Supporting Feature Analysis with Runtime Annotations *
– Position Paper –

Marcus Denker, Orla Greevy and Oscar Nierstrasz
Software Composition Group, University of Bern, Switzerland

Abstract

The dynamic analysis approach to feature identification describes a technique for capturing feature behavior and mapping it to source code. Major drawbacks of this approach are (1) large amounts of data and (2) lack of support for sub-method elements. In this paper we propose to leverage sub-method reflection to identify and model features. We perform an on-the-fly analysis resulting in annotating the operations participating in a feature's behavior with meta-data. The primary advantage of our annotation approach is that we obtain a fine-grained level of granularity while at the same time eliminating the need to retain and analyze large traces for feature analysis.

Keywords: behavioral reflection, annotations, dynamic analysis, feature analysis, reverse engineering, program comprehension, software maintenance

1 Introduction

Traditionally, reverse engineering techniques focused on analyzing source code of a system [2]. In recent years, researchers have recognized the significance of centering reverse engineering activities around the behavior of a system, in particular, around features [8, 17, 21]. Reasoning about object-oriented systems in terms of features is difficult, as they are not explicitly represented in the source code. The first step therefore is to define what is meant by a feature, establish a feature representation and to locate the relevant parts of the source code that participate in its behavior.

Most existing feature analysis techniques capture traces of method events but they do not capture behavioral data of sub-method elements such as variable assignments [21, 17]. Furthermore, modeling features themselves poses some problems: features are typically modeled as traces of runtime activity resulting in the manipulation and interpretation of large amounts of trace data.

*Proceedings of the 3rd International Workshop on Program Comprehension through Dynamic Analysis (PCODA 2007), pp. 29-33

In this paper we address the following issues relevant to dynamic feature analysis:

- Dynamic feature analysis implies a need to manipulate large amounts of trace data.
- Current feature analysis techniques do not consider analysis to the granularity of sub-method elements (*i.e.* variable assignments).

Our goal is to show how *feature annotation* eliminates the need to manipulate large traces and thus makes it possible to collect fine-grained detail about the parts of the code that are involved in the runtime of a feature.

Paper structure. In the next section we briefly describe the current dynamic analysis approach to feature analysis and highlight problems such as the manipulation of large amounts of runtime data and the extraction of fine-grained behavioral information. In Section 3 we briefly introduce *sub-method reflection*, as it serves as a basis for our approach. Subsequently, in Section 4 we introduce our feature annotation approach. We discuss different aspects of our approach in Section 5. Section 6 outlines related work in the fields of dynamic analysis and feature identification. Finally we conclude in Section 7.

2 Dynamic Feature Analysis in a Nutshell

The goal of feature analysis is to reason about a system in terms of its features. A fundamental step of any feature analysis approach is to first apply a *feature identification* technique to locate features in source code. As a basis for feature analysis we use a model which expresses features as first class entities and their relationships to the source entities that implement their behavior [10]. Once the representation of a feature is established, we can reason about a system in terms of its features. Furthermore, we can enrich the static source code perspective with knowledge of the roles of classes and methods in the set of modeled features.

The generally adopted definition of a feature is a unit of observable behavior of a system triggered by a user

[1, 8, 16, 24, 25]. Techniques for feature identification through dynamic analysis typically instrument a system, capture traces of feature behavior and establish links to source code. However, capturing dynamic data to represent features raises many issues that need to be taken into consideration.

Large Amounts of Data. The volume of trace data generated represents a threat to the scalability of any feature analysis approach. As the granularity required for an experiment increases, so too does the volume of information generated.

Dynamic analysis approaches adopt different strategies to deal with large amounts of data. Some of the most popular strategies adopted by researchers to tackle and analyze dynamic data are: (1) summarization through metrics [7], (2) filtering and clustering techniques [14, 26], (3) visualization [3, 12] (4) selective instrumentation and (5) query-based approaches [20]. Many techniques apply a combination of these strategies.

Instead of trying to compress the trace data, we need to question the idea of modeling features as execution traces.

Fine-Grained Analysis. Traditionally, dynamic analysis techniques for feature analysis focused on execution traces consisting of a sequence of method executions [8, 25]. Some dynamic analysis approaches trace additional properties of behavior such as the message receiver and arguments or instance creation events [4, 13]. However very little work in feature analysis has focused on a means to model which sub-method entities are part of a feature.

Fine-grained analysis down to the operation level should be possible.

Before we present our solution to these issues, we briefly introduce the extended reflection mechanism that enables *feature annotation* in Section 4.

3 Sub-Method Reflection

Reflection in programming languages is a paradigm that supports computations about computations, so-called *meta-computations*. Metacomputations and base computations are arranged in two different levels: the *metalevel* and the *base level* [22]. Because these levels are causally connected any modification to the metalevel representation affects any further computations on the base level [19]. Structurally reflective systems contain a first-class, causally connected model of their own structure: classes and methods are objects and changing these objects directly changes the system [9].

Structural reflection stops at the granularity of the method: a method is an object, but the operations the method contains are not modeled as objects. Examples of these operations would be message sends, variable reads or assignments. *Sub-Method Reflection* [5] extends the traditional model of structural reflection to encompass sub-method elements in addition to classes and methods. This is done by associating an extended AST (*Abstract Syntax Tree*) representation with the method.

Before execution, the AST is compiled on demand to a low-level representation that is executable, for example to byte-codes executable by a virtual machine.

Another mechanism provided by sub-method reflection is the annotation. Sub-method reflection provides a framework for annotating any program element with meta-data. An open compiler infrastructure supports the definition of compiler plugins that react to annotations by transforming the generated code.

We have extended Squeak Smalltalk to support sub-method reflection. More in-depth information about this system and its implementation can be found in the paper on sub-method reflection [5].

3.1 Partial Behavioral Reflection

Structural reflection is concerned with modeling the static structure of the systems. *Behavioral reflection* provides a model for execution and a way to intercept and change the execution of a program.

Whereas structural reflection is about classes, methods and the instructions inside the methods, behavioral reflection is concerned with execution events, *i.e.* method execution, message sends, or variable assignments.

One model for behavioral reflection is *Partial Behavioral Reflection* as pioneered by Reflex [23]. We have argued in the past [6] that this model is particularly well suited for dynamic analysis. It supports a very fine-grained, temporal and spatial selection of what exactly to reflect on. Thus it provides control of where and in which context dynamic analysis should be deployed in the system.

The core concept of the Reflex model of partial behavioral reflection is the *link* (see Figure 1). A link invokes messages on a metaobject at occurrences of selected operations. Link attributes enable further control of the exact message sent to the meta-object. One example for a link attribute is the activation condition which controls if the link is really invoked.

The original implementation of partial behavioral reflection for Java was based on bytecode transformation. Sub-Method Reflection provides a natural implementation substrate for realizing partial behavioral reflection. Links are annotations on the operations provided by sub-method-reflection. A plugin enables the compiler to take the links into account when generating the bytecode for a method.

4 Feature Annotation

Feature identification (*i.e.* locating which parts of the code implement a feature) by dynamic analysis is done at

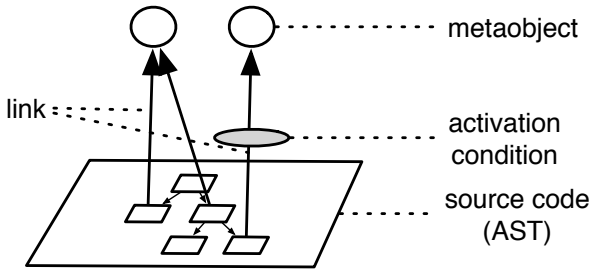

Figure 1. The reflex model

runtime: the feature is executed and the execution path is recorded. For example, when exercising *Login* feature of an application, we record all methods that are called as a result of triggering this feature. This trace of called methods then encompasses exactly all those methods that are part of the login feature.

4.1 Behavioral Reflection and Features

Trace-based feature analysis can be easily implemented using partial behavioral reflection. In a standard trace-based system, the tracer is the object responsible for recording the feature trace. This tracer is the meta-object (see Figure 2). We define a link that calls this meta-object with the desired information passed as a parameter (e.g. the name and class of the executed method). The link then is installed on the part of the system that we want to analyze. When we then exercise the feature, the trace meta-object will record a trace.

The resulting system is very similar to existing trace-based systems, with one exception: tracing now can easily cover sub-method elements, if required.

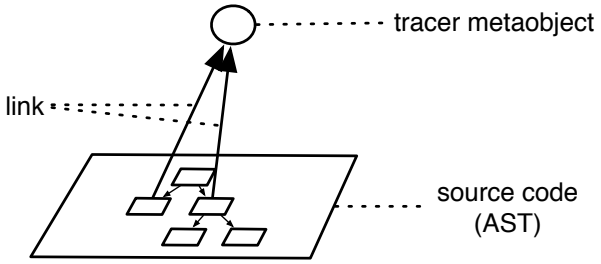

Figure 2. A tracer realized with partial behavioral reflection

4.2 Feature Annotation with Behavioral Reflection

In contrast to traditional dynamic feature analysis approaches, our sub-method reflection based approach does not need to retain a trace. The goal of feature identification is to map features to the source code. With the annotatable representation provided by sub-method reflection, we can annotate every statement that participates in the behavior of a feature. Instead of recording traces, we tag all the AST nodes that are executed as part of a feature with a *feature annotation* at runtime.

The annotation at runtime is realized using partial behavioral reflection. We do not need a dedicated tracer application anymore, instead the meta-object that models an instruction (the AST node) tags itself if it is part of a feature. For this, we define a link that calls a method on the node on which it is installed. This method tags the appropriate node with a feature annotation (see Figure 3).

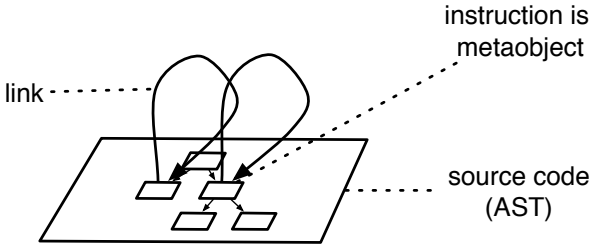

Figure 3. Annotating nodes using partial behavioral reflection

We install the link on all the AST nodes of the system that we plan to analyze. Exercising the feature subsequently annotates all methods or instructions that take part in a feature execution. In this way we do not need to retain traces, resulting in less data to be managed. We have not yet conducted extended case studies using our technique. However preliminary studies with feature traces captured using the traditional approach to tracing show that for the number of methods that are part of a trace, we get 10 times more method execution events. Thus there is a factor of 10 between the number of method execution events and the number of distinct methods in a trace.

The idea of *feature annotation* has many implications, both for how to model and analyze features. This position paper presents the basic idea of feature annoations, the next section discusses some of the possibilities and drawbacks of our approach.

5 Discussion

Our feature annotation approach can easily support many of the existing feature analysis approaches. For example, we could exercise a feature multiple times with different parameters to obtain multiple paths of execution. This can be important, as the traces obtained can very considerably

depending on the input data.

For trace-based approaches this results in 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.

Instead of multiple runs resulting in one feature annotation, the feature annotations can be parametrized with the amount of executions that are the result of exercising the feature. We can, for example, record a metric if a statement is always part of a feature or only in certain contexts similar to the reconnaissance metric of Wilde and Scully [24] or our other feature analysis work [11]. Other information that can be captured is *e.g.*, instance information or feature dependencies as described in the approaches of Salah *et al.* or Lienhard *et al.* [21, 18]. Naturally, the more information gathered at runtime, the more memory would be 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.

A downside of the filtering at runtime is that dynamic information is lost. It is crucial to define which information is necessarry for a given feature analysis. A change in a selection strategy implies a need to exercise a feature again. In contrast approaches based on complete traces can perform a variety of postmortem analyses of feature traces, each requiring different level of detail.

6 Related work

We review dynamic analysis approaches to system comprehension and feature identification approaches and discuss these in the context of our work.

Dynamic Analysis for Program Comprehension. Many approaches to dynamic analysis focus on the problem of tackling the large volume of data. Many of these works propose compression and summarization approaches to support the extraction of high level views [26, 11, 15].

Feature Identification through dynamic analysis. Dynamic analysis approaches to feature identification have typically involved executing the features of a system and analyzing the resulting execution trace [24, 25, 8, 1]. 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 artefact 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. Thus, the feature identification analysis is based on manipulating and analyzing large traces. Furthermore, many of the dynamic analysis approaches do not capture fine-grained details such sub-method execution events. The main limiting factor is the amount of trace data that would result. Our approach eliminates the need to retain execution traces. 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. Thus these approaches do not explicitly express the relationship between behavioral data and source code entities. Thus to extract high level views of dynamic data, we need to process the large traces. Other works [1, 10] 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.

7 Conclusions and Future Work

In this paper we have presented *feature annotation*, a technique that solves some issues found in traditional trace-based dynamic feature analysis systems. Feature Annotation support the analysis on a sub-method level and does not require to store complete trace data.

We have implemented a prototype of feature annotation, future work includes using it on large case-studies. We plan to analyze both performance and memory characteristics and compare our approach to a trace collecting feature

analysis system.

Another interesting direction of future work is to experiment with advanced scoping mechanisms, *e.g.* we want to experiment with the idea of scoping dynamic analysis towards a feature instead of static entities like packages and classes.

Acknowledgments.. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Analyzing, capturing and taming software change" (SNF Project No. 200020-113342, Oct. 2006 - Sept. 2008).

References

[1] G. Antoniol and Y.-G. Guéhéneuc. Feature identification: a novel approach and a case study. In *Proceedings IEEE International Conference on Software Maintenance (ICSM 2005)*, pages 357–366, Los Alamitos CA, Sept. 2005. IEEE Computer Society Press.

[2] E. Chikofsky and J. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, Jan. 1990.

[3] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J. van Wijk, and A. 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.

[4] W. De Pauw, D. Lorenz, J. Vlissides, and M. Wegman. Execution patterns in object-oriented visualization. In *Proceedings Conference on Object-Oriented Technologies and Systems (COOTS '98)*, pages 219–234. USENIX, 1998.

[5] M. Denker, S. Ducasse, A. Lienhard, and P. Marschall. Sub-method reflection. *Journal of Object Technology*, 6(9):231–251, Oct. 2007.

[6] M. Denker, O. Greevy, and M. Lanza. Higher abstractions for dynamic analysis. In *2nd International Workshop on Program Comprehension through Dynamic Analysis (PCODA 2006)*, pages 32–38, 2006.

[7] S. Ducasse, M. Lanza, and R. 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] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Computer*, 29(3):210–224, Mar. 2003.

[9] J. Ferber. Computational reflection in class-based object-oriented languages. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, volume 24, pages 317–326, Oct. 1989.

[10] O. Greevy. *Enriching Reverse Engineering with Feature Analysis*. PhD thesis, University of Berne, May 2007.

[11] O. Greevy and S. 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.

[12] O. Greevy, S. Ducasse, and T. Gîrba. Analyzing software evolution through feature views. *Journal of Software Maintenance and Evolution: Research and Practice (JSME)*, 18(6):425–456, 2006.

[13] T. Gschwind and J. Oberleitner. Improving dynamic data analysis with aspect-oriented programming. In *Proceedings of CSMR 2003*. IEEE Press, 2003.

[14] A. Hamou-Lhadj and T. Lethbridge. A survey of trace exploration tools and techniques. In *Proceedings IBM Centers for Advanced Studies Conferences (CASON 2004)*, pages 42–55, Indianapolis IN, 2004. IBM Press.

[15] A. Hamou-Lhadj and T. Lethbridge. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In *Proceedings of International Conference on Program Comprehension (ICPC 2006)*, pages 181–190, Washington, DC, USA, 2006. IEEE Computer Society.

[16] R. Koschke and J. Quante. On dynamic feature location. *International Conference on Automated Software Engineering, 2005*, pages 86–95, 2005.

[17] J. Kothari, T. Denton, S. Mancoridis, and A. Shokoufandeh. On computing the canonical features of software systems. In *13th IEEE Working Conference on Reverse Engineering (WCRE 2006)*, Oct. 2006.

[18] A. Lienhard, O. Greevy, and O. Nierstrasz. Tracking objects to detect feature dependencies. In *Proceedings International Conference on Program Comprehension (ICPC 2007)*, pages 59–68, Washington, DC, USA, June 2007. IEEE Computer Society.

[19] P. Maes. *Computational Reflection*. PhD thesis, Laboratory for Artificial Intelligence, Vrije Universiteit Brussel, Brussels Belgium, Jan. 1987.

[20] T. Richner and S. Ducasse. Using dynamic information for the iterative recovery of collaborations and roles. In *Proceedings of 18th IEEE International Conference on Software Maintenance (ICSM'02)*, page 34, Los Alamitos CA, Oct. 2002. IEEE Computer Society.

[21] M. Salah and S. Mancoridis. A hierarchy of dynamic software views: from object-interactions to feature-interacions. In *Proceedings IEEE International Conference on Software Maintenance (ICSM 2004)*, pages 72–81, Los Alamitos CA, 2004. IEEE Computer Society Press.

[22] B. C. Smith. Reflection and semantics in a procedural language. Technical Report TR-272, MIT, Cambridge, MA, 1982.

[23] É. Tanter, J. Noyé, D. Caromel, and P. Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In *Proceedings of OOPSLA '03, ACM SIGPLAN Notices*, pages 27–46, nov 2003.

[24] N. Wilde and M. Scully. Software reconnaisance: Mapping program features to code. *Software Maintenance: Research and Practice*, 7(1):49–62, 1995.

[25] E. Wong, S. Gokhale, and J. Horgan. Quantifying the closeness between program components and features. *Journal of Systems and Software*, 54(2):87–98, 2000.

[26] A. Zaidman and S. Demeyer. Managing trace data volume through a heuristical clustering process based on event execution frequency. In *Proceedings IEEE European Conference on Software Maintenance and Reengineering (CSMR*

2004), pages 329–338, Los Alamitos CA, Mar. 2004. IEEE Computer Society Press.