Understanding Software Evolution using a Flexible Query Engine

Michele Lanza, Stéphane Ducasse, Lukas Steiger
Software Composition Group, University of Berne
Neubrückstrasse 12, CH – 3012 Berne, Switzerland
{lanza,ducasse,steiger}@iam.unibe.ch — http://www.iam.unibe.ch/∼scg/

Published in the Formal Foundations of Software Evolution Workshop Proceedings of CSMR 2001

Abstract

One of the main problems which arises in the field of software evolution is the sheer amount of information to be dealt with. Compared to reverse engineering where the main goal is the main understanding of one single system. In the field of software evolution this information is multiplied by the number of versions of the system one wants to understand. To counter this problem we have come up with a flexible query engine which can perform queries on the different versions of a system. In this paper we give an outlook on our current work in the field of software evolution and focus particularly on the concepts behind the query engine we have built.

Keywords: Reverse Engineering, Evolution, Moose, Object-Oriented Programming

1 Introduction

Understanding software systems that have evolved over several versions is difficult because of two main obstacles:

- The changes on a system during its development are often not or badly documented for several reasons. We believe one of the main forces is the weak enforcement of change documentation policies in companies: the people who perform the changes know what they are doing, so what's the point of documenting it?
- The original design document is not updated according to the performed changes, which leads to a rapid decay in the original design coherence.
- The amount of information is multiplied by the number of versions of the subject system: coping with such amounts of information is difficult and time-consuming.

Software Evolution is confronted with the difficulty of recovering such changes through the analysis of two or more versions of the same system. The main problem here is the amount of useless "noise" (i.e. false positives) which is returned.

To counter this problem we have come up with the idea of a flexible query engine similar to those used for professional databases. In a query language like SQL it is farily easy to define a query which can retrieve a certain set of data out of a possibly huge collection of data. Moreover it is also possible to further refine the query by adding more criteria.

This paper is structured as follows: in the next section we present the concepts and prerequisites of our query engine. We then show how the queries are made. Then we shortly present the tool which was realized using those concepts, and present some results obtained using the query engine on several case studies. In the final section of the paper we discuss the current and future work that we plan to do in this domain.

2 The Concepts and Prerequisites of the Query Engine

2.1 The Concept

The whole concept of such a query engine is based on the Composite Pattern[7]: The intent is to compose objects (in our case queries) into tree structures to represent part-whole hierarchies. A composite lets clients treat individual objects (queries) and compositions of objects (composed queries) uniformly.

A composed query can thus be seen as a hierarchy of queries and subqueries glued together by binary logical operators, i.e. AND and OR. A query can of course also be negated by assigning a unary NOT operator to the query. A name can be assigned to a query, through which it can be included by reference in other queries.

2.2 The Prerequisites

A query engine like ours has some prerequisites which must be fulfilled. The following prerequisites must hold:

- **A Collection of Data.** The primary prerequisite for such a query engine is a collection of data which behaves like a database on which queries can be performed. In our case we have our reengineering environment Moose[6] that we have developed during the FAMOOS ESPRIT project[4]. Note that Moose keeps all entities in memory, instead of using a file based approach like a database. Although we know that a database is more scalable we have not encountered size problems until now.

  The Moose reengineering environment is an implementation of the language independent FAMIX metamodel[3]. At this time the following languages can be represented in our metamodel: Smalltalk, Java, C++ and COBOL.

  We parse the source code (directly in the case of Smalltalk and using parsers in the case of the other languages) and end up with a collection of entities which are an internal representation of software artifacts. In the context of evolution it is important that we can have several metamodels (e.g. several versions of the same software) parallel to each other at the same time in memory.

- **A Query Language.** Although we could have used Smalltalk as the query language, we have decided to build a textual query language which can be expressed at a graphical user interface level. The benefits of this are that non-Smalltalkers can also make use of it and a bigger ease of expression.

- **A Metrics Framework.** Most of the queries we perform are based on metric properties of the entities. For that purpose we have implemented a large framework of metrics (at this time more than 50), which is better explained in [8].

3 A Taxonomy of Queries

In this section we explain what kinds of queries we can build and how they can be composed into more complex ones. Note that the notation we use in this paper does not reflect the actual notation we use, which is much more verbose. For the sake of simplicity and readability we have decided on this easy-to-understand notation.

In this section we will show how with our query engine we can compose step by step a query which in the end will return the following result:

3.1 Basic Queries and Composite Queries

A basic query checks whether a certain condition holds or not, i.e. it iterates over one or several metamodels and returns entities which match the query. We now present how basic queries can be combined to compose a refined query which returns specific results. We distinguish four kinds of basic queries, i.e.

1. **Type Query**

   A type query returns all entities which belong to a certain type. The example below returns all classes of a system.

   ```
   ClassesQuery :=
   [Type(x) = CLASS]
   ```

2. **Name Query**

   This is a simple name matching query including wildcards. The example below returns all classes whose name contains the string “Abstract”.

```
AbstractClassesQuery :=
[ClassesQuery] AND
[Name(x) = '*Abstract*']
```

3. **Property Query**

   In our metamodel we can annotate properties on an entity. Examples of such properties include whether a class is abstract, whether a method is an accessor (i.e. get/set), whether an attribute is private, etc. A property query tries to match a property which always returns a boolean value. The example below returns all classes which contain the substring “Abstract” in their name but in fact are not abstract.

   ```
   FalseAbstractClassesQuery :=
   [AbstractClassesQuery] AND
   [Property.Abstract(x) = FALSE]
   ```

4. **Metric Query**

   Moose provides an extensive set of metrics for the entities, including most of the metrics mentioned in [1] and [9]. In the case of such a query we either match the exact value or check on whether a metric value of an entity is above or below a certain threshold. The example below returns the false abstract classes in the system which implement more than 30 methods.

   ```
   LargeFalseAbstrClassesQuery :=
   [ FalseAbstractClassesQuery ]
   AND
   [ NOM(x) > 30 ]
   ```

3.2 Software Evolution Queries

A query can be composed of other (sub)queries. Those can be combined using binary logical operators, i.e. AND and OR like we have seen above.

In the case of Software Evolution Queries, we build queries which return results from different versions of the software and combine those results using logical unary (NOT) and binary (AND,OR) operators.

Suppose we have three versions of the software *Foo*. We call the versions *Foo1, Foo2, Foo3*. If we consider only *Foo1* and *Foo2*. We want to find all find all classes which from one version to next increased their number of methods by more than 20 (e.g. the class grew rapidly).

For that purpose we build the query

```
GrowQuery :=
[(NOM(x.new) - NOM(x.old)) > 20]
```

Here x represents the classes present in the new and the old version of the software and NOM is the value of the metric “Number of Methods” for x. This will return the results for (*Foo1, Foo2*). We can apply the same query to (*Foo2, Foo3*).

The combination of these through a logical AND operator will return the classes which grew constantly by at least 20 methods over the whole time frame we are considering. The combination of these through a logical OR operator will return the classes which grew at an arbitrary point in time.

3.3 Defining the Environment of a Query

Sometimes it is necessary to define a subquery on a query. We call this subquery the *environment* of the query. As an example, we want to find out all classes who grew by addition of methods and whose subclasses (at least one of them) shrank by removal of methods. Our guess is that in such a case the step in between performed by the developers was to push up the functionality of the subclasses into the superclass which grew. The criteria are in this case:

```
PushUpCandidates :=
[(NOM(x.new) - NOM(x.old)) > 0]
AND
[((NOM(subclasses(x.new) -
    NOM(subclasses(x.old)) < 0]
```

3.4 The Renamed Entity Tracking Problem

One of the major problems which must be dealt with, is that although conceptually two different versions of the same software entity have a “becomes” relationship, in our metamodel those are two different objects. To establish the connection between them, the obvious way is to go over the naming: if two entities have the same unique name, they are two versions of the same software artifact. However, what happens if an entity has been renamed?

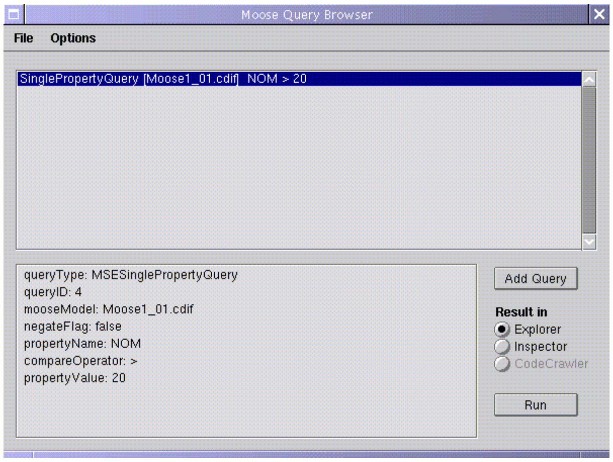
We have found two simple and effective solutions to this problem which cover almost all cases:

1. Using the metrics. We compare the metric measurements of the "new" entities (i.e. those which have appeared for the first time in a certain version of the software) with those of the previous ones and see if there is a match. This solution is straight forward but not very effective.

2. In the case of classes or higher level software constructs like packages, etc. we go over the entities contained in them. As an example, in the case of a renamed class we check if we have a match regarding the methods: if the name of certain methods stays the same, but the unique name (i.e. including their class name) changes we can be nearly sure that we have a renamed class.

These two approaches work well enough for us, although in both cases there are false positives. However, the only bullet-proof way to track the renamed entities would be to have a versioning software which tracks all entities including the renamed ones.

4 The Implementation of the Query Engine



SinglePropertyQuery [Moose1_01.cdif] NOM > 20

queryType: MSESinglePropertyQuery
queryID: 4
mooseModel: Moose1_01.cdif
negateFlag: false
propertyName: NOM
compareOperator: >
propertyValue: 20

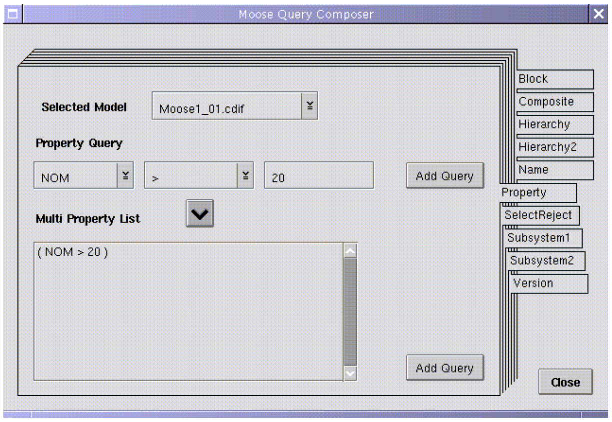
Figure 1: The Main Window of MooseFinder.

We have implemented the concept of the query engine in a tool called MooseFinder.

We have seen that the ease and flexibility of the query composition mechanism is very important: Often a query which works (i.e. returns useful results) in one context must be changed for another context.

For that purpose MooseFinder supports an easy and graphical way to compose queries including drag and drop support. This is necessary to enable the user to quickly adapt complex query structures to new contexts.

The window shown in Figure 1 is the main interface of MooseFinder. Here we can select the queries and run them.



Selected Model: Moose1_01.cdif

Property Query: NOM > 20

(NOM > 20)

Figure 2: The Query Composition Window of MooseFinder.

The Query Composition Window shown in Figure 2 enables the user to build the basic queries and compose them into composite ones.

5 Applying the Approach

The result of the approach we are working on, is to obtain a set of queries which return meaningful results in the field of software evolution. For that purpose we have set up a number of large and very large case studies we want to work on.

This work is still under way but we have already identified some useful queries. We list here what we can detect with each query:

- Introduction of a class on top of a large hierarchy

- Subclasses that become the sibling of their superclasses, i.e. that have been pushed up one hierarchy level
- Classes where methods and/or attributes have been pushed up into their superclass
- Classes that have rapidly grown/shrunk from one version to the next
- Classes which have been merged
- Entities which have been added to/removed from the software at a certain point
- Classes which have been renamed

6 Conclusions and Future Work

The preliminary results obtained using this approach have already shown that it is indeed useful and can return meaningful results. However, we have encountered the following problems:

- The usefulness of the approach is tied to the flexibility and power of the query language. This is on one hand the query language per se, on the other hand the user interface with which we can compose the queries.
- This approach goes into the direction of data mining and data reverse engineering. One of the main problems in those fields is the representation of the results. For the time being we still use textual representations, although we can easily interface with visualization software.
- The more general and less specific a query is, the more results it will return. On the other hand a very specific query can return an empty set of results. The fine-tuning of the queries requires a considerable deal of expertise on side of the user and flexibility on side of the query engine.

Our future work in this context includes the publication of a paper with the major results obtained with this approach applied on several large and very large case studies.

Furthermore we will extend the query engine and its query language to render it as flexible and powerful as possible.

We also plan to use the software visualization tool CodeCrawler [8, 2, 5] in this context.

References

[1] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[2] S. Demeyer, S. Ducasse, and M. Lanza. A hybrid reverse engineering platform combining metrics and program visualization. In F. Balmas, M. Blaha, and S. Rugaber, editors, *Proceedings WCRE'99 (6th Working Conference on Reverse Engineering)*. IEEE, October 1999.

[3] S. Demeyer, S. Tichelaar, and P. Steyaert. FAMIX 2.0 - the FAMOOS information exchange model. Technical report, University of Bern, August 1999.

[4] S. Ducasse and S. Demeyer, editors. *The FAMOOS Object-Oriented Reengineering Handbook*. University of Bern, October 1999. See http://www.iam.unibe.ch/˜famoos/handbook.

[5] S. Ducasse and M. Lanza. Towards a methodology for the understanding of object-oriented systems. *Technique et science informatiques*, 20(4):539–566, 2001.

[6] S. Ducasse, M. Lanza, and S. Tichelaar. Moose: an extensible language-independent environment for reengineering object-oriented systems. In *Proceedings of the Second International Symposium on Constructing Software Engineering Tools (CoSET 2000)*, June 2000.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, Reading, Mass., 1995.

[8] M. Lanza. Combining metrics and graphs for object oriented reverse engineering. Diploma thesis, University of Bern, October 1999.

[9] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics: A Practical Guide*. Prentice-Hall, 1994.