Modeling Features at Runtime

Marcus Denker, Jorge Ressia, Orla Greevy, Oscar Nierstrasz

Modeling Features at Runtime ⋆

Marcus Denker[1], Jorge Ressia[2], Orla Greevy[3], Oscar Nierstrasz[2]

[1]INRIA Lille Nord Europe - CNRS UMR 8022 - University of Lille (USTL
http://rmod.lille.inria.fr
[2]Software Composition Group, University of Bern, Switzerland
http://scg.unibe.ch
[3]Sw-eng. Software Engineering GmbH Berne, Switzerland
http://www.sw-eng.ch

Abstract. A feature represents a functional requirement fulfilled by a system. Since many maintenance tasks are expressed in terms of features, it is important to establish the correspondence between a feature and its implementation in source code. Traditional approaches to establish this correspondence exercise features to generate a trace of runtime events, which is then processed by post-mortem analysis. These approaches typically generate large amounts of data to analyze. Due to their static nature, these approaches do not support incremental and interactive analysis of features. We propose a radically different approach called *live feature analysis*, which provides a model at runtime of features. Our approach analyzes features on a running system and also makes it possible to "grow" feature representations by exercising different scenarios of the same feature, and identifies execution elements even to the sub-method level.

We describe how live feature analysis is implemented effectively by annotating structural representations of code based on abstract syntax trees. We illustrate our live analysis with a case study where we achieve a more complete feature representation by exercising and merging variants of feature behavior and demonstrate the efficiency or our technique with benchmarks.

Keywords: models at runtime, behavioral reflection, feature annotations, dynamic analysis, feature analysis, software maintenance, feature growing

1 Introduction

Many researchers have recognized the importance of centering reverse engineering activities around a system's behavior, in particular, around features [8,15,22]. Bugs and change requests are usually expressed in terms of a system's features, thus knowledge of a system's features is particularly useful for maintenance [17].

⋆ Proceedings of MODELS 2010, Springer LNCS, to appear.

This paper presents a novel technique to perform fine-grained feature analysis incrementally at runtime, while minimizing adverse effects to system performance. For an in-depth discussion about features and feature analysis in general, we refer the reader to our earlier publications [10–12].

Features are abstract notions, normally not explicitly represented in source code or elsewhere in the system. Therefore, to leverage feature information, we need to perform feature analysis to establish which portions of source code implements a particular feature. Most existing feature analysis approaches [15, 22] capture traces of method events that occur while exercising a feature and subsequently perform post-mortem analysis on the resulting feature traces.

A post-mortem feature analysis implies a level of indirection from a running system. This makes it more difficult to correlate features and the relevant parts of a running system. We lose the advantage of interactive, immediate feedback which we would obtain by directly observing the effects of exercising a feature. Post-mortem analysis does not exploit the implicit knowledge of a user performing acceptance testing. Certain subtleties are often only communicated to the system developer when the user experiences how the system works while exercising the features. Clearly, in this case, a model-at-runtime of features, with the added ability to "grow" the feature representation as the user exercises variants of the same feature offers advantages of context and comprehension over a one-off capture of a feature representation and post-mortem analysis.

We propose an approach which we call *live feature analysis*. Essentially, our technique is to annotate the structural representation of the source code based on *Abstract Syntax Trees* (ASTs). We build up a model of the features over time, as variants of feature behavior are being exercised. Our feature representation is thus immediately available for analysis in the context of the running system. The goal of this paper is to highlight the advantages of using a runtime model of feature over a static, post-mortem feature representation.

Live feature analysis addresses various shortcomings of traditional post-mortem approaches:

- *Data volume.* By tracking features in terms of structured objects rather than linear traces, the volume of data produced while exercising features is constant with respect to the number of source code entities. Data produced by traditional feature analysis approaches is proportional to the execution events of the source code entities, thus higher. Different strategies to deal with large amounts of data have been proposed, for example: (1) summarization through metrics [7], (2) filtering and clustering techniques [13, 26], (3) visualization [2, 12] (4) selective instrumentation and (5) query-based approaches [19]. The net effect of applying these strategies however, is loss of information of a feature's behavior in the feature representation to be analyzed.
- *Feature growing.* Variants of the same feature can be exercised iteratively and incrementally, thus allowing the analysis representation of a feature to "grow" within the development environment. The problems of tracing at sub-method granularity are performance and large amount of gathered data. Traditional

approaches deliver a tree of execution events which are hard to manipulate. Our technique delivers a set of source code entities on top of which any logical operation can be applied.
- *Sub-method feature analysis.* Many feature analysis techniques are restricted to the level of methods. Our technique allows us to dive into the structure of more complex methods to see which branches are dedicated to particular features. The fine-grained sub-method information is particularly relevant when "growing" features as it is typically at this level that variations in the execution paths occur.
- *Interactive feature analysis.* Traditional feature analysis approaches perform a post-mortem analysis on the trace data gathered at runtime to correlate feature information with the relevant parts of the source code. Any changes in the source code will require the data gathering and the post-mortem analysis to be run again to refresh the feature data. The essential drawback of the post-mortem approach is due to its static, snapshot representation of a feature, limited to data captured during only one path of execution. Post-mortem analysis is incompatible with an interactive exploration of features as it is typically disconnected from the running application. With traditional feature analysis approaches the feature information only becomes available after the features of the application have been exercised and a post-mortem analysis has been performed.

The key contributions of this paper are:

1. We describe our live feature analysis approach based on partial behavioral reflection and AST annotation;
2. we show how this technique supports feature analysis at sub-method granularity;
3. we describe how to iteratively and incrementally grow feature analysis using our technique; and
4. we demonstrate the advantages of using a runtime model of features.

We previously proposed the notion of using sub-method reflection in the context of feature tagging in a workshop paper [6]. In this paper, we expand on this idea to focus on its application to runtime feature analysis (*live feature analysis*). We present our working implementation and validate our technique by applying it to the analysis of the features of a content-management system, Pier, as a case-study.

In the next section, we provide a brief overview of *unanticipated partial behavioral reflection*, as it serves as a basis for our approach. We introduce our feature annotation mechanism and present the feature runtime model. Using the feature annotation and the runtime model we obtain the flexibility to grow feature representations incrementally. We validate our approach in Section 3 by means of a case study and detailed benchmarks. Section 4 outlines related work in the fields of dynamic analysis and feature identification. Finally, we conclude in Section 5 by outlining possible future work in this direction.

2 Live Feature Analysis with Partial Behavioral Reflection

To achieve *live feature analysis*, we need a means to directly access runtime information associated with features and at the same time minimize the negative impact of slowing down the running application under investigation due to instrumentation. In a previous work [5] we proposed that *Partial Behavioral Reflection* as pioneered by Reflex [23] is particularly well-suited for dynamic analysis as it supports a highly selective means to specify where and when to analyze the dynamics of a system. Before explaining *Partial Behavioral Reflection* in more detail, we present a brief review of Structural Reflection and Behavioral Reflection.

Structural reflection models the static structure of the system. Classes and methods are represented as objects and can both be read and manipulated from within the running system. In today's object-oriented systems, structural reflection does not extend beyond the granularity of the method: a method is an object, but the operations the method contains, such as method invocations, variable reads and assignments, are not modeled as objects. To overcome this limitation we have extended Pharo Smalltalk[1] to support sub-method structural reflection. More in-depth information about this system and its implementation can be found in the paper on sub-method reflection [4].

Behavioral reflection provides a way to intercept and change the execution of a program. It is concerned with execution events, *i.e.,* method execution, message sends, or variable assignments. We developed a framework called Reflectivity [3], which leverages an extended AST representation provided by our sub-method structural reflection extension to realize behavioral reflection. Prior to execution, the AST is compiled on demand to a low-level representation that is executable, for example to bytecodes executable by a virtual machine.

Partial behavioral reflection offers an even more flexible approach than pure Behavioral Reflection. The key advantage is that it provides a means to selectively trigger reflection, only when specific, predefined events of interest occur. The core concept of the Reflex model of partial behavioral reflection is the *link* (see Figure 1). A link sends messages to a meta object upon occurrences of marked operations. The attributes of a link enable further control of the exact message to be sent to the meta-object. Furthermore, an activation condition can be defined for a link which determines whether or not the link is actually triggered.

Links are associated with nodes from the AST. Subsequently, the system automatically generates new bytecode that takes the link into consideration the next time the method is executed.

Reflectivity was conceived as an extension of the Reflex model of *Partial Behavioral Reflection* [23]. Reflex was originally realized with Java. Therefore, our approach can in be implemented in a more static mainstream language like Java. The reason for choosing Smalltalk and Reflectivity for this work is

[1] `http://pharo-project.org/`

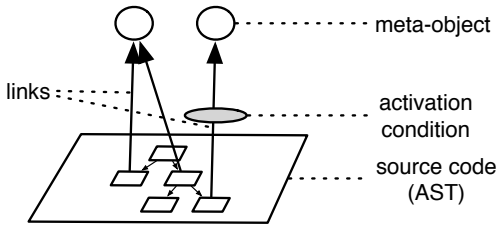

Fig. 1. The reflex model

that it supports *unanticipated* use of reflection at runtime [3] and is integrated with an AST based reflective code model [4]. A Java solution would likely be more static in nature: links cannot be removed completely (as code cannot be changed at runtime) and the code model would not be as closely integrated with the runtime of the language.

2.1 Dynamic Analysis with Reflectivity

A trace-based feature analysis can be implemented easily using partial behavioral reflection. In a standard trace-based feature analysis approach, a tracer is the object responsible for recording a feature trace of method invocations. With our partial behavioral reflection approach, we define the tracer as the meta-object (as shown in Figure 2). We define a link to pass the desired information for feature representation (*e.g.,* the name and class of the executed method) as a parameter to a meta-object. The link then is installed on the part of the system that we want to analyze. This means that the link is associated with the elements of the system that we are interested in. Subsequently, when we exercise a feature and our tracer meta-object will record a trace.

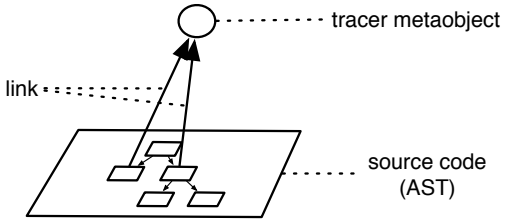

Fig. 2. A tracer realized with partial behavioral reflection.

The resulting system is similar to existing trace-based systems, but with one key advantage: the scope of tracing can now easily be extended to cover sub-method elements of the execution, if required.

2.2 Feature Annotation

In contrast to typical dynamic feature analysis approaches, our reflection-based approach does not need to retain a large trace of executed data. This is because our analysis is live rather than post-mortem. Our technique focuses on exploiting feature knowledge directly while the system is running. With the annotatable representation provided by sub-method reflection, our analysis can annotate every statement that participates in the behavior of a feature. So instead of recording traces, the analysis tags with a feature annotation all the AST nodes that are executed as a result of invoking features at runtime. To achieve this, we define a link to call our *FeatureTagger* meta-object, as shown in Figure 3.

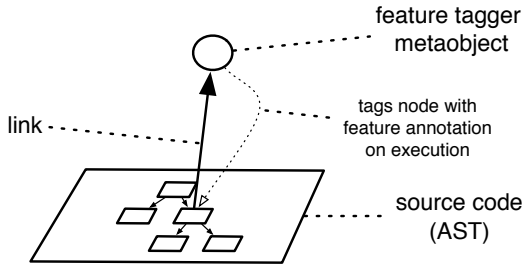

Fig. 3. The nodes are tagged at runtime with feature information.

Subsequently, we install this link on all the AST nodes of the system that we plan to analyze. When a feature is exercised, different AST nodes (method, instructions, assignments, etc.) linked to the FeatureTagger meta-object are executed. This meta-object is executed as well and each executed node is annotated stating that this node belongs to a specific feature. Due to tagging we no longer need to retain large traces. Thus our approach results in less data to be managed and analyzed (see Section 3). Information about multiple executions of the same methods or order of execution can be efficiently stored in the annotation.

2.3 Model-at-Runtime

In Figure 4 we can see the runtime feature model on top of the structural representation of the language. The Feature abstraction is used for annotating which parts of the AST belongs to a feature. Multiple features can be attached to a single AST node providing a way of specifying that this node belongs to several features.

Since the user or developer has the implicit feature knowledge, he must specify which feature is going to be exercised. The activeFeature in the class RuntimeFeatureAnalyzer models this. This feature is used for annotating the AST nodes. If no feature is specified no annotation will occur.

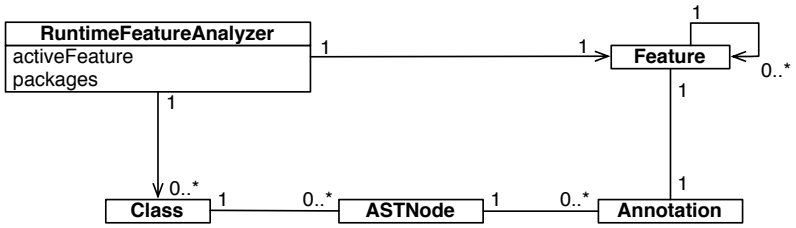

Fig. 4. Feature runtime model

The RuntimeFeatureAnalyzer is also responsible for adapting the AST nodes. The user has to specify over which nodes he would like to perform the feature analysis. At present we offer the possibility to specify which packages should be adapted, but any AST granularity could be used. All the classes, methods and AST nodes of the specified packages will be adapted. New nodes or packages could be added at any point due to new development being carried out. The link installed in the AST nodes specifies that whenever the node is executed the same node has to be annotated with the activeFeature specified in the RuntimeFeatureAnalyzer. This information is immediately available for the different development tools, like the code browser or the debugger.

Performance is a major consideration when performing dynamic analysis. To minimize adverse effects on performance, our goal is therefore to limit both where and when we apply behavioral reflection. The *where* can be controlled by installing the link only on those places in the system that we are interested in analyzing. For the *when*, we leverage the fact that we can directly uninstall links at runtime from the meta-object. The RuntimeFeatureAnalyzer takes care of removing the links when there is no active feature.

2.4 Growing Features

Our feature annotation analysis can easily support many existing feature analysis techniques. For example, we could exercise a feature multiple times with different parameters to obtain multiple paths of execution. This can be important, as the number of traces obtained can be considerable depending on the input data. For trace-based approaches this results in multiple traces being recorded. One feature is represented by multiple traces and therefore it is needed to manage a many-to-one mapping between features and traces. Using our approach, if the execution path differs over multiple runs, newly executed instructions will be tagged in addition to those already tagged. Thus we can use our approach to iteratively build up the representation of a feature covering multiple paths of execution.

2.5 Towards Optional Tracing

Instead of multiple runs resulting in one feature annotation, the feature annotations can be parameterized with the number of executions that are the result

of exercising the feature. Our approach also accommodates capturing instance information or feature dependencies as described in the approaches of Salah *et al.* [22] and Lienhard *et al.* [16]. Naturally, the more information that is gathered at runtime, the more memory that is required. In the worst case, recording everything would result in recording the same amount of information as a complete trace of fine-grained behavioral information. The execution cost will also be higher due to the inserted code of the behavioral reflection technique.

One major disadvantage when adopting filtering strategies to reduce the amount of data gathered at runtime is the loss of dynamic information that may have been relevant for the analysis. Thus it is crucial to define which information is necessary when performing a feature analysis. A change in a selection strategy implies a need to exercise a feature again. In contrast to filtered traces, analysis approaches based on complete traces provide the flexibility to perform a variety of post-mortem analyses of feature traces, where each analysis focuses on a different level of detail. For example one analysis may choose to focus on the core application code and filtering out library code, whereas another analysis would choose to home in on the use of library code by a feature.

With our approach we can extend feature annotation gradually. Instead of uninstalling the tagger meta-object after the first execution, we can use it to gather an invocation count. This approach would be useful for building so-called feature heat maps *i.e.,* visualizations that show how often a part of the system (*i.e.,* a specific node) takes part in a feature [21]. Even adding optional support for recording full traces can be interesting: the node can reference the trace events directly, providing an interesting data-structure for advanced trace analysis.

3 Validation

We present the feature analysis experiment we performed to highlight the difference between the amount of data gathered with a trace-based approach and the annotation based approach. Our experiment was performed using a medium-sized software system of approximately 200 classes, a content-management system (CMS) Pier, which also encompasses Wiki functionality [18] as a case-study.

For our experiment we selected a set of Pier features similar to those chosen for a previous post-mortem experiment (login, edit page, remove page, save page) with the same case study [9] In this paper, we illustrate the application of our analysis technique on the *login* feature.

3.1 Case Study: The Pier CMS

We apply our live feature analysis to show some empirical results on an example feature of the Pier CMS. We chose the *login* feature as it is easy to explain the concept of growing a feature as a result of exercising variants of feature behavior. When exercising a login feature, different behavior is invoked when a user enters a correct username and password, as opposed to when incorrect information is entered and an error message is displayed. Our first execution scenario of our

experiment expresses when the username and password are entered correctly and the user successfully logs into the system. Variants of execution behaviors are:

(i) no information is entered and the user presses login,
(ii) only the username is entered,
(iii) only the password is entered,
(iv) an invalid username or password is entered.

3.2 Live Analysis

	Individual features	Feature growing
(i) Empty username and password	1668	1668
(ii) Only username	1749	1749
(iii) Only password	1668	1749
(iv) Valid username and password	2079	2126

Table 1. Number of annotations with features exercised individually, and with feature growing.

In the first column of Table 1 we can see the results of applying sub-method feature analysis to the login feature. For the different possible input combinations we obtain different numbers of annotated AST nodes. This reveals that different parts of the code inside the methods, namely sub-method elements, are being executed. We can also see that when no username is provided, case (i) and case (iii), the number of nodes associated to the feature is the same. One possible explanation for this is that a validation is performed whether or not the username is valid before dealing with the password. We presented our results to the developer of the application who verified our findings.

3.3 Growing Features

One of the key characteristics of our technique is the ability to grow feature representation over multiple executions, thus achieving a more complete analysis of the feature under consideration. By taking into account all possible execution paths of a particular feature from the user's perspective, we increase the precision of our dynamic analysis approach. It is possible to achieve the same feature coverage with a classical post-mortem feature analysis by gathering the data at the end of each run and then by merging this data into one representation of a feature. The advantage of our approach is that it exploits an exploratory analysis that is both iterative and interactive. It benefits from immediate feedback and direct access to growing feature representation, as different scenarios are executed and different code fragments (*i.e.,* methods, branches and variable accesses) are highlighted as belonging to the current feature under analysis.

Considering the CMS login feature of our Pier case study, we grow our representation of our login feature without interrupting the dynamic session to perform offline, post-mortem analysis. We simply exercise different variants of

behaviors, while our technique builds up the feature representation. In the second column of Table 1 we illustrate how our login feature grows. The different behavioral variants were tested in the same order as presented in the table. In case (iii) we see that in fact no new parts of the code were executed for this variant as case (iii) is in fact contained in case (i). And finally for case (iv) the number of annotations is higher than the regular analysis since we are keeping the annotations for the code of all possible paths. An important aspect of our technique is that the entire live feature analysis can be performed at runtime and that there is no need to interrupt a system under analysis to access the feature representation information. This is a key advantage over traditional techniques as it is not always possible to interrupt production systems to gather information about malfunctioning features.

3.4 Benchmarks

Performance is critical for the practical application of any runtime analysis technique. A key characteristic of our live feature analysis approach is that we can uninstall the trace code (the links) at runtime as soon as possible to avoid a long term adverse effect on performance of the system under analysis. We have carried out a benchmark to test our hypothesis that uninstalling links at runtime actually improves performance. We take advantage of the extensive test-suite of Pier to generate a substantial amount of runtime data. We install our feature annotation system on all the Pier classes in the package Pier (179 classes, 1292 methods, *i.e.,* not considering library classes). Subsequently, we performed our live feature analysis with three different annotation scenarios:

- **No feature annotation**. We execute the unit tests without feature annotation installed as the base case.
- **Feature annotation removed.** Feature tagging is enabled and the link is removed immediately after setting the annotation on the node.
- **Feature annotation retained.** To perform this scenario, we use a modified version of our feature tagging code where the link is not removed.

We ran the unit tests of Pier three times (see Table 2) for each of the scenarios (run1, run2 and run3 show the results for each run).[2]

Our results demonstrate that uninstalling the link improves performance, despite the fact that in this case the bytecode needs to be regenerated for all the methods where links are removed. This is shown in the time difference between run2 and run3: the second time the code has to be recompiled without the link and the third time it is executed normally.

Removing feature tagging once the feature annotation has been performed delivers a zero penalty after the third execution due to dynamic recompilation of code. However, keeping the feature tagging enabled has a penalty of 16 times slower. But this negative impact is only perceived in the nodes that have been annotated. The rest of the system has no performance penalty.

[2] The benchmarks were performed on a MacBook Pro Core2Duo 2.4Ghz.

	run1 (msecs)	run2 (msecs)	run3 (msecs)
No feature annotation	680	667	637
Feature annotation removed	34115	2344	649
Feature annotation retained	38584	13067	10920

Table 2. Performance of annotation – three runs.

3.5 Number of Events Generated

Our paper on sub-method reflection [4] discusses the memory properties of the AST model. In the context of live feature analysis, it is interesting to assess the difference in space required to annotate static structures by live feature analysis as opposed to that required to record traces in a postmortem approach. We compare the number of events generated in each case. To measure the size of a trace, we install a counter mechanism that records method invocations that are activated while exercising our example features. When we annotate features, the result are annotations on the static structure. Therefore, we count the resulting annotations.

Our benchmarking was performed once again on the Pier case study. We annotate the entire Pier package. Table 3 shows the numbers of events and the number of annotations required for different features. Our results show that the number of dynamic events that a Tracer records is far higher than the resulting entities annotated with feature annotation.

Feature	Number of events	Number of annotations	Factor
Display Page	24792	1956	12.67
Call Page Editor	28553	2348	12.16
Save Page	41737	3195	13.06
Page Settings	17709	1879	9.42

Table 3. Dynamic events compared to number of annotations.

Depending on the feature exercised our approach generates up to 13 times less data for representing the same information compared to a traditional approach.

4 Related Work

We review dynamic analysis system comprehension and feature identification approaches and discuss the similarities and shortcomings of these approaches in the context of our live feature analysis technique.

Many approaches to dynamic analysis focus on the problem of tackling the large volume of data. Many of these works propose compression and summarization techniques to support the extraction of high level views [10, 14, 26]. Our technique is in effect also a summarization technique in that the annotation does not need to be repeated for multiple executions of the same parts of the code. However, our annotation can easily encompass information about number of executions of a mode in the code representation and also order of execution, so that the dynamic information is efficiently saved in a compressed form and would be expandable without loss of information on demand.

Dynamic analysis approaches to feature identification have typically involved executing the features of a system and analyzing the resulting execution trace [1, 8, 24, 25]. Typically, the research effort of these works focuses on the underlying mechanisms used to locate features (*e.g.,* static analysis, dynamic analysis, formal concept analysis, semantic analysis or approaches that combine two or more of these techniques).

Wilde and Scully pioneered the use of dynamic analysis to locate features [24]. They named their technique *Software Reconnaissance*. Their goal was to support programmers when they modify or extend functionality of legacy systems.

Eisenbarth *et al.* described a semi-automatic feature identification technique which used a combination of dynamic analysis, static analysis of dependency graphs, and formal concept analysis to identify which parts of source code contribute to feature behavior [8]. For the dynamic analysis part of their approach, they extended the Software Reconnaissance approach to consider a set of features rather than one feature. They applied formal concept analysis to derive a correspondence between features and code. They used the information gained by formal concept analysis to guide a static analysis technique to identify feature-specific *computational units* (*i.e.,* units of source code).

Wong *et al.* base their analysis on the *Software Reconnaissance* approach and complement the relevancy metric by defining three new metrics to quantify the relationship between a source artifact and a feature [25]. Their focus is on measuring the closeness between a feature and a program component.

All of these feature identification approaches collect traces of method events and use this data to locate the parts of source code that implement a feature. Feature identification analysis is thus based on manipulating and analyzing large traces. A key difference to our live feature analysis approach is that these dynamic analyses focus primarily on method execution, thus do not capture fine-grained details such as sub-method execution events such as conditional branching and variable access. This information may prove essential when focusing on a malfunctioning feature or on boundary conditions. A main limiting factor of extracting this level of detail is the amount of trace data that would result. The key advantage of our approach is that we eliminate the need to retain execution traces for post-mortem analysis, as we perform a more focused, live analysis and discard the information when it is no longer needed. Thus there is no limitation to annotating all events (methods and sub-methods) involved in a feature's behavior.

Furthermore, a key focus of feature identification techniques is to define measurements to quantify the relevancy of a source entity to a feature and to use the results for further static exploration of the code. These approaches do not explicitly express the relationship between behavioral data and source code entities. To extract high level views of dynamic data, we need to process the large traces. Other works [1, 9] identify the need to extract a model of behavioral data in the context of structural data of the source code. Subsequently feature analysis is performed on the model rather than on the source code itself.

5 Conclusions and Future Work

In this paper we have proposed a *live feature analysis* approach based on Partial Behavioral Reflection to address some of the limitations of traditional trace-based feature analysis approaches. This technique relates features to the running code, thus opening up a number of possibilities to exploit feature information interactively and in context of the running system. While running the application to assess a reported problem a feature representation is built up at the same time, which is available for analysis in the context of the running system.

Our implementation of live feature analysis is based on the Reflectivity framework. We presented the implementation and applied our approach to a case-study that illustrates the usefulness and practicality of the idea, in particular from the point of view of growing features and obtaining fine-grained sub-method information in the feature representation.

To summarize, we revisit the goals defined in Section 1:

- **Data volume reduction.** Since feature information is recorded in live object structures, considerably less space is needed than with a post-mortem trace, which necessarily contains a great deal of redundant information. As an immediate side effect, the recorded information is easier to access and analyze.
- **Feature Growing.** Variants of the same feature can be exercised iteratively and incrementally, thus allowing analysis of the feature to "grow" within the development environment.
- **Sub-method feature analysis.** As shown in the validation section, feature analysis can be performed at sub-method level. This is of particular interest for long and complex methods.
- **Interactive feature analysis.** Features are analyzed at runtime for selected parts of the application. Features can be analyzed on-demand, depending on the task at hand.

The performance impact of our implementation is visible only at the first execution as we have shown that certain techniques can be introduced to attenuate this impact.

Having feature annotations represented in the system at the statement level opens up many possibilities for visualizing fine-grained feature behavior and interaction between different features. We plan to explore the idea of visualizing feature interaction at a sub-method level with microprints [20].

Another interesting direction for future work is to experiment with advanced scoping mechanisms. We want to experiment with the idea of scoping dynamic analysis towards a feature instead of static entities like packages and classes. In addition, we plan to explore the notion of scoping feature analysis itself towards features, which leads to the notion of analyzing the dependencies between features.

When analyzing which sub-method entity takes part in a feature, we up to now install the link very generously on all nodes of the AST. This is not needed

as we only need to make sure that we put annotations at runtime on all nodes in the method where the control-flow diverges. In a second step, we can use static analysis to propagate the information down the tree. We plan to experiment with this scheme in the future.

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the projects "Bringing Models Closer to Code" (SNF Project No. 200020-121594, Oct. 2008 - Sept. 2010) and "Biologically inspired Languages for Eternal Systems" (SNF Project No. PBBEP2-125605, Apr. 2009 - Dec. 2009) and CHOOSE, the Swiss Group for Object-Oriented Systems and Environments. We also thank Nicolas Anquetil and Lukas Renggli for their detailed reviews of drafts of this paper.

References

1. Giuliano Antoniol and Yann-Gaël Guéhéneuc. Feature identification: a novel approach and a case study. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'05)*, pages 357–366, Los Alamitos CA, September 2005. IEEE Computer Society Press.
2. Bas Cornelissen, Danny Holten, Andy Zaidman, Leon Moonen, Jarke J. van Wijk, and Arie van Deursen. Understanding execution traces using massive sequence and circular bundle views. In *Proceedings of the 15th International Conference on Program Comprehension (ICPC)*, pages 49–58. IEEE Computer Society, 2007.
3. Marcus Denker. *Sub-method Structural and Behavioral Reflection*. PhD thesis, University of Bern, May 2008.
4. Marcus Denker, Stéphane Ducasse, Adrian Lienhard, and Philippe Marschall. Sub-method reflection. In *Journal of Object Technology, Special Issue. Proceedings of TOOLS Europe 2007*, volume 6/9, pages 231–251. ETH, October 2007.
5. Marcus Denker, Orla Greevy, and Michele Lanza. Higher abstractions for dynamic analysis. In *2nd International Workshop on Program Comprehension through Dynamic Analysis (PCODA 2006)*, pages 32–38, 2006.
6. Marcus Denker, Orla Greevy, and Oscar Nierstrasz. Supporting feature analysis with runtime annotations. In *Proceedings of the 3rd International Workshop on Program Comprehension through Dynamic Analysis (PCODA 2007)*, pages 29–33. Technische Universiteit Delft, 2007.
7. Stéphane Ducasse, Michele Lanza, and Roland Bertuli. High-level polymetric views of condensed run-time information. In *Proceedings of 8th European Conference on Software Maintenance and Reengineering (CSMR'04)*, pages 309–318, Los Alamitos CA, 2004. IEEE Computer Society Press.
8. Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating features in source code. *IEEE Computer*, 29(3):210–224, March 2003.
9. Orla Greevy. *Enriching Reverse Engineering with Feature Analysis*. PhD thesis, University of Bern, May 2007.
10. Orla Greevy and Stéphane Ducasse. Correlating features and code using a compact two-sided trace analysis approach. In *Proceedings of 9th European Conference on Software Maintenance and Reengineering (CSMR'05)*, pages 314–323, Los Alamitos CA, 2005. IEEE Computer Society.
11. Orla Greevy, Stéphane Ducasse, and Tudor Gîrba. Analyzing feature traces to incorporate the semantics of change in software evolution analysis. In *ICSM'05*, pages 347–356, Los Alamitos, September 2005. IEEE Computer Society.

12. Orla Greevy, Stéphane Ducasse, and Tudor Gîrba. Analyzing software evolution through feature views. *Journal of Software Maintenance and Evolution: Research and Practice (JSME)*, 18(6):425–456, 2006.
13. Abdelwahab Hamou-Lhadj and Timothy Lethbridge. A survey of trace exploration tools and techniques. In *CASON 2004*, pages 42–55, Indianapolis IN, 2004. IBM Press.
14. Abdelwahab Hamou-Lhadj and Timothy Lethbridge. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In *ICPC'06*, pages 181–190, Washington, DC, USA, 2006. IEEE Computer Society.
15. Jay Kothari, Trip Denton, Spiros Mancoridis, and Ali Shokoufandeh. On computing the canonical features of software systems. In *WCRE 2006*, October 2006.
16. Adrian Lienhard, Orla Greevy, and Oscar Nierstrasz. Tracking objects to detect feature dependencies. In *ICPC'07*, pages 59–68, Washington, DC, USA, June 2007. IEEE Computer Society.
17. Alok Mehta and George Heineman. Evolving legacy systems features using regression test cases and components. In *Proceedings ACM International Workshop on Principles of Software Evolution*, pages 190–193, New York NY, 2002. ACM Press.
18. Lukas Renggli. Magritte — meta-described web application development. Master's thesis, University of Bern, June 2006.
19. Tamar Richner and Stéphane Ducasse. Using dynamic information for the iterative recovery of collaborations and roles. In *ICSM'02*, page 34, Los Alamitos CA, October 2002. IEEE Computer Society.
20. Romain Robbes, Stéphane Ducasse, and Michele Lanza. Microprints: A pixel-based semantically rich visualization of methods. In *Proceedings of 13th International Smalltalk Conference (ISC'05)*, pages 131–157, 2005.
21. David Röthlisberger, Oscar Nierstrasz, Stéphane Ducasse, Damien Pollet, and Romain Robbes. Supporting task-oriented navigation in IDEs with configurable HeatMaps. In *ICPC 2009*, pages 253–257, Los Alamitos, CA, USA, 2009. IEEE Computer Society.
22. Maher Salah and Spiros Mancoridis. A hierarchy of dynamic software views: from object-interactions to feature-interacions. In *ICSM 2004*, pages 72–81, Los Alamitos CA, 2004. IEEE Computer Society Press.
23. Éric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In *OOPSLA'03*, pages 27–46, nov 2003.
24. Norman Wilde and Michael Scully. Software reconnaisance: Mapping program features to code. *Journal on Software Maintenance: Research and Practice*, 7(1):49–62, 1995.
25. Eric Wong, Swapna Gokhale, and Joseph Horgan. Quantifying the closeness between program components and features. *Journal of Systems and Software*, 54(2):87–98, 2000.
26. Andy Zaidman and Serge Demeyer. Managing trace data volume through a heuristical clustering process based on event execution frequency. In *CSMR'04*, pages 329–338, Los Alamitos CA, March 2004. IEEE Computer Society Press.