach to concept location in source code. In *WCRE*, 2004.

[19] L. Passos, K. Czarnecki, S. Apel, A. Wąsowski, C. Kästner, and J. Guo. Feature-oriented software evolution. In *VaMoS*, 2013.

[20] A. Pleuss and G. Botterweck. Visualization of variability and configuration options. *International Journal on Software Tools for Technology Transfer*, 14(5):497–510, 2012.

[21] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering*, 33(6):420–432, June 2007.

[22] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering*, 33(6):420–432, 2007.

[23] M. P. Robillard and G. C. Murphy. Representing concerns in source code. *ACM Trans. Softw. Eng. Methodol.*, 16(1), Feb. 2007.

[24] J. Rubin and M. Chechik. A survey of feature location techniques. In *Domain Engineering*. 2013.

[25] P. Shao and R. K. Smith. Feature location by ir modules and call graph. In *ACM-SE 47*, 2009.

[26] N. Walkinshaw, M. Roper, and M. Wood. Feature location and extraction using landmarks and barriers. In *ICSM*, 2007.

[27] J. Wang, X. Peng, Z. Xing, and W. Zhao. How developers perform feature location tasks: a human-centric and process-oriented exploratory study. *Journal of Software: Evolution and Process*, 25(11):1193–1224, 2013.

[28] A. Y. Yao. Cvssearch: Searching through source code using cvs comments. In *ICSM*, 2001.