Dynamix - a Meta-Model to Support Feature-Centric Analysis

Orla Greevy

Software Composition Group
University of Berne, Switzerland
greevy@iam.unibe.ch

Abstract

Many researchers have identified the potential of exploiting domain knowledge in a reverse engineering context. Features are abstractions that encapsulate knowledge of a problem domain and denote units of system behavior. As such, they represent a valuable resource for reverse engineering a system. The main body of feature-related reverse engineering research is concerned with feature identification, *a technique to map features to source code. To fully exploit features in reverse engineering, however, we need to extend the focus beyond feature identification and exploit features as primary units of analysis.*

To incorporate features into reverse engineering analyses, we need to explicitly model features, their relationships to source artefacts, and their relationships to each other. To address this we propose Dynamix, am meta–model that expresses feature entities in the context of a structural meta-model of source code entities. Our meta-model supports feature-centric reverse engineering techniques that establish traceability between the problem and solution domains throughout the life-cycle of a system.

Keywords: meta-model, feature analysis, program comprehension, software maintenance

1 Introduction

From an external perspective, users understand a system as a collection of features that correspond to system behaviors to fulfill requirements. As such, features are well-understood abstractions that encapsulate domain knowledge and denote a system's behavioral units. However, the software engineer cannot identify and manipulate features, as they are not explicitly represented in the source code of object-oriented systems. Typically, feature implementation cross-cuts the structural boundaries (*i.e.,* packages and classes) of an object-oriented system [9].

A software engineer is frequently confronted with features. Typically, change requests and bug reports are expressed in a language that reflects the features of a system [8]. Therefore, to perform maintenance tasks, a software engineer needs to maintain a mental map between the features and their implementation as source artefacts.

To support the software engineer during maintenance activities, system comprehension techniques need to incorporate and explicitly model the notion of a feature as a first-class entity.

Typically the software developer is familiar with a structural representation of a system's source code, for example a UML class diagram. UML describes sequence diagrams to represent runtime behaviors. To support comprehension, we need to capture the relationship between the structural perspective of a system in terms of software artefacts and the dynamic behavioral entities of an object-oriented system, namely object instantiations and message sends. To represent features we understand dynamic behavior in terms of units that correspond to the features of a system.

In this paper, we describe Dynamix, our meta-model that supports feature-centric analysis of object-oriented systems by focusing on features as first-class entities of analysis in the context of the structural entities, namely packages, classes and methods.

Structure of the Paper. In the next section, we identify the motivation for descibing a meta-model for features. In Section 3 we introduce Dynamix our meta-model for expressing features as first-class entities in the context of a structural model of the source code. We present related work in Section **??** and finally in Section **??** we outline our conclusions.

2 Motivating a Meta-Model for Feature-Centric Analysis

Our work is centered around the notion of a feature. We adopt the definition of a feature proposed by Eisenbarth *et* *al.* [4] as a unit of behavior of a system triggered by the user, as it is generally accepted by other researchers in a reverse engineering context [1, 5]:

The main goal of our research is to identify how we can exploit domain knowledge of object-oriented systems that is inherent in a user's perspective of how a system behaves at runtime so that (1) existing reverse engineering analyses can be enriched with semantic context, and (2) we can define reverse engineering analysis techniques that exploit the notion of features as first-class entities [7]. We establish the goals of Dynamix, our meta-model to express features in the context of a system's behavioral and structural entities as follows:

1. *Behavior.* Due to language features like polymorphism and late binding of object-oriented systems, behavior of a system cannot be completely automatically determined by analyzing its source code alone. Thus, to capture a system's behavior, we need to perform dynamic analysis.

2. *Exploiting Domain Knowledge.* Our research question is centered around the problem of exploiting domain knowledge to enhance system comprehension. We consider features to be units of behavior encapsulating domain knowledge.

3. *Combining Static and Dynamic Analysis.* Two main distinct approaches to system comprehension have dominated reverse-engineering research efforts [2]: dynamic analysis approaches and static analysis approaches. Both perspectives are necessary to support the understanding of object-oriented systems [3].To complement structural analysis of a system, roles of source artefacts need to be enriched with feature context (*i.e.,* how they participate in features at runtime).

4. *Features as First-Class Entities.* During the lifetime of a system, software engineers are constantly required to modify and adapt application features in response to changing requirements. A reverse engineering analysis needs to support this activity by breaking the system into groupings that reflect its features. As a basis of any feature-centric analysis we need to define a meta-model that treats features as first class entities (*i.e.,* primary units) and establishes relationships between features and source artefacts implementing their functionality. Therefore, an underlying model should unify behavioral data of features and structural data of source code such as packages, classes and methods. A unified model would provide a framework for our feature-centric analysis. The model needs to be generic, extensible and should easily accommodate metrics from other feature analysis techniques.

5. *Feature Relevancy Measurements.* Feature identification represents the foundation of our work. Thus, a feature-centric analysis approach needs to provide a measurement to quantify the relevance of a software artefact to a feature, or set of features.

6. *Feature Relationships.* Software engineers need to understand relationships between features, as modifications to one feature may inadvertently affect other features. Furthermore, feature relationships reflect constraints and dependencies in a problem domain. Thus, they are important sources of information for system comprehension. A feature-centric analysis approach needs to identify and quantify relationships and dependencies between features.

3 Dynamix

We introduce Dynamix, our meta-model to specify behavioral entities of feature execution data and their relationships. Dynamix also specifies the relationships between the behavioral entities and the structural entities representing source artefacts. Dynamix is MOF 2.0 compliant [1]. Our OCL specifications comply with OCL 2.0 [2].

To obtain a model of dynamic and static data of a system under study, we first extract a structural model by parsing a system's source code. Then, we extract *feature traces* by exercising a set of features on an instrumented system. We transform the execution data of feature traces into Dynamix entities and establish the relationships between the execution entities and the source entities of the structural model.

In Figure 1 we show the entities of our model in a UML 2.0 diagram [6]. The *Features* package represents the dynamic behavioral data of the feature traces. The *Structure* package models the entities of the source code. We model behavioral data of features using three entities: *Feature*, *Activation* and *Instance*.

Feature. Each feature trace we capture during dynamic analysis of a system is modeled as a *Feature* entity. A *Feature* entity is uniquely identified by a name. The *Feature* entity allows us to collectively manipulate all the *Activations* that correspond to the events of the feature trace which it models. It maintains a list (modeled as an ordered collection) of all of its *Activations* for ease of manipulation. The first *Activation* of the list represents the root of a feature trace. We assign properties to *Feature* entities based on the *Activations* and their relationships to other entities (*e.g.,* number of *Activations*, number of *Instances* created, number of *Methods* referenced, and feature affinity properties). Relationships between features are shown in the

[1]http://www.omg.org/docs/ptc/03-10-04.pdf

[2]http://www.omg.org/docs/formal/06-05-01.pdf

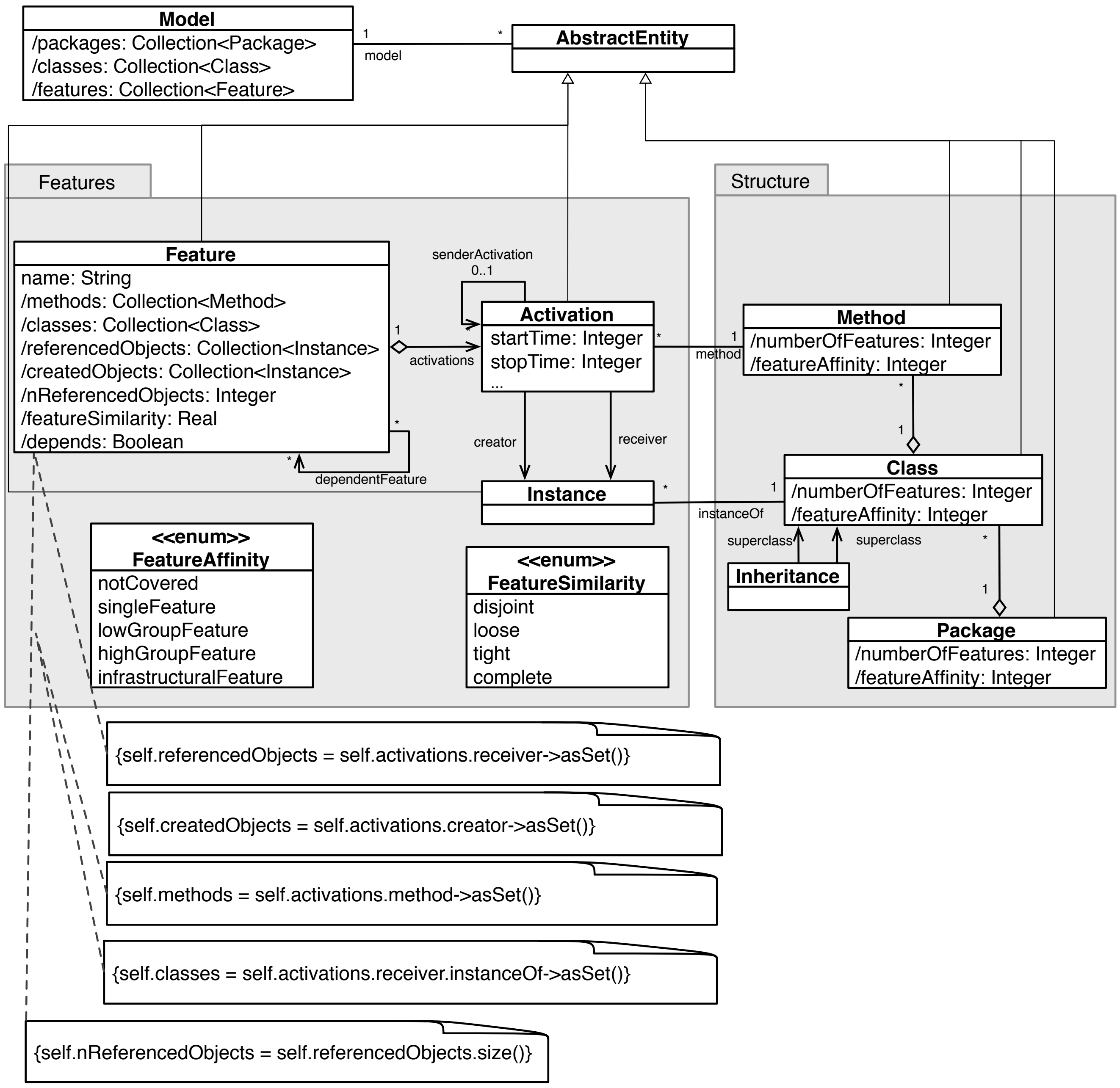
Model
/packages: Collection<Package>
/classes: Collection<Class>
/features: Collection<Feature>
1
model
*
AbstractEntity
Features
Structure
Feature
name: String
/methods: Collection<Method>
/classes: Collection<Class>
/referencedObjects: Collection<Instance>
/createdObjects: Collection<Instance>
/nReferencedObjects: Integer
/featureSimilarity: Real
/depends: Boolean
senderActivation
0..1
1
activations
*
Activation
startTime: Integer
stopTime: Integer
...
*
1
method
Method
/numberOfFeatures: Integer
/featureAffinity: Integer
*
1
creator
receiver
*
*
dependentFeature
Instance
*
1
instanceOf
Class
/numberOfFeatures: Integer
/featureAffinity: Integer
superclass
superclass
*
Inheritance
1
Package
/numberOfFeatures: Integer
/featureAffinity: Integer
<<enum>>
FeatureAffinity
notCovered
singleFeature
lowGroupFeature
highGroupFeature
infrastructuralFeature
<<enum>>
FeatureSimilarity
disjoint
loose
tight
complete
{self.referencedObjects = self.activations.receiver->asSet()}
{self.createdObjects = self.activations.creator->asSet()}
{self.methods = self.activations.method->asSet()}
{self.classes = self.activations.receiver.instanceOf->asSet()}
{self.nReferencedObjects = self.referencedObjects.size()}

Figure 1. The Dynamix Meta-Model

model with a *depends* association. We provide the OCL definition for this relationship between features in Figure 2.

Activation. An *Activation* in our model represents a method execution. It holds a reference to its sender *Activation*. In this way Dynamix models the tree structure of a feature trace. Thus, the model preserves the sequence of execution of method executions of a feature trace. Time is captured and modeled with two attributes, namely *startTime* (*i.e.,* the timestamp in milliseconds, when the method was invoked) and *stopTime* (*i.e.,* the timestamp in milliseconds when it completed execution) of an activation . Each *Activation* is associated with a *Method* entity in the structural model. The *Method* entity of the structural model has a relationship to the *Class* entity where it is defined. In this way, we model relationships between features and source entities. Furthermore, an *Activation* is associated with an *Instance* entity which represents the receiver instance of a message. The sender instance is accessible via its sender *Activation*. Thus, Dynamix models the actual object that invokes a method. This does not necessarily correspond to the static relationship between *Method* and *Class* entities, due to inheritance in object-oriented systems. The return value of a message is also stored as a reference to an *Instance* entity in the *Activation* that models the message send.

Instance. We model every instantiated object of a feature trace as an *Instance* entity. An *Instance* is created by an *Activation* and maintains a list of references to all *Activations* that hold a reference to this object (*i.e., Activations* reference the receiver instance of a message, *Activations* that hold a reference to the *Instance* in the return value of a message send). The *Instance* is associated with its defining *Class* entity of the structural model.

Dynamix supports feature analysis from different levels of granularity. We exploit relationships between *Feature* entities and source entities to view a system at the package, class or method level of detail. When analyzing large and complex systems, we may need to obtain a *big picture* perspective to locate where features are implemented. In this case, we focus on the relationships between features and packages. For more fine-grained perspectives of feature implementation, we analyze feature-to-class and feature-to-method relationships.

Figure 1 shows an *AbstractEntity* from which the entities (Structure and Feature entities) of our model, are derived. A *Model* comprises every entity, and every entity is associated with the *Model* entity. For example a *Method* entity obtains a collection of all the *Feature* entities in the model via this association.

Our Dynamix model as shown in Figure 1 models (1) sequential programs, (2) one path of execution of features. We show how Dynamix can be extended to model multi-threaded applications and multiple execution paths of features and discusses how this influences the analysis approaches described in this dissertation in our previous work [7].

4 Related Work

5 Conclusion

To fully exploit features in reverse engineering, we need to treat features as primary units of analysis. We motivated the need to describe a meta-model for features : (1) to enrich reverse engineering analysis techniques that extract structural views of a system with semantic knowledge about roles of source artefacts in features of a system, and (2) to reason about a system in terms of features themselves and relationships between features.

We describe Dynamix, a meta-model that expresses the execution entities of feature behavior and their relationships. Furthermore our meta-model expresses the relationships between the execution entities and a structural model of source code. Dynamix supports analysis that combines static and dynamic views of a system.

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] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *Proceedings of 15th International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*, pages 166–178, New York NY, 2000. ACM Press. Also appeared in ACM SIGPLAN Notices 35 (10).

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

[5] A. Eisenberg and K. De Volder. Dynamic feature traces: Finding features in unfamiliar code. In *Proceedings IEEE International Conference on Software Maintenance (ICSM 2004)*, pages 337–346, Los Alamitos CA, Sept. 2005. IEEE Computer Society Press.

[6] M. Fowler. *UML Distilled*. Addison Wesley, 2003.

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

[8] A. Mehta and G. 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.

[9] 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.

```
context Feature
  def: importedObjects : Set(Instance) =
         self.referencedObjects->excluding(self.createdObjects)

context Feature
  def: depends(aFeature: Feature) : Boolean =
  ( self.importedObjects->intersection(aFeature.createdObjects)->size() > 0)
               and not (self = aFeature)
```

Figure 2. OCL specification of depends relationship between features.