Feature Location Benchmark with ArgoUML SPL

Jabier Martinez[1], Nicolas Ordoñez[2], Xhevahire Tërnava[1], Tewfik Ziadi[1], Jairo Aponte[2], Eduardo Figueiredo[3], Marco Tulio Valente[3]
[1]Sorbonne Université, CNRS, Laboratoire d'Informatique de Paris 6, LIP6, France
[2]Universidad Nacional de Colombia, Colombia
[3]Universidade Federal de Minas Gerais, Brazil
jabier.martinez@lip6.fr, nordonezc@unal.edu.co, xhevahire.ternava@lip6.fr, tewfik.ziadi@lip6.fr, jhapontem@unal.edu.co, figueiredo@dcc.ufmg.br, mtov@dcc.ufmg.br

ABSTRACT

Feature location is a traceability recovery activity to identify the implementation elements associated to a characteristic of a system. Besides its relevance for software maintenance of a single system, feature location in a collection of systems received a lot of attention as a first step to re-engineer system variants (created through clone-and-own) into a Software Product Line (SPL). In this context, the objective is to unambiguously identify the boundaries of a feature inside a family of systems to later create reusable assets from these implementation elements. Among all the case studies in the SPL literature, variants derived from ArgoUML SPL stands out as the most used one. However, the use of different settings, or the omission of relevant information (*e.g.*, the exact configurations of the variants or the way the metrics are calculated), makes it difficult to reproduce or benchmark the different feature location techniques even if the same ArgoUML SPL is used. With the objective to foster the research area on feature location, we provide a set of common scenarios using ArgoUML SPL and a set of utils to obtain metrics based on the results of existing and novel feature location techniques.

KEYWORDS

Feature location, Software Product Lines, Benchmark, Reverse-engineering, Extractive Software Product Line Adoption, ArgoUML

1 INTRODUCTION

ArgoUML [1] is an open-source tool for modeling software systems in UML. It supports all standard UML 1.4 diagram types, such as use case diagrams, sequence diagrams or class diagrams apart from other functionalities such as source code generation. A screen-shot of ArgoUML is shown in Figure 2 presenting the class diagram editor. In 2011, Couto, Valente, and Figueiredo [5] used the original Java source code of ArgoUML as a case study to extract a Software Product Line (SPL) allowing to derive variants of ArgoUML through the selection of a set of optional features. A feature is defined as a prominent or distinctive characteristic, quality or user-visible aspect of a software system or systems [8]. For example, an ArgoUML variant can be derived without the `ClassDiagram` feature making it impossible to use the diagram editor shown in Figure 2, but allowing to edit any other type of diagram. Their main objective was to provide the research community with a public [2] and realistic SPL, and also to report on the major challenges in extracting features from real-world systems using conditional compilation. Indeed, the first step for the extraction in [5] was to manually locate the source code associated to a feature which is a time-consuming, tedious and error-prone task.

The location of features is not only intended for software maintenance tasks such as a feature's bug fixing [6], but also in reengineering tasks as the one shown in the transition from ArgoUML to ArgoUML SPL. This transition was related to feature location in a single-system. However, beyond ArgoUML, feature location is also a key activity in reengineering a set of product variants (*e.g.*, variants created through clone-and-own to satisfy the needs of different customers) into an SPL [3]. In all of these cases (single-system or family of systems), the research community has proposed many approaches to try to automate the feature location activity [3, 13].

ArgoUML SPL is widely used to create software variants to evaluate feature location techniques. It is the most used case study by the extractive SPL adoption community [3] and the catalog of extractive SPL adoption case studies [10] counts more than a dozen publications that have used it for evaluating their approaches. This practice of deriving variants from an SPL to evaluate feature location techniques receives several criticism, because it is presented as a way to simulate real-world clone-and-owned variants from where we aim to locate features. On the one hand, variants "synthetically"

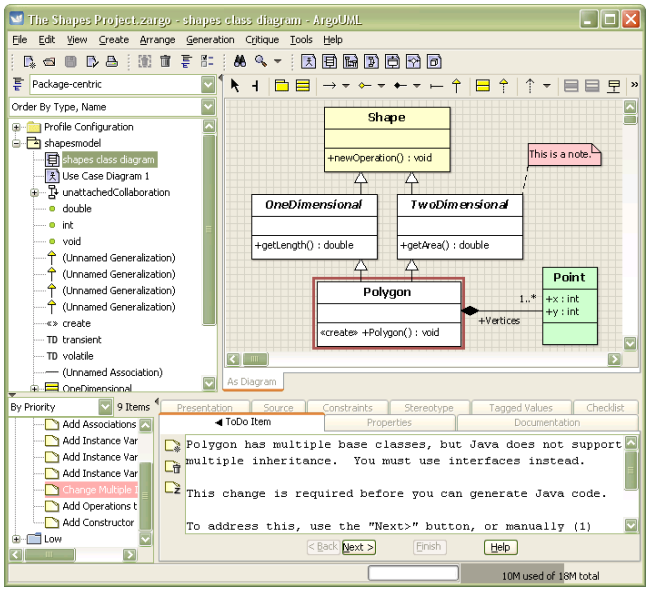

Figure 1: ArgoUML screen-shot from the documentation

[1]ArgoUML: http://argouml.tigris.org
[2]ArgoUML-SPL: https://github.com/marcusvnac/argouml-spl

created from an SPL are not realistic cases of clone-and-own practices (*e.g.*, the features are already "modularized" and there are no bug fixes or evolutions which are not propagated to all variants). On the other hand, it allows to have a non-ambiguous ground-truth from which evaluation metrics can be obtained.

Despite all the work using ArgoUML SPL as case study, it is not possible to compare them directly. There is no public, documented and common ground-truth used by all of them. For example, they use different views of the granularity of the implementation elements, there are differences in the number of ArgoUML variants used as input and in their feature configurations, or the metrics are computed in different ways. Also, some of them overlook the importance of feature interactions (some way in which a feature or features modify or influence another feature in describing or generating the system's overall behavior [16]) and negations (source code fragments needed when a feature is not present). These facts prevent the correct benchmarking of techniques even if all of them use ArgoUML variants. Given that we need more common benchmarking frameworks in our field, it is still interesting to provide synthetic cases like ArgoUML SPL variants.

This benchmark is challenging for feature location techniques because:

- ArgoUML is a project of considerable size which is the result of a real software development process with several involved developers.
- The retrieved traces must consider not only features to code mappings, but also feature interactions and feature negations. Dealing with feature interactions has been already reported as challenging for reasoning and detection [2].
- The granularity of the retrieved traces must be fine-grained and not coarse-grained as just source code files. The trace granularity is not exactly at the level of source code statements, however, it is needed to identify traces to complete Java classes or methods and more fine-grained cases where there is at least one statement associated to a feature inside a Java class or method.
- Different predefined scenarios enable to check optimistic cases (high number of variants and high diversity of feature combinations among the variants) and less optimistic cases, as well as checking techniques' scalability regarding the number of variants.

The ArgoUML SPL Benchmark with the ground-truth, the scenarios and the automatic calculation of the feature location metrics can be found at:
https://github.com/but4reuse/argouml-spl-benchmark
including technical information for how to use it.

This paper is structured as follows: Section 2 presents background information on feature location and ArgoUML SPL. Section 3 details the benchmark and Section 4 concludes the paper.

2 BACKGROUND

We present background information on feature location and more details about the ArgoUML SPL.

2.1 Feature location

The main goal of a feature location technique is to identify and establish the mapping between features and their respective implementation in the system (or in the family of systems as it is usually the case in extractive SPL adoption). This mapping is usually referred to as feature to code traces. Feature location is gaining an increasing interest by the research community with a high proliferation of techniques using different approaches such as intersection-based approaches in the case of several variants (*e.g.*, Formal Concept Analysis) static analysis (*e.g.*, Program Dependency Graphs), information retrieval (*e.g.*, Latent Semantic Indexing), dynamic analysis (*e.g.*, execution traces), search-based approaches or combinations of them [1, 3, 4, 7, 11, 13]. In the case of a family of systems, most of the feature location techniques assume that the feature presence or absence in the product variants is known upfront (*e.g.*, [7, 12]). We use this assumption also in this benchmark so the feature location technique can use this information to locate the implementation elements associated to a feature, *i.e.*, the benchmark provides this information.

For the evaluation of techniques, several case studies have been used such as ArgoUML, Linux kernel, eCos kernel or FreeBSD [4, 6, 10]. Some of those more extensively used case studies are also proposed and publicly available as benchmarks. As such, available feature location benchmarks are the Linux kernel for C code [15] or Eclipse for component-based (plugins) systems [12]. ArgoUML SPL stands out as the most used case study for evaluating feature location techniques but there is no common or established way to use it. We provide a common benchmark providing a ground-truth, a way to calculate and compare the result metrics and a set of predefined scenarios. This benchmark is complementary to the previous benchmarks mentioned before.

2.2 ArgoUML SPL

ArgoUML is a Java-based open source tool with 120 KLOC [5] and ArgoUML SPL is an extracted SPL from ArgoUML [5]. The respective code for the implementation of its eight studied features has been annotated using conditional compilation directives. The annotated features are `State Diagrams`, `Activity Diagrams`, `Use Case Diagrams`, `Collaboration Diagrams`, `Sequence Diagrams`, `Use Case Diagrams`, `Deployment Diagrams`, `Sequence Diagrams`, `Cognitive Support` and `Logging`. `Cognitive Support` is sometimes referred in ArgoUML to as `Design critics` so it was added as a synonym for the benchmark. Figure 2 presents the feature model of ArgoUML SPL with these eight optional features.

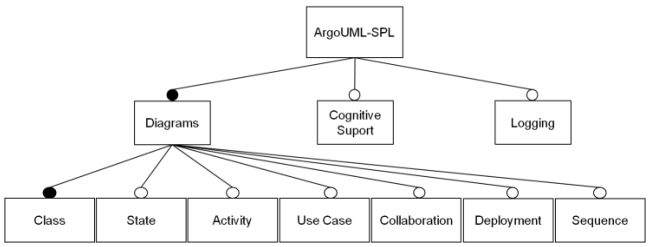

Figure 2: ArgoUML-SPL feature model [5]

Table 1 shows a brief definition and the number of LOC required to implement each of these optional features. Other metrics related to the granularity of the annotations or the place of the annotations inside the source code can be consulted [5].

Among all the features in ArgoUML, these features were selected in [5] because (1) they represent both relevant functional requirements (*e.g.*, UML diagrams) and non-functional requirements (*e.g.*, Logging) and (2) Logging and Cognitive features are crosscutting in ArgoUML. This crosscutting behavior means that the implementation of each feature requires a considerable amount of code spread over several methods, classes, and packages that might belong to other features. This is challenging for feature location techniques if we compare it with loosely coupled features making little use (*e.g.*, source code dependencies) of other source code beyond the feature source code.

Since Java does not provide native support for preprocessor directives, javapp [3] was used to annotate the feature code. This tool extends Java by supporting preprocessor directives similar to the ones that exist in C/C++, including `#ifdef`, `#ifndef`, and `#else`. 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 define different configurations (feature combinations) and generate different variants of ArgoUML SPL (*e.g.*, all features except one optional feature). The feature model of ArgoUML SPL enables to define 256 different configurations and the preprocessor directives enable their derivation into ArgoUML variants.

The granularity of the fragments can range from coarse-grained (*e.g.*, annotating a whole class) to fine-grained (*e.g.*, a source code statement). For example, the excerpt at the first line of the actual Java class *SequenceDiagramGraphModel* will make that the class will only be present if `Sequence` diagram was selected:

```
//#if defined(SEQUENCEDIAGRAM)
```

As another actual example, this time of the fine-grained category, the following source code statement will be only included in the Java class *ArgoEventPump* if the `Logging` feature was selected.

```
//#if defined(LOGGING)
LOG.error("Invalid event:" + event.getEventType());
//#endif
```

1,287 pieces of Logging code like this one are annotated with conditional compilation directives. The ArgoUML SPL has feature interactions as result of preprocessor directives like the following annotation:

```
//#if defined(COGNITIVE) and defined(DEPLOYMENTDIAGRAM)
```

This is an example how the class *CrInterfaceWithoutComponent* is annotated, meaning that it will be present in the final software product only when both features `Cognitive` and `DeploymentDiagram` are selected. Another case where certain code is only present with a combination of features is when, for example, in a class where having the annotation for the whole class to a feature, we also have a method inside this class annotated with another feature. The method will only be present when both features are selected (*e.g.*, the class *ActionAddClassifierRole*).

[3] javapp: http://www.slashdev.ca/javapp

Table 1: Optional features in ArgoUML SPL. Feature names and descriptions from [5], and number of lines of code (LOC)

Feature	Description	LOC
State Diagrams	State diagrams are used to describe the behavior of a system. State diagrams describe all of the possible states of an object as events occur. Each diagram usually represents objects of a single class and track the different states of its objects through the system.	3,917
Activity Diagrams	Activity diagrams describe the workflow behavior of a system. Activity diagrams are similar to state diagrams because activities are the state of doing something. The diagrams describe the state of activities by showing the sequence of activities performed.	2,282
Use Case Diagrams	A use case is a set of scenarios that describing an interaction between a user and a system. A use case diagram displays the relationship among actors and use cases. The two main components of a use case diagram are use cases and actors.	2,712
Collaboration Diagrams	Collaboration diagrams are used to show how objects interact to perform the behavior of a particular use case, or a part of a use case. Along with sequence diagrams, collaborations are used by designers to define and clarify the roles of the objects that perform a particular flow of events of a use case. They are the primary source of information used to determining class responsibilities and interfaces.	1,579
Deployment Diagrams	A deployment diagram models the run-time architecture of a system. It shows the configuration of the hardware elements (nodes) and shows how software elements and artifacts are mapped onto those nodes.	3,147
Sequence Diagrams	Interaction diagrams model the behavior of use cases by describing the way groups of objects interact to complete the task. Sequence Diagram displays the time sequence of the objects participating in the interaction. This consists of the vertical dimension (time) and horizontal dimension (different objects).	5,379
Cognitive Support, Design critics	Simple agents that continuously execute in a background thread of control. They analyze the design as the designer is working and suggest possible improvements. These suggestions range from indications of syntax errors, to reminders to return to parts of the design that need finishing, to style guidelines, to the advice of expert designers.	16,319
Logging	The purpose of debug log and trace messages is to provide a mechanism that allows the developer to enable output of minor events focused on a specific problem area and to follow what is going on inside ArgoUML.	2,159

3 ARGOUML SPL BENCHMARK

The challenge of the ArgoUML SPL benchmark is to implement feature location techniques maximizing information retrieval metrics: precision, recall, F1 score, and time performance detailed in Section 3.4.

For this, the benchmark provides:

- The list of features to locate based on the eight optional features defined in ArgoUML SPL. Also their descriptions as presented in ArgoUML SPL. Names and descriptions are the same as in Table 1.
- A set of scenarios with a set of ArgoUML variants each. We describe the scenarios in Section 3.1.
- A common ground-truth to be used for all the techniques based on the analysis of ArgoUML SPL's feature annotations. The feature location results should be adapted to have the same format as the ground-truth. We describe this format in Section 3.2 and we present details about how the ground-truth was extracted in Section 3.3.
- A Java program to automatically calculate and plot the metrics based on the feature location results and the ground-truth. We describe the metrics in Section 3.4.

3.1 Scenarios

We defined and provided the utils to create the following set of scenarios from the ArgoUML SPL. This diversity of scenarios can help to better characterize feature location techniques so authors should provide the results for each of them.

- **Traditional scenario**. A set of 10 variants that was used in some of the references in the literature (*e.g.*, [1]). One variant without all the optional features, one with all the features, and then, for each feature, one variant with all the features enabled and this feature disabled.
- **Randomly selected variants**. A set of eleven scenarios of 100, 50, 10, 9, 8, 7, 6, 5, 4, 3 and 2 variants respectively created through a random selection of features. The random selection process consists of two steps: 1) randomly selecting the desired number of configurations without repetition from the 256 possible variants, 2) checking if the eight features of ArgoUML SPL are covered in the selected configurations and going back to step 1 if they are not covered. These scenarios aim to show how sensible is the feature location technique to the number of variants. An example of the interest of this kind of scenarios can be observed in [7].
- **Pair-wise feature coverage**. A set of 9 variants. From the feature model, we used the T-wise algorithm from FeatureIDE [14] (version 3.3.0) to get these pair-wise configurations. Given the variants diversity, these 9 variants can show differences with the scenario of 9 variants that were randomly selected.
- **Original ArgoUML**: Only one system, the one that was used by Couto *et al.* [5] to create ArgoUML SPL. This case is representative to show the results when we can implement an SPL from a single-system instead of from previous clone-and-owned variants.
- **All variants**. The 256 possible variants of ArgoUML SPL obtained from FeatureIDE [14]. This case is mainly intended to show the scalability of the technique regarding the number of variants.

The benchmark provides ant scripts to create the scenarios. These scripts internally use a modification of the ant scripts from SPLEvo [9]. The benchmark script also removes all the variability-related source code comments as Couto *et al.* added extra meta-data in each variability annotation. This way, techniques relying on comments in the source code will not be "polluted" by variability-related comments and they will be able to use all the other source code comments.

This pre-processing is needed to prevent that techniques using source code comments will use this information.

3.2 Expected format for the results of the feature location technique

The results of the feature location technique should be adapted to conform to the same format as the provided ground-truth. The ground-truth (the actual mapping from features to implementation elements) of the ArgoUML SPL is extracted from the pre-processor directives through an automatic program that we make available. However, to use the benchmark, there is only need to consume the provided ground-truth.

The expected format of any feature location technique using the benchmark is a set of txt files where the file name is the name (id) of the feature (*e.g.*, `COLLABORATIONDIAGRAM`), or feature combination (*e.g.*, `LOGGING_and_USECASEDIAGRAM`) or negation (*e.g.*, `not_COGNITIVE`). In the case of feature combinations, the order must be alphabetical. Then, inside of these files, there is a set of lines where each line represents a *trace* that can be of four types:

- **Class qualified name**: The whole class is only present if the feature is selected.
  Example: *org.argouml.uml.ui.ActionCollaborationDiagram*
  This class is only present if `COLLABORATIONDIAGRAM` is selected.
- **Method qualified name**: The whole method is only present if the feature is selected.
  Example: *org.argouml.uml.ui.ActionNewDiagram createCollaboration(Object)*
  This method is only present if `COLLABORATIONDIAGRAM` is selected. The class containing this method (ActionNewDiagram) will be present independently of this feature, *i.e.*, it is only the method presence which depends on this feature and not the class. Given that different methods can have the same name (method overloading), the comma-separated list of the types of the argument list must be provided.
- **Class qualified name plus "Refinement" tag**: The class has imports or variable declarations which are only present if the feature is selected. This does not include when a whole method depends on a feature as this will correspond to the Method qualified name type.
  Example: *org.argouml.ui.explorer.ExplorerPopup Refinement*
  The ExplorerPopup class does not belong to any of the features but it imports classes belonging to features. For example, in the import declarations we find the source code line
  `import org.argouml.uml.diagram.activity.ui.UMLActivityDiagram;`
  which will be present only if the `ACTIVITYDIAGRAM` feature is selected.
- **Method qualified name plus "Refinement" tag**:
  The method has statements which are only present if the feature is selected.

Example: *org.argouml.ui.explorer.ExplorerPopup initMenuCreateDiagrams() Refinement*
This method has the following statement

```
createDiagrams.add(new ActionActivityDiagram());
```

that will be present only if `ACTIVITYDIAGRAM` is selected.

In this benchmark, we do not consider the location of the exact statements, but the feature location technique should be able to detect, at least, that a refinement is happening at method or class level ("Refinement" tag). The information about the exact statements is available in ArgoUML SPL through the annotations, however, we decided not to require this level of detail in this version of the benchmark as it might highly complicate the expected format of the feature location results. We also do not consider the ordering of the traces (*e.g.*, [7]), that means that in our format, the trace lines inside a file can be provided in any order.

The benchmark provides a simplistic example of a feature location technique to illustrate how they can be implemented. This technique, just uses the first name of each feature and visits the variants (those from the scenario containing this feature) to create class traces for the Java classes containing this feature name in the class name. Only the feature traces that are present in all variants containing this feature are kept.

3.3 The ground-truth and its extractor

The ground-truth contains a total of 24 txt files corresponding to each feature (8 files), existing feature combinations (13 files of pair-wise feature interactions and 1 file of three-wise feature interaction) and feature negations (2 files). In these 24 files, the ground-truth has a total of 439 traces to complete classes, 44 traces to complete methods, 388 traces of class refinements and 871 traces of method refinements. We present the ground-truth extractor that we implemented to clarify how the ground-truth was obtained and to give more details about the traces' format and granularity. The following source code is an illustrative example of a Java class with javapp annotations:

```
//#if defined(A)
package myPackage;
public class HelloWorld {
    //#if defined(B)
    public int x = 0;
    //#endif
    //#if defined(C)
    public static void sayHello() {
        System.out.print("Hello");
        //#if defined(D)
         System.out.println("World");
        //#endif
    }
    //#endif
}
//#endif
```

The resulting traces for the extracted ground-truth will be:

- `A`: *myPackage.HelloWorld*
- `A_and_B`: *myPackage.HelloWorld Refinement*
- `A_and_C`: *myPackage.HelloWorld.sayHello()*
- `A_and_C_and_D`: *myPackage.HelloWorld.sayHello() Refinement*

In the second case, it is `A_and_B` because the whole class will not exist if `A` is not present. As an example to illustrate feature negations, we can have:

```
package myPackage;
public class HelloWorld {
    public static void sayHello() {
        System.out.println("Hello");
        //#if defined(A)
        System.out.println("World");
        //#else
        System.out.println();
        //#endif
    }
}
```

Given the javapp annotation //#else, the method has statements that must be there if `A` is not selected. The resulting traces for the extracted ground-truth will be:

- `A`: *myPackage.HelloWorld.sayHello() Refinement*
- `not_A`: *myPackage.HelloWorld.sayHello() Refinement*

If the technique is using the Java JDT parser (as the ground-truth extractor does), we provide a helper class to obtain the expected class and method qualified names. Otherwise, the developers will need to find a way to conform to this format.

3.4 Metrics

We use two traditional measures for retrieval effectiveness: **precision** and **recall**. A feature location technique assigns a set of traces to each feature. From now onwards, we will use the word *features* to indicate features, feature combinations and feature negations. Correctly retrieved traces are *true positives* (TP) which are also known as *hit*, and incorrect ones are *false positives* (FP) which are also known as *false alarms*. Precision, as shown in Equation 1, is the percentage of hits relative to the total of retrieved traces by the technique.

$$precision = \frac{TP}{TP + FP} = \frac{traces\,hit}{traces\,hit\ +\ traces\,false\,alarm} \quad (1)$$

Ground-truth traces which are not included in the retrieved set are *false negatives* (FN) which are also known as *miss*. Recall, as shown in Equation 2, is the percentage of correctly retrieved traces from the set of the ground-truth.

$$recall = \frac{TP}{TP + FN} = \frac{traces\,hit}{traces\,hit\ +\ traces\,miss} \quad (2)$$

The feature location technique precision and recall is then calculated as the average of the precision and recall of all the features in the ground-truth. The benchmark also provides the traditional **F1 score** (F-measure) which relates precision and recall as shown in Equation 3.

$$F1score = 2 * \frac{precision * recall}{precision + recall} \quad (3)$$

In addition to these metrics, we provide others that can help in comparing the effectiveness of a technique. There can be cases where, for a given ground-truth feature, the feature location technique does not retrieve any trace. In the case where there are no

traces hit nor false alarm, the equation for precision is not applicable as the denominator is zero. We decided that the benchmark will return zero precision for this feature. However, for not to omit this information, the benchmark metrics also report the features from the ground-truth where nothing was retrieved (no correct nor incorrect). We refer to the total number of these cases as **features without retrieved**. In addition, it also reports the number of retrieved features which do not correspond to any feature in the ground-truth (e.g., the technique retrieves traces for `Y_and_Z` but there is no `Y_and_Z` in the ground-truth). We refer to the total number of these cases as **inexistent features retrieved**.

The Java program for the calculation of the metrics also outputs a gnuplot script to create boxplot graphs for precision, recall and F1 score. Finally, the **time performance** of the feature location technique should be measured separately by the providers of the feature location technique and reported with a description of the system used to launch it.

3.5 Extra assets and human-in-the-loop

The benchmark provides the variants and the feature information that will enable to use automatic static analysis and information retrieval techniques. However, some techniques may want to use extra assets. The following list is not exhaustive but it shows examples of available assets:

- Documentation such as the user manual [4]
- Architectural information and technical documents that can be found in the developer resources wiki [5]
- Version control system and commits information found in the official software versioning and revision control system [6] or in reports of open source projects' analysis tools [7]
- Information of enhancement requests and bug reports found in the issue tracking system [8]
- Execution traces from existing ArgoUML tests or from real usage
- Software engineering ontologies

Any extra assets beyond the ones that we provide can be used but they should be clearly mentioned and explained. If the feature location technique is semi-automatic, it should be clearly explained which are the steps that requires the user and which is the interaction protocol and the provided visualizations.

4 CONCLUSION

We have presented a feature location benchmark that uses the source code base of ArgoUML and ArgoUML variants as a challenging ground for feature location techniques. A public and unambiguous ground-truth of feature to code traces is provided in an easy format. This ground-truth was automatically extracted thanks to a program that analyzed the conditional compilation directives available in ArgoUML SPL. The benchmark provides several scenarios regarding the number of variants and their characteristics, and a program to calculate and plot common comparable metrics is provided as well.

[4] ArgoUML user manual: http://argouml-stats.tigris.org/documentation
[5] ArgoUML developers designs: http://argouml.tigris.org/wiki/Design
[6] ArgoUML svn: http://argouml.tigris.org/source/browse/argouml/trunk/src/
[7] ArgoUML open source project metrics: https://www.openhub.net/p/argouml
[8] ArgoUML bug tracking system: http://argouml.tigris.org/project_bugs.html

Several authors of feature location techniques have been using ArgoUML SPL variants. However, and given that they used different settings and scenarios among them, this benchmark is the first opportunity to share a common framework towards an unbiased comparison of precision, recall and F1 score that can foster the research on the field of feature location for extractive SPL adoption.

ACKNOWLEDGMENTS

We would like to thank Marcus Vinicius Couto for creating ArgoUML SPL as well as Benjamin Klatt for creating helpful build scripts to ease the derivation of ArgoUML SPL variants. This work was partially supported by the ITEA3 15010 REVaMP^2 project: FUI the Île-de-France region and BPI in France.

REFERENCES

[1] Ra'Fat Al-Msie'deen, Abdelhak Seriai, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, and Hamzeh Eyal Salman. 2013. Feature Location in a Collection of Software Product Variants Using Formal Concept Analysis. In *ICSR (Lecture Notes in Computer Science)*, Vol. 7925. Springer, 302–307.

[2] Sven Apel, Sergiy S. Kolesnikov, Norbert Siegmund, Christian Kästner, and Brady Garvin. 2013. Exploring feature interactions in the wild: the new feature-interaction challenge. In *5th International Workshop on Feature-Oriented Software Development, FOSD '13, Indianapolis, IN, USA, October 26, 2013*, Andreas Classen and Norbert Siegmund (Eds.). ACM, 1–8. https://doi.org/10.1145/2528265.2528267

[3] Wesley K. G. Assunção, Roberto E. Lopez-Herrejon, Lukas Linsbauer, Silvia R. Vergilio, and Alexander Egyed. 2017. Reengineering legacy applications into software product lines: a systematic mapping. *Empirical Software Engineering* 22, 6 (2017), 2972–3016.

[4] Wesley Klewerton Guez Assunção and Silvia Regina Vergilio. 2014. Feature location for software product line migration: a mapping study. In *Proceedings of the 18th International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools-Volume 2*. ACM, 52–59.

[5] Marcus Vinicius Couto, Marco Tulio Valente, and Eduardo Figueiredo. 2011. Extracting Software Product Lines: A Case Study Using Conditional Compilation. In *15th European Conference on Software Maintenance and Reengineering, CSMR 2011, 1-4 March 2011, Oldenburg, Germany*. IEEE Computer Society, 191–200.

[6] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. 2013. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process* 25, 1 (2013), 53–95. https://doi.org/10.1002/smr.567

[7] Stefan Fischer, Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2014. Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants. In *ICSME*. IEEE Computer Society, 391–400.

[8] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report. Carnegie-Mellon University Soft. Eng. Institute.

[9] Benjamin Klatt. 2014. *Consolidation of Customized Product Copies into Software Product Lines*. Ph.D. Dissertation. Karlsruhe Institute of Technology, Germany. http://digbib.ubka.uni-karlsruhe.de/volltexte/1000043687

[10] Jabier Martinez, Wesley K. G. Assunção, and Tewfik Ziadi. 2017. ESPLA: A Catalog of Extractive SPL Adoption Case Studies. In *SPLC*. ACM, 38–41.

[11] Jabier Martinez, Tewfik Ziadi, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2015. Bottom-up adoption of software product lines: a generic and extensible approach. In *Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015*. ACM, 101–110. https://doi.org/10.1145/2791060.2791086

[12] Jabier Martinez, Tewfik Ziadi, Mike Papadakis, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. Feature Location Benchmark for Software Families Using Eclipse Community Releases. In *15th International Conference, ICSR 2016, Limassol, Cyprus, June 5-7, 2016, Proceedings (Lecture Notes in Computer Science)*, Vol. 9679. Springer, 267–283.

[13] Julia Rubin and Marsha Chechik. 2013. A survey of feature location techniques. In *Domain Engineering*. Springer, 29–58.

[14] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. 2014. FeatureIDE: an extensible framework for feature-oriented software development. 79, 0 (2014), 70–85.

[15] Zhenchang Xing, Yinxing Xue, and Stan Jarzabek. 2013. A large scale Linux-kernel based benchmark for feature location research. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. IEEE Computer Society, 1311–1314.

[16] Pamela Zave. 2009. Modularity in Distributed Feature Composition. In *Software Requirements and Design: The Work of Michael Jackson*.