Supporting Evolution Recovery: A Query-based Approach

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

Submitted to Architectural Evolution International workshop of ECOOP'2000

Abstract

As software industry is shifting its focus from monolithic application development towards the use of frameworks and system families because of the higher reusability, the understanding of the evolution of a software system has grown in importance. The major problem which in this context is the management of the huge quantity of information in a meaningful way. We are currently exploring recovery of evolution information by means of queries over several versions of a system. This paper describes our approach and focuses on the advantages and drawbacks, as well as our future work in the area of software evolution.

Keywords: *Reengineering, Reverse Engineering, Refactoring, Language Independent, Software Metrics, Object-Oriented Programming, Queries*

1 Introduction

During the development of a software system over the years, a version or a release of the system is not always a good indicator of the changes which have been performed by the developers. The documentation itself is seldom synchronized with the versions and it becomes difficult to maintain a meaningful documentation of the system changes. In some cases the database containing the version information only records the addition or deletion of code entities, but not the changes themselves. This leads to a semantically weak information repository about the history of the system. In such a case not only can system changes become difficult to perform as they are based on insufficient information about the system, it may also happen that the system is changed in a way which has already been performed but which has been forgotten[9].

This leads to a situation where companies have a large number of versions of their systems, but have lost the meaning of the changes between the versions.

The history of a system has grown in importance over the recent years, since software industry has shifted the focus from monolithic applications towards the use of frameworks and system families for the sake of reuse. A system family is defined as a group of systems sharing a common set of assets (domain model, reference architecture, components) defined from earlier developed systems belonging to the same product line.

The analysis of the evolution of a system or system family has thus become important in the context described above.

Providing evolution analysis would help the developers to :

- gain an overview of the system evolution,
- qualify the parts of a system in terms of their stability over the versions,
- identify system components which could be reused in other systems.
- identify the migration of components between system parts, and
- identify components that have appeared or disappeared over the life-cycle of the system.

In this position paper we present our vision to address the problem of understanding software evolution. Our approach is based on the identification of source code artifacts over several versions of the same system. This identification is based on the combination of metrics, filtering and the construction of an application map that represents the evolution of the application itself.

The paper is structured as follows: first we present our approach in depth. Then we present the tools that we are currently developing and sketch one experience obtained from a case study..

2 Axes in Evolution Recovery

Our approach to support the recovery of application evolution is divided on two axes: one axis represents the influence of versions on the analysis and the other one is the granularity of the analyzed entities. These axes define the space over which we can analyze the evolution of an application.

Axis picture

The influence of versions. A single version, two specific versions or several versions can be analyzed regarding the evolution of the source code entities.

The analysis of two specific versions can identify the entities that have changed from one version to the other: some may have been added, removed, renamed, or may have changed in size, complexity and importance regarding the system. The same applies in the case of several versions of the system. In such a case we can observe the life-cycle of single entities and follow what changes have been performed on an entity.

From simple entities to groups. The analysis can work at the source code entity level like classes, methods or attributes, as described above. It may also work at a higher abstraction level: we can group entities. A group represents a set of source code entities which fulfill certain conditions.

An example of a group is the one that represents all classes defined in a subdirectory. Another example is the group of methods which access a certain attribute or the group of classes which belong to a certain inheritance hierarchy. Taking such groups into consideration can provide a higher view of the evolution: we can see how the system has evolved in terms of components or which entities have moved from one group to another.

2.1 Using the results

The results of the analysis produces an application evolution map that is based on the evolution of the entities in the system. For example, a class evolution map represents the evolution of classes and helps to identify for example:

- All the stable classes, methods or groups regarding certain properties: we can track classes did not change their position in an inheritance hierarchy, or classes which have kept their number of instance variables or methods over several versions of system. We could also identify classes which have changed their number of methods by 5%.
- All the source code entities that have disappeared or appeared.
- All the source code entities that have been renamed (we can track a lot of them through the identification of certain properties).
- All the classes that have moved from one subsystem to another one.
- All the subsystems that have been eliminated. In such a case it would also be intersting to see where their components have migrated.
- All the subsystems that have a regular growth.
- All the subsystems that increase and decrease in size over time.

3 Tool Support for Evolution Recovery

We are currently developing MOOSE, a platform that supports the reengineering of object-oriented applications [5]. MOOSE provides facilities for the analysis of multiple versions a system and is the base of MOOSE FINDER, a tool that supports the recovery of software artifact evolution.

3.1 Moose: A language Independent Reengineering Environment

During the FAMOOS project we built an environment called MOOSE to reverse engineer and reengineer object-oriented systems. MOOSE has the following characteristics:

- It supports reengineering of applications developed in different object-oriented languages, as its core model is *language independent* and can be *customized* to incorporate language specific features.
- It is *extensible*. New entities like associations or measurements can be added to the environment.
- It supports reengineering by providing facilities for analyzing and storing multiple models, for refactoring and support for analysis methods such as metrics and inference of source code entity properties.
- Being fully object-oriented, MOOSE provides a complete description of the meta-model entities in terms of objects that are easily parameterized and/or extended.

3.2 Moose Finder: A Query based Approach Recovery of Evolution

MOOSE FINDER allows the definition of queries over several MOOSE models. These queries allow the identification of entities which possess certain properties like type, names, metrics within a single model and between two model or several models.

The queries can be composed into other and more complex queries through aggregation and the use of logical operators.

We explain the composition of such a query in depth in Section 4.

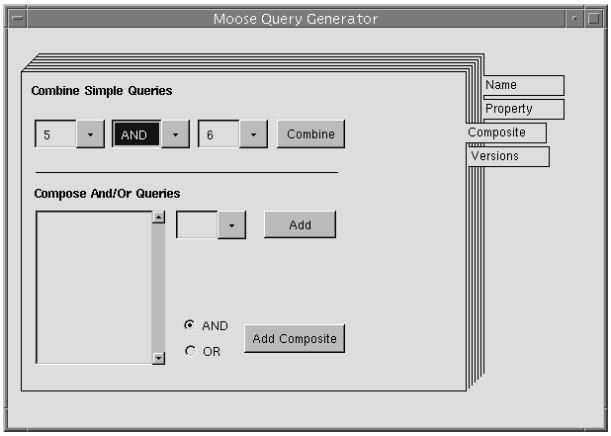

Figure 1. The query composition interface

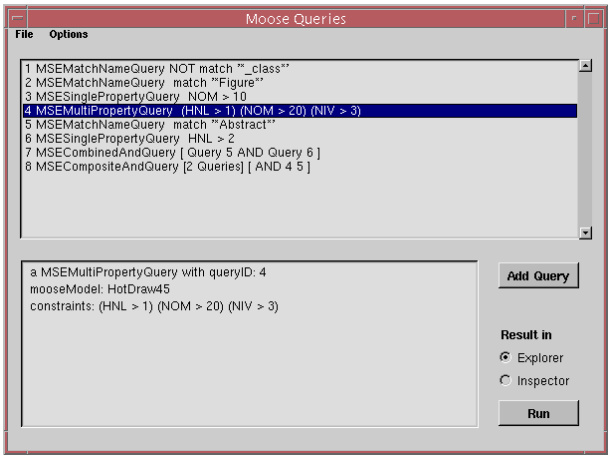

Figure 2. The query repository

3.3 Scalability issues

Calculating simple differences between the versions of a system produces a huge amount of data. The treatment and condensing of this information into a meaningful content is crucial for providing useful information.

In the first experiment we made, we use CODECRAWLER [3, 8] to help us to visualize the results of the queries.

4 Case Study

HotDraw[1] is a two-dimensional graphics framework for structured drawing editors, implemented in Smalltalk by John Brant. You can easily create new graphical objects and special manipulation tools for your drawings. HotDraw also has functionalities to animate drawings. For our case study we used two different versions of Hotdraw: Hotdraw V4.1 (18 Jul 94) for VislualWorks 1.0 and Hotdraw V4.5 (14-Jan-98?) for the actual VisualWorks 3.0.

A First Impression. The queries allow us to identify in one version for example a simple matching of entity names or properties that exceed a defined threshhold. Figure 3 shows all subclasses of the class VisualPart with NOM (number of methods) exceeding 10 and having at least one instance variable.

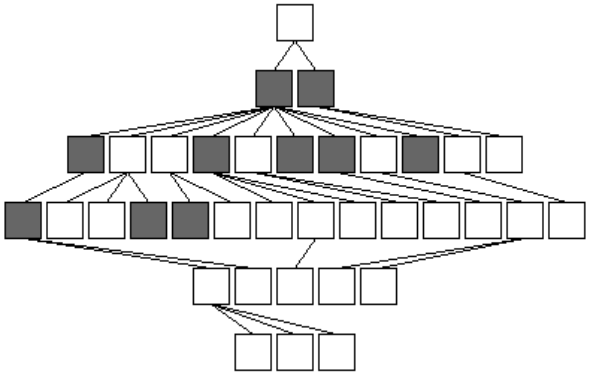

Figure 3. CodeCrawler showing the classes having NOM > 10 and NIV > 0

Getting more complex. More complex queries can be set up to analyze the change of property values over different versions. Comparing the two versions, we first looked at the class names in both versions to see which classes have kept their names in the new version. These classes are shaded in light gray. Figure 4 also shows in black all classes that have changed position in the hierarchy.

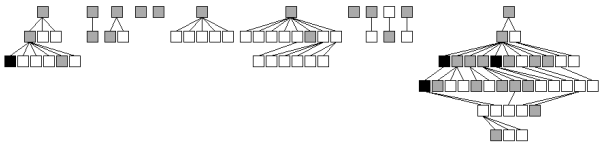

Figure 4. CodeCrawler showing the classes which have changed in their HNL value

Tracking changes. The next example shows a screenshot of class Figure in this hierarchy tree. Classes with have decreased their number of methods in the new release are in lightgray, classes with increased NOM in dark gray. The screenshot shows that the class Figure has a higher number of methods in the new release. Most of the direct subclasses of Figure have a lower number of methods in the new version. This indicates that common responsibility may have been moved up in class Figure.

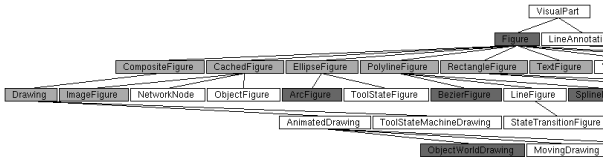

Figure 5. CodeCrawler showing the classes which have changed in their NOM value

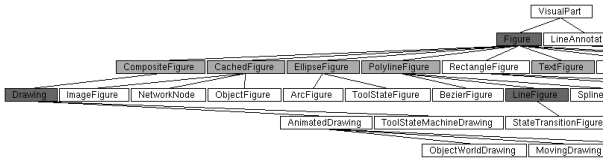

Figure 6. CodeCrawler showing the classes which have changed in their NIV value

Identifying the stable parts. To identify stable parts of the system we look for classes that have not changed their number of methods (NOM) and their number of instance variables (NIV). Figure 7 shows all such classes in HotDraw 4.5. Most of these classes are base classes that are high in the class hierarchy.

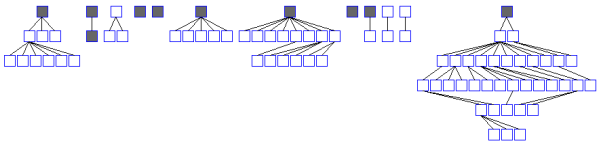

Figure 7. CodeCrawler showing the classes which have not changed in both their NIV and NOM value

Constructing a query. The query for this example can be set up as follows: A query that matches entities over their names in different versions is used to create a composite query. Queries that identify changements of property values (NOM,NIV) between versions are then added as supplemental constraints to the composite query. The composite query finally runs over a default input which is in this case the set of all entities of the first Moose model in the model list of the first query of the composite query. The composite query returns a collection of entities that satisfy all defined constraints. This collection is passed on to CodeCrawler to highlight the respective nodes in a graph.

```
| composite |
composite := MSENameCompositeQuery new.
composite addQuery: (MSEPropertyCompareQuery
createWith: #( #HotDraw41 #HotDraw45) asList
propertyName: #NOM
operator: #=
changeValue: 0).
composite addQuery: (MSEPropertyCompareQuery
createWith: #( #HotDraw41 #HotDraw45) asList
propertyName: #NIV
operator: #=
changeValue: 0).
composite runOnDefault
```

5 Conclusion and Future plans

We have seen that with the facilities provided by MOOSE and the power of expression of our query tool, we are able to identify software entities which are of interest to us regarding their life cycle in the context of the overall evloution of the whole system.

The information which we can gather through these means, can be used to understand the system in terms of its evolution.

Up to now, our tools do not support the definition of groups. In the future we plan to extend MOOSE and its tools to support the definition of such groups, as well as the definition of metrics based on those groups.

Groups as classification entities. A group can be viewed as a recursive entity containing at the leaf level elementary code entities like classes, packages, methods and attributes [7, 4]. A group can be created by extension or by intention, and can be merged with other groups.

Group based Metrics. Metrics are used to compute the size and complexity of software artifacts[2, 6], as well as for quality assessment reasons. A group should provide a way to manipulate and compute the metrics of the entities that compose it. In particular it should also

support the possibility to compute group specific metrics.

References

[1] J. Brant. Hotdraw. Master's thesis, University of Illinois, 1995.

[2] 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.

[3] 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, *WCRE'99 Proceedings (6th Working Conference on Reverse Engineering)*. IEEE, Oct. 1999.

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

[5] S. Ducasse, M. Lanza, and S. Tichelaar. Moose: an extensible language-independent environment for reengineering object-oriented systems. 2000. Submitted to COSET'2000 (International Symposium on Constructing Software Engineering Tools).

[6] N. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. International Thomson Computer Press, London, UK, second edition, 1997.

[7] K. D. Hondt. *A Novel Approach to Architectural Recovery in Evolving Object-Oriented Systems*. Ph.D. thesis, Vrije Universiteit Brussel - Departement of Computer Science - Pleinlaan 2, Brussels - Belgium, Dec. 1998.

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

[9] C. Riva. Visualizing software release histories: The use of color and third dimension. Master's thesis, Politecnico di Milano, Milan, 1998.