ULRR

# FLINTS: A tool for architectural-level modeling of features in software systems

| Item Type | Meetings and Proceedings |
| --- | --- |
| Authors | Buckley, Jim;Rosik, Jacek;Herold, Sebastian;Wasala, Asanka;Botterweck, Goetz;Exton, Chris |
| Citation | ECSAW 2016: Proceedings of the 2016 European Conference on Software Architecture Workshops;article no: 14 |
| Publisher | Association for Computing Machinery |
| Download date | 2026-05-12 00:49:41 |
| Item License | https://creativecommons.org/licenses/by-nc-sa/1.0/ |
| Link to Item | https://hdl.handle.net/10344/5506 |

# FLINTS: A Tool for Architectural-level Modeling of Features in Software Systems


Jim Buckley
Lero/CSIS
University of Limerick
Castletroy, Limerick
00353 (0)61213531
jim.buckley@ul.ie

Jacek Rosik
Lero/CSIS
University of Limerick
Castletroy, Limerick
00353 (0)61213028
jacek.rosik@lero.ie

Sebastian Herold
Department of Mathematics
and Computer Science
Karlstad University, Karlstad
0046 (0)54 7002104
sebastian.herold@kau.se

Asanka Wasala
Lero/CSIS
University of Limerick
Castletroy, Limerick
00353 (0)61213028
asanka.wasala@lero.se

Goetz Botterweck
Lero/CSIS
University of Limerick
Castletroy, Limerick
00353 (0)61234309
Goetz.botterweck@lero.ie

Chris Exton
Lero/CSIS
University of Limerick
Castletroy, Limerick
00353 (0)61213108
chris.exton@lero.ie



## ABSTRACT

Reflexion Modeling has been proposed as a means of creating and refining a functional model of software systems at the architectural level. Such a model allows developers to maintain a consistent understanding of the relationships between different functionalities of their system as it evolves, and allows them to analyze the system at a functional-chunk level rather than at the traditional, structural levels more typically presented by IDEs.

This paper describes a prototype tool built to enable this approach. The tool assists developers in moving to this functional perspective by supporting them as they first attempt to locate specific functionalities in the code. This support is based on design principles identified by observing experienced software developers in-vivo, as they performed this task manually. After the code associated with several such functionalities is located in the code-base, a graphical view allows the developer to assess the source code dependencies between these features and between these features and the rest of the system. This helps the developer understand the inter-functional interfaces, and the representation can be reviewed over time, as features are added and removed, to ensure on-going consistency between the architect's perspective of the features in the system and the code-base.



## CCS Concepts

**• Software and its Engineering → Software System Structures →Software Architectures**

## Keywords

Software Architecture; Feature Modeling; Feature Location; Consistency.



Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.


## 1. INTRODUCTION

A feature can be defined as a user-observable unit of system functionality [1]. There are many scenarios where a feature perspective of a software system is useful. For example, when an end-user comes across a bug, they typically describe it in terms of the functionality they were performing at the time, not in terms of its location in the code-base. It is then up to the developer to locate that functionality in the code-base. Likewise, a development team charged with migrating a system to services, where each service equates to a user-observable functionality, must first locate the code that implements each specified functionality, before trying to extract it and encapsulate it with an interface.

In considering this latter scenario, the prototype described in this paper focuses on helping the developers of software systems to obtain a feature-based perspective of their systems: a perspective whereby the features and their inter-feature dependencies are made explicit. The first step in this approach is to help developers locate features in the code-base, a large academic field in itself [1], [2]: In complicated software systems, code for a single feature can be scattered over dozens of files and folders. This increases the amount of code navigation and analysis necessary to locate the entire feature and understand how it interacts with other parts of the system, thus prompting loss of context among even experienced developers and generating a considerable mental burden [4]. These issues have prompted research into alternative, automated Feature Location (FL) techniques, but full automation of the process seems unrealistic [14].

Hence, the tooling presented in this paper is semi-automatic in supporting the activity of feature location. It does this by allowing the developer in question state an initial 'code foothold' or location in the code where the feature is instantiated. It then presents them with views of the system that allows them expand and refine the code location to that of the full feature. It does so by adopting design principles derived from an observational study where experienced software developers were asked to perform this task manually, in-vivo. The tool aims to be congruent with the

practices they employed and present them with the information they seemed to require as they located all the places in the commercial code-base they were maintaining, where the feature was implemented.

After several features are located in this manner, a graphical representation of the features and their (source code) dependencies can be presented by the tool, providing a graphical overview of the system's functionality at an architectural level of abstraction. In explicitly presenting the inter-dependencies between features and allowing developers to explore them at a source code level, the view allows those developers to refine their thoughts on a feature's locations in the code-base, and helps them identify both 'provides' and 'depends-on' interfaces for each feature. Finally it allows them achieve a consistency between the code-base and the feature model over the evolution of the system.

The design principles for the tool, primarily derived from observing experienced software developers as they manually located features [3], are presented in Section 2 of this paper. (The previous reporting of this study focused on participants' successful/unsuccessful usage of IDE and Unix searching tools like grep during FL). This is followed by a description of the tool in Section 3. Future directions, including a suggested evaluation for the tool, are then proposed in Section 4 and the paper concludes in Section 5.

# 2. EMPIRICALLY DERIVING DESIGN PRINCIPLES

Two experienced professional software engineers were observed during the study [3]. Participant 1 had 35 years of professional experience, working on the commercial system under study for 12 years. Participant 2 had 23 years of professional experience, working on the system for 2 years. The subject system is a Financial Services system and is approximately 3 MLOC in size. It was released about 34 years ago, consisting predominantly of COBOL and several proprietary Domain Specific Languages (DSLs), which are often interleaved within the same files. Some components are also written in Java and C although this study focussed on COBOL and the DSLs. There are approximately 40 developers working on the system at the moment.

The company is interested in incrementally re-engineering this flagship system to a set of interacting feature-based services. The plan for the modernization process is to perform it, one feature/service at a time, to limit the impact on their customers. In this scenario, the feature selected to be modernized must first be located in the source code and, its interaction with rest of the system must be made explicit. After time, as other features are identified the interactions between features will become equally important. Hence, they are interested in Feature Location Techniques (FLTs) and in determining a set of appropriate interfaces for each resultant feature. The features selected for this study were the company's candidates for modernization to services, making the study more ecologically valid [16]. Participant 1 performed FL on two features while Participant 2 performed it on only one feature. At the start of the sessions, participants were asked to find the complete feature of interest in the code-base using their normal approaches and tools (Emacs and TextPad editors, along with Unix command-line facilities like grep, find and ls).

## 2.1 Data Gathering and Analysis

At the start of their sessions participants were asked to 'state everything that comes into your mind, as it comes into your mind' and when they fell silent, they were prompted by a researcher saying 'what are you thinking now?' This is in line with best-practice for capturing think-aloud data [5]. The researcher helped the participant with any technical issues that arose during their sessions. The screens and think-aloud data of the participants, as they carried out their FL tasks, were recorded using Camtasia [6] screen capturing software.

The data generated in each session underwent immersive analysis by two researchers independently, using an open coding/memoing protocol [7] directed at work-practice detection. These independent results were cross-checked and reconciled between the two researchers. Finally, a series of group meetings was held to reconcile the findings across sessions across all researchers and the results presented back to the participants for verification.

## 2.2 Resultant Design Principles

One of the key findings was that participants were able to identify a source code foothold into their features with relative ease: they were quickly able to recall certain implementation artifacts that were parts of the feature's code-base and were always certain of their choice. This suggests the first design principle: ***(a) that software engineers familiar with the system find footholds into features quickly and accurately. Hence, for this user-profile, support for finding a feature foothold should be excluded.***

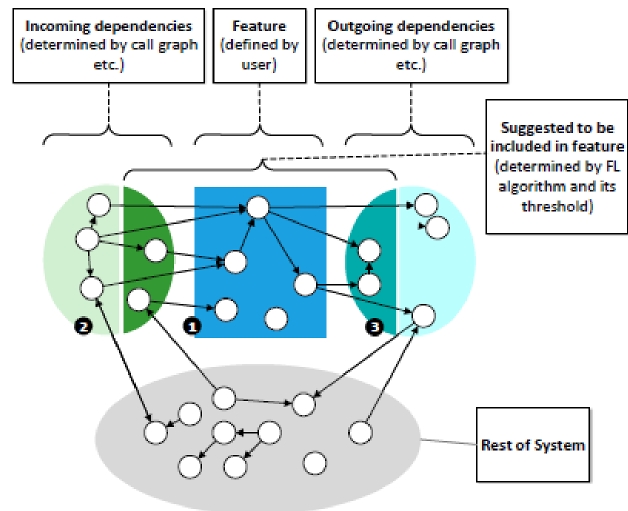


**Figure 1: A Conceptual Representation of the Feature Location Step**

The participants generally expanded their knowledge of the source code location of the feature by moving out, one structural-dependency (for example, data accesses, code invocations) from the source code entities already selected as part of the feature; initially the feature foothold. If, on exploring the code one structural dependency away, they included additional code as relevant to their feature, they would typically repeat this navigation process for the source code entities now associated with the feature. Hence, the tool should ***(b) support recording the source code associated with the feature and (c) make explicit source code one structural dependency away from this source***

***code.*** This iterative-refinement approach is in line with the evaluation literature on Feature Location Techniques (FLTs), which suggests that existing, fully-automated techniques are inappropriate when seeking the full location of a feature in the code-base [9], as they have low effectiveness, in terms of their recall [1]. Consequently, ***(d) any automated FLTs brought in to support the engineers in their feature location with the tool should augment this interactive, iterative refinement and not replace it.***

Participants often had problems when resuming an interrupted task, a frequent occurrence in commercial software development contexts [8]. In this study participants were interrupted with questions from other team members: a normal attribute of a real, working environment [29]. Given the scattering of the associated code over many files/folders these interruptions often caused loss of context among participants [10] and, when resuming their task, they had problems finding the code that had already been processed. This led to redundancy in analysing some entities, wasting time and effort. Sometimes they used a paper notebook or a spreadsheet to track what had been covered, but this still led to errors. Consequently, the tool ***(e) should have mechanisms that allow users to keep track of the code entities that have already been analysed and/or discarded.***

In addition, the vast amount of decisions taken with respect to inclusion of source code in the feature were based on a given artefact's name and/or location, using knowledge of the system's conventions. The participants seldom studied the source code itself. Hence the tool ***(f) should primarily represent the system at the file and folder level of abstraction.***

Finally, given the overall goal of deriving feature-based services, the tool ***(g) should provide support for the generation of an inter-feature model where the inter-feature interfaces can be defined based on the dependencies between the source code associated with each feature.***

## 3. THE FLINTS TOOL

FLINTS (Feature Location and INTerface Specification) is a tool that can be broken up into two conceptual phases. In the first phase, represented by the conceptual model in Figure 1, the objective is to support the developer in locating the source code associated with each feature of the ultimate feature model. The second phase involves analysis of dependencies between the features, towards derivation of their interfaces.

The FLINTS tool is implemented as an Eclipse plug-in. The user interface is presented in Figure 2 where views (1), (2) and (3) align with the corresponding elements of the conceptual model presented in Figure 1.

### 3.1 Phase 1: Feature Location

Most of the views shown in Figure 2 are new contributions to the Eclipse's user interface. The main exception is the *Project Explorer* view (6) which is a standard view that allows the developer a high-level, hierarchical perspective over a system's entire code. It is this view that developers will use to navigate around when manually locating (design principle (a)) source-code entities that can act as initial footholds into the feature, at file or folder level. He/she will drag and drop the identified source code entities onto the graphical view (4), as a node. Nodes in this view represent features and can be named by the developer to reflect that feature.

This is the logical equivalent of moving a source code entity from the rest of the system in Figure 1 to the feature view ❶. Specifically, when the node is selected in the graphical view, it populates FLINTS view (1) with the source code entities (files/folders) that the developer associated with that feature. This in turn populates FLINTS views (2) and (3) with the source code entities that are structurally related to the source code entities in view (1). The entities that depend on entities in view (1) are show in view (2) while the entities that entities in view (1) dependent on are shown in view (3).

The developer can then select source code entities in views (2) and (3) that are part of the active feature, via the pop-up (8). Those entities are assigned to the feature and presented in view (1). The change to the contents of view (1) causes an automatic update to views (2) and (3) making the tool iterative and interactive. These attributes, along with the explicit representation of the contents of the feature, align with design principles (b) and (c).

To supplement this approach, an automated FLTs can be employed to recommend a subset of the code entities in views (2) and (3) as likely candidates for inclusion in the feature. The current realisation of this in the FLINTS tool is an algorithm loosely based on Robillard's proposal [12]. This algorithm analyses the relative strength of the dependencies between the source code entities in views (2) and (3) and the source code entities in view (1), compared to the strength of their dependencies with the rest of the system. Depending on the degree to which their dependencies are more focused on the feature, as specified by sensitivity bar (5), different source code entities in views (2) and (3) will be emboldened (which

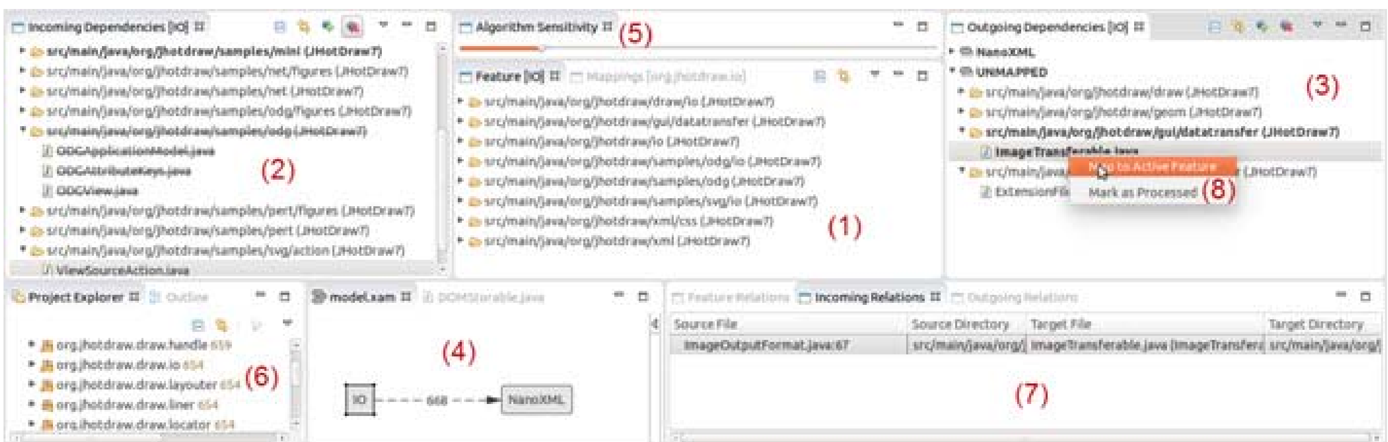


**Figure 2: FLINTS user interface.**

corresponds to the shaded parts of ovals ❷ and ❸ in Figure 1). This aligns the tool with design principle (d).

The other option on pop-up (8) is to mark a source code entity in views (2) and (3) as processed, and this results in the source code entity being crossed-out, as has happened to several of the source code entities in view (2), Figure 2. This is in line with design principle (e): to support engineers keeping track of the files and folders they have visited. As the views presented are at the level of files and folders FLINTS also aligns with design principle (f).

The above process is repeated until there are no unprocessed entities remaining in views (2) and (3) or until the developer is satisfied with their feature definition.

## 3.2 Phase 2: Feature Modeling

The feature modelling capabilities of FLINTS are centred on its graphical view (4). In Figure 2 above, for example, the engineer has identified two features, each represented by a node in this model. Because each feature has been associated with its source code entities, the dependencies between the features can be determined by the tool and this is represented by an automatically generated edge in the model. This edge shows the engineer that there are 668 source code dependencies between the two features. While 668 seems quite large, inspection of the JHotDraw system presented in Figure 2 suggests that many of these dependency instances are calls to, or data accesses to, the same source code entity. As per the Reflexion Modelling approach [15], the engineer can select an edge, and the relation views (7) will present the list of the source code dependencies underpinning that edge.

This is important from two perspectives. Firstly, it may give a further indication as to the quality of the code-to-feature mapping. For example, if code has been mapped to an inappropriate feature a dependency between the feature it should have been mapped to and the inappropriate feature might indicate the poor quality mapping and, if explored, might result in a correction to that mapping.

In addition, the source code dependencies underpinning the edge directed towards the feature provides a proxy for the API that the feature presents to the other feature. Likewise, in exploring the source code dependencies underpinning the edge from a feature to another feature, the engineer gets an indication of that feature's depends-on interface.

This graphical model could also be used to depict the relationships between a feature and the rest of the system (the rest of the system can be modelled in the tool implicitly through the 'UNMAPPED' node in views (2) and (3) or explicitly by creating a Rest-Of-System node). This node could then be used to visually explore the connections/interface that may exist between a feature and the remaining system. Again, the developer could focus in on the interface presented by a feature and the interface it depends on (with respect to the rest of the system). This is particularly useful when the ultimate goal is to derive services from this feature perspective, but it is also important if the goal is to manipulate the system at feature level in general. For example, if an engineer wished to remove a seldom-used feature in their software system, they could identify the dependencies other features had on that feature, thus giving them a better idea of the code that they could realistically remove without breaking the system.

A video demonstration of the FLINTS tool is available at https://youtu.be/olsQZYfCLUk.

# 4. RELATED WORK

## 4.1 Feature Location Approaches

Automated Feature Location techniques can be broadly categorized as static, dynamic, textual and hybrid, where 'hybrid' refers to a combination of several other approaches together [1]. Static analysis involves examining structural information (such as control or data-flow dependencies) in code without executing it [1]. Hence, often one of the first steps in static FLTs is to construct a dependence graph by extracting information (i.e. source code entities and their relationships) from the source code [17][18][13][19][20]. Static analysis then typically involves processing or traversing the constructed dependence graph to identify source code entities relevant to a given set of software artefacts [1][21]. The approach presented here uses static analysis primarily, in support of software engineers.

Textual FLTs consider source code as textual documents with identifiers and comments considered as meaningful lexicons. Dynamic analysis involves executing the system and collecting trace information as it runs. Empirical evaluations have shown that neither are suitable in isolation for identifying the complete location of features in source code [1]: dynamic FLTs because only a subset of all possible executions can be realistically traced by such techniques; textual FLTs because there is a paucity of lexicons in source code and they are not always application-domain oriented. Interestingly the initial empirical study carried out (Section 2) suggested that the software engineers seldom reverted to a dynamic strategy, although they did use textual searches.

Peng et al. [11] observed that existing FL techniques generally perform a one-time analysis with the initial query input and, therefore, these techniques are sensitive to the quality of the input. They propose an iterative FL approach like ours to mitigate this limitation. Kastner et al. [9] showed that such approaches can yield high recall and precision. In another example, Petrenko and Rajlich [21] propose a FLT called Dependency Search which combines static and Information Retrieval (IR) based textual strategies. They use IR for ranking methods first, essentially based on their textual similarity to an iteratively-formed feature query. Then they allow developers to select methods from the resultant ranked list to further explore their structural dependencies. In a similar vein, [23] and [24] use a combination of IR and structural dependencies analysis. Our approach is exclusively static, although FLINTS' ability to incorporate other FLTs allows for the inclusion of textual, IR techniques going forward.

## 4.2 Tools in the Feature Location/Modelling Space

FLINTS has its Feature Location origins in the Dependency Browser proposed by Liu et al. [25] and its Feature Modelling origins in the proposal for a tool called FORM, by Herold and Buckley [22]. The Dependency Browser is an Eclipse plugin that supports dependency searches on Java applications. It provides a similar interface to FLINTS showing, via a list-like interface, the code dependent on a specified piece of code. However, it only allows users to view the dependencies of a single source code entity. FLINTS allows viewing of dependencies for a set of source code entities. The FORM tool suggested by Herold and Buckley, proposed the application of Reflexion Modelling [22] to the task of Feature modelling, leveraging the dependency insights that Reflexion Modelling provides to refine the feature's source code location and to help define the feature's interface. This aligns with the graphical representation provided by FLINTS.

Robillard et al. present FEAT [19] and ConcernMapper [20], which are similar to the tool presented here. Using FEAT, different features can be associated with underlying source code and represented in an abstract model called Concern Graphs. The features' relationships with each other can be probed, but less holistically than with the graphical view of FLINTS. Likewise, while ConcernMapper presents a textual-tree view of features hierarchies, the relationships between features that are not in the same hierarchy are obscured.

Building on this previous work [12], Warr and Robillard [13] present a tool called Suade to assist developers in software investigation tasks. Given some starting footholds such as fields and methods, Suade presents users with a ranked list of source code entities related to the footholds. Suade achieves this by analysing the topology of structural dependencies in the software and ranking scores according to the algorithm specified in [12].

Chen and Rajlich [17] present a semi-automatic tool called 'Ripples' to support both concept location and change propagation in legacy software written in C. Likewise, Buckner et al. [18] present a tool called JRipples, developed with similar intentions but for Java. Ripples facilitates developers visually traversing an Abstract System Dependence Graph (ASDG) constructed from a software system. The developers can annotate nodes and edges of the extracted ASDG using some annotation markers (such as candidate, visited) while traversing the graph top-down or bottom-up. FLINTS does not support explicit annotation of source code entities in this fashion but offers intuitive UI options to achieve desirable functionality like the ability to hide unrelated source code entities.

Although FEAT [19], ConcernMapper [20], Ripples [17], JRipples [17], Suade [13] and our tool share many similar characteristics, the architecture and emphasis of our tool differs significantly. Firstly, FLINTS allows the integration of different parsers, to populate a Language Independent Repository (LIR). This allows for the analysis of systems with a heterogeneous implementation in multiple (programming) languages. Moreover, it also allows different FLTs to be plugged-in (although only one is currently incorporated) again through the LIR: The other tools reviewed here integrate tightly coupled parsers with FLTs lessening this flexibility, although efforts have been made to make ConcernMapper extensible in terms of the FLTs it can employ [20].

In terms of 'emphasis', the FLINTS UI presents a graphical view of the inter-function relationships, regardless of their hierarchical structure, allowing for a holistic analysis of several (seemingly) unrelated features and their dependencies. The other tools described here [18] [17] [19] [13] [20] display either a graphical view or a list of source code entities, providing little insight as to the potential interfaces of the features.

While many of these tools work at the method and field level, it should be noted that currently, FLINTS works at the class/file/folder level. This is in line with our original observations/design principles, and the scale of our collaborator's software systems. But, based on developers' initial reviews of our prototype FLINTS tool, it is intended to move to finer levels of granularity, particularly with respect to data-field access.

Finally, it is worth noting that the primary evaluation focus for most of these tools is on their performance (precision, recall, memory usage, and speed) [17] [18] [20] [23] [21] [24]. Only a handful pay attention to developer-focused issues [9] [3], neglecting the utility of the tool to the developer, as we aim to do with FLINTS.

# 5. FUTURE DIRECTION

A number of avenues will be pursued in this work:

- Evaluation of FLINTS: FLINTS is currently being evaluated, in-vivo, by our commercial partner, who is interested in migrating their monolithic, flag-ship system to services. The first batch of evaluations has been carried out and has focussed on its ability to locate specific features correctly in the code-base. This evaluation was in the form of five case studies where five features were sought, one by each of five different, experienced developers of the system. In addition, a longitudinal study is underway where the tool will reside on seven developers' desktops and where they note the difficulties that they experience with the tool in a log-book. As features are located by the individual developers in the code-base a cumulative feature model will be generated, and be presented back to the company in order to help them derive a set of services from their system and to perform various feature-driven maintenance tasks. This model will also be assessed for utility in these roles and the design of this study has begun.
- A number of enhancements are envisaged for the tool. As per other offerings like ConcernMapper [20] and FEAT [19] and, as per the initial feedback obtained from developers when evaluating the tool, fine level dependency analysis will be carried out. In particular, the developers expressed a strong desire to see data dependencies at the data-field level.
- Likewise, FLINTS has been designed to allow for the incorporation of several different FLTs. Ultimately, we plan a suite of supportive FLTs for the tool, but initially a text-based FLT will be incorporated in line with Revelle and Poshyvanyk's approach [26]. This will provide a counterpoint to the static FLT analysis already in place and it will be interesting to see why and when experienced, professional developers choose one FLT over the other.
- The feature model will be enriched. For example, ports [27] may be defined to refine the model's ability to guide the developer to feature interfaces. Likewise additional algorithms may be developed to prune the actual dependency instances between features to a more manageable, defined interface.

# 6. CONCLUSIONS

This paper has presented FLINTS, a tool for semi-automated, static-analysis-based FL and Reflexion-based, architectural-level, inter-feature modeling. The first prototype of the tool is currently under evaluation in our commercial partner and the initial feedback is good. The company has asked for the tool on seven developers' desktops and FLINTS is being used by two professional software developers already, on an on-going basis.

However, much work remains to be done. The evaluation of the architectural modeling aspect of the tool has still to be carried out and both FL and feature modeling aspects need to be enhanced. This will be done as part of our ongoing, iterative Action Research [28] program with our industrial collaborator.

## 7. ACKNOWLEDGMENTS

This work was supported with the financial support of the Science Foundation Ireland grant 13/RC/2094 and co-funded under the European Regional Development Fund through the Southern & Eastern Regional Operational Programme to Lero – the Irish Software Research Centre (www.lero.ie).

## 8. REFERENCES

[1] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. "Feature location in source code: a taxonomy and survey," Journal of Software: Evolution and Process, vol. 25, no. 1, pp. 53–95, 2013. DOI= http://dx.doi.org/10.1002/smr.567.

[2] J. Rubin, and M. Chechik. A Survey of Feature Location Techniques. In I. Reinhartz-Berger, A. Sturm, T. Clark, S. Cohen and J. Bettin, eds., Domain Engineering. Springer Berlin Heidelberg, 2013, 29–58.

[3] H.R. Jordan, J. Rosik, S. Herold, G. Botterweck, and J. Buckley, "Manually locating features in industrial source code: the search actions of software nomads," in Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension, ICPC 2015, Florence/Firenze, Italy, May 16-24, 2015, A. D. Lucia, C. Bird, and R. Oliveto, Eds. ACM, 2015, pp. 174–177. DOI= http://dx.doi.org/10.1109/ICPC.2015.26.

[4] M. Desmond, M.A. Storey, and C. Exton. Fluid Source Code Views for Just In Time Comprehension. *SPLAT 2006*, (2006).

[5] K. A. Ericsson and H. A. Simon, "Verbal reports as data." Psychological Review, vol. 87, no. 3, p. 215, 1980.

[6] Techsmith — Camtasia, screen recorder and video editor. Accessed: 2016-03-29. DOI= https://www.techsmith.com/camtasia.html.

[7] A. Strauss, J. Corbin et al., Basics of qualitative research. Newbury Park, CA: Sage, 1990, vol. 15.

[8] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: a study of developer work habits," in 28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006, L. J. Osterweil, H. D. Rombach, and M. L. Soffa, Eds. ACM, 2006, pp. 492–501. DOI= http://doi.acm.org/10.1145/1134355.

[9] C. Kästner, A. Dreiling, and K. Ostermann, "Variability mining: Consistent semi-automatic detection of product-line features," IEEE Trans. Software Eng., vol. 40, no. 1, pp. 67–82, 2014. DOI= http://dx.doi.org/10.1109/TSE.2013.45.

[10] M. Desmond, M.-A. Storey, and C. Exton, "Fluid source code views for just in-time comprehension," 2006. DOI = https://ulir.ul.ie/handle/10344/1793.

[11] X. Peng, Z. Xing, X. Tan, Y. Yu, and W. Zhao, "Iterative contextaware feature location," in Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu, HI, USA, May 21-28, 2011, R. N. Taylor, H. C. Gall, and N. Medvidovic, Eds. ACM, 2011, pp. 900–903. DOI= http://doi.acm.org/10.1145/1985793.1985939.

[12] M. P. Robillard, "Automatic generation of suggestions for program investigation," in Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005, M. Wermelinger and H. C. Gall, Eds. ACM, 2005, pp. 11–20. DOI= http://doi.acm.org/10.1145/1081706.1081711.

[13] F. W. Warr and M. P. Robillard, "Suade: Topology-based searches for software investigation," in 29th International Conference on Software Engineering (ICSE 2007). IEEE Computer Society, 2007, pp. 780–783. DOI= http://dx.doi.org/10.1109/ICSE.2007.80.

[14] 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 (2013), 1193–1224.

[15] G. C. Murphy, D. Notkin, and K. J. Sullivan, "Software reflexion models: Bridging the gap between source and high-level models," in SIGSOFT '95, Proceedings of the Third ACM SIGSOFT Symposium on Foundations of Software Engineering, Washington, DC, USA, October 10-13, 1995, G. E. Kaiser, Ed. ACM, 1995, pp. 18–28. DOI= http://doi.acm.org/10.1145/222124.222136.

[16] M. Mitchell and J. Jolley, Research Design Explained. Thomson Wadsworth, 2004. Available at: https://books.google.ie/books?id=67juAAAAMAAJ.

[17] K. Chen and V. Rajlich, "RIPPLES: tool for change in legacy software," in 2001 International Conference on Software Maintenance, ICSM 2001, Florence, Italy, November 6-10, 2001. IEEE Computer Society, 2001, pp. 230–239. DOI= http://dx.doi.org/10.1109/ICSM.2001.972736

[18] J. Buckner, J. Buchta, M. Petrenko, and V. Rajlich, "Jripples: A tool for program comprehension during incremental change," in 13th International Workshop on Program Comprehension (IWPC 2005), 15-16 May 2005, St. Louis, MO, USA. IEEE Computer Society, 2005, pp. 149–152. DOI= http://dx.doi.org/10.1109/WPC.2005.22.

[19] M. P. Robillard and G. C. Murphy, "FEAT. A tool for locating, describing, and analyzing concerns in source code," in Proceedings of the 25th International Conference on Software Engineering, May 3-10, 2003, Portland, Oregon, USA, L. A. Clarke, L. Dillon, and W. F. Tichy, Eds. IEEE Computer Society, 2003, pp. 822–823.

[20] M. P. Robillard and F. W. Warr, "Concernmapper: simple view-based separation of scattered concerns," in Proceedings of the 2005 OOPSLA workshop on Eclipse Technology eXchange, ETX 2005, San Diego, California, USA, October 16-17, 2005, M. D. Storey, M. G. Burke, L. Cheng, and A. van der Hoek, Eds. ACM, 2005, pp. 65–69. DOI= http://doi.acm.org/10.1145/1117696.1117710.

[21] M. Petrenko and V. Rajlich, "Concept location using program dependencies and information retrieval (depir)," Information & Software Technology, vol. 55, no. 4, pp. 651–659, 2013. DOI= http://dx.doi.org/10.1016/j.infsof.2012.09.013.

[22] S. Herold. and J. Buckley. "Feature Oriented Reflexion Modelling" in the 2nd SAEroCon Workshop 2015. DOI= 10.1145/2797433.2797494.

[23] K. Nie and L. Zhang, "Software feature location based on topic models," in 19th Asia-Pacific Software Engineering Conference, APSEC 2012, Hong Kong, China, December 4-7, 2012, K. R. P. H. Leung and P. Muenchaisri, Eds. IEEE,

2012, pp. 547–552. DOI= http://dx.doi.org/10.1109/APSEC.2012.116.

[24] G. Scanniello and A. Marcus, “Clustering support for static concept location in source code,” in The 19th IEEE International Conference on Program Comprehension, ICPC 2011, Kingston, ON, Canada, June 22-24, 2011. IEEE Computer Society, 2011, pp. 1–10. DOI= http://dx.doi.org/10.1109/ICPC.2011.13.

[25] D. Liu, S. Xu, and Z. Liu, “An eclipse plug-in: Dependency browser,” in 2007 IEEE/ACS International Conference on Computer Systems and Applications (AICCSA 2007), 13-16 May 2007, Amman, Jordan. IEEE Computer Society, 2007, pp. 40–46. DOI= http://dx.doi.org/10.1109/AICCSA.2007.370862.

[26] M. Revelle and D. Poshyvanyk. "An Exploratory Study on Assessing Feature Location Techniques" in the Proc. of ICPC, The International Conference on Program Comprehension, May 2009. pp 218-222. DOI= 10.1109/ICPC.2009.5090045.

[27] I. Gorton. "Essential Software Architecture" 2nd Edition. Springer, 2011. ISBN: 978-3642191756

[28] M.N. Saunders, M. Saunders, P. Lewis, & A. Thornhill. Research methods for business students, 2011. 5/e. Pearson Education India.

[29] A.J. Ko, R. DeLine. and G. Venolia. “Information Needs in Collocated Software Development Teams” International Conference on Software Engineering (ICSE 2007), IEEE Computer Society. pp. 344-353. DOI= 10.1109/ICSE.2007.45