Moose: an Agile Reengineering Environment

Stéphane Ducasse
Software Composition Group
University of Berne
Switzerland
ducasse@iam.unibe.ch

Tudor Gîrba
Software Composition Group
University of Berne
Switzerland
girba@iam.unibe.ch

Oscar Nierstrasz
Software Composition Group
University of Berne
Switzerland
oscar@iam.unibe.ch

ABSTRACT

Software systems are complex and difficult to analyze. Reengineering is a complex activity that usually involves combining different techniques and tools. MOOSE is an reengineering environment designed to provide the necessary infrastructure for building new tools and for integrating them. MOOSE centers on a language independent meta-model, and offers services like grouping, querying, navigation, and meta-descriptions. Several tools have been built on top of MOOSE dealing with different aspects of reengineering like: visualization, evolution analysis, semantic analysis, concept analysis or dynamic analysis.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Maintenance—*Restructuring, reverse engineering, and reengineering*

General Terms

Measurement, Design

1. INTRODUCTION

Reengineering is a complex activity usually consisting of a combination of techniques such as: parsing the code, building a model of the code, measuring the code, visualizing, etc. For different techniques we need different tools, but we need these tools to collaborate and complement each other. In the same time we need to not pre-impose the sequence of using these tools and techniques.

We present MOOSE - an agile reengineering environment that allows tools to collaborate. MOOSE is driven by three principles: *extensibility*, *exploration* and *scalability*.

MOOSE is extensible to allow for different tools to implement different analyses. For example, as the different analyses need different meta-models, we built our environment to provide for an extensible meta-model. We accomplish this by providing a meta-description mechanism based on the MOF. Every entity in our meta-model has a corresponding meta-description. A tool can add its own entity to the meta-model along with its description, and the environment interprets the description.

MOOSE enables the exploration because reengineering is not a linear task. For example, each entity in the meta-model has associated a menu, and each tool can register itself to the menu. Thus, no matter where we are in the environment we can invoke a tool that can be applied to the selected entity.

Reengineering can only be studied if we are able to analyze real-life systems. Real-life systems come in different shapes and sizes, therefore we need our tools to scale [6].

Further we talk about MOOSE overall architecture. We briefly describe the underlying meta-model and describe the tools built on top of MOOSE. Before concluding we mention the availability of MOOSE.

2. THE ARCHITECTURE OF MOOSE

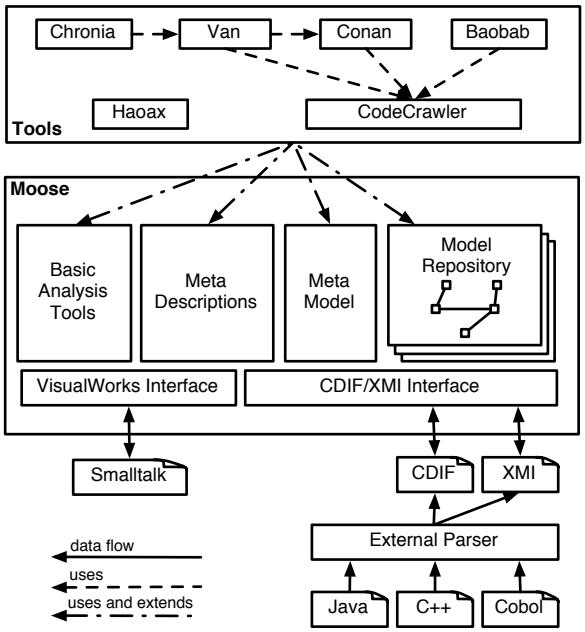

Figure 1: The architecture of Moose.

Moose uses a layered architecture (see Figure 1). Information is transformed from source code into a source code

ESEC-FSE'05, September 5–9, 2005, Lisbon, Portugal.

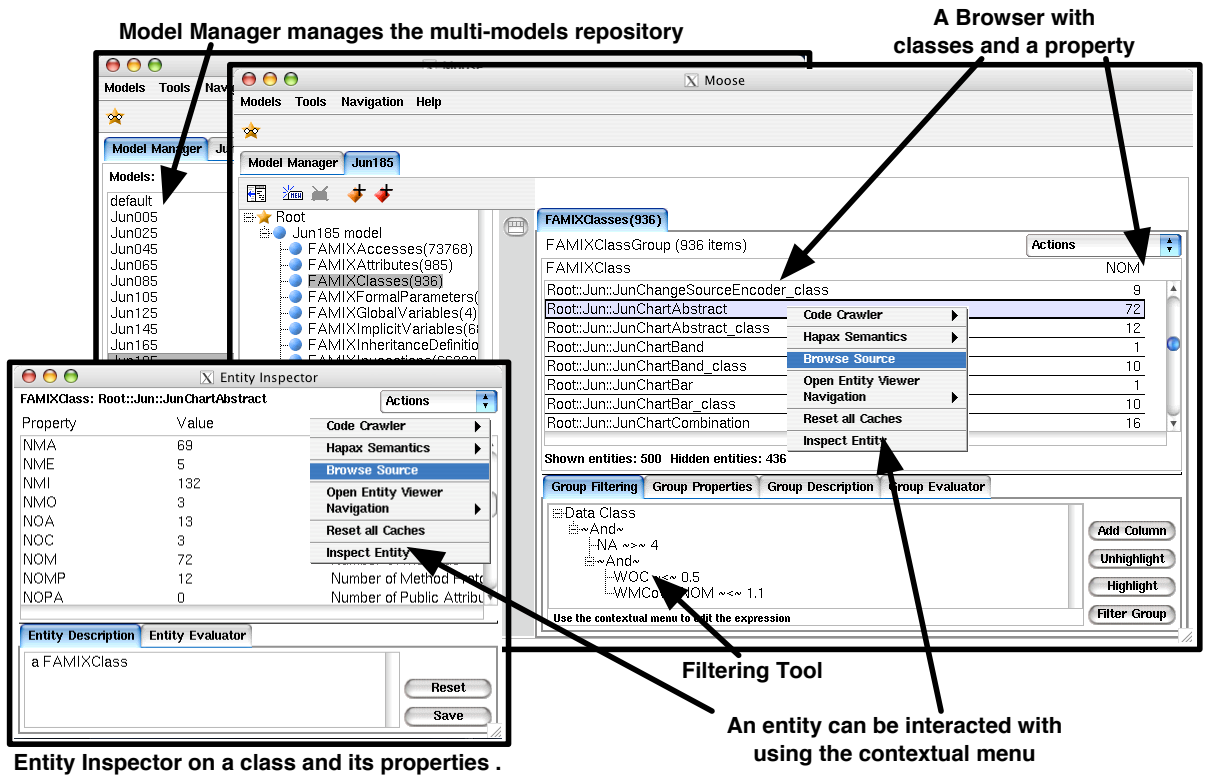

Figure 2: Moose

model. The models are based on the FAMIX language independent meta-model [3]. The information in this model, in the form of entities representing the software artifacts of the target system, can be analyzed, manipulated and used to trigger code transformations by means of refactorings. We will describe the architecture of Moose starting from the bottom.

Export/Import. MOOSE supports multiple languages via the FAMIX meta-model. Source code can be imported into the meta-model in two different ways. In the case of VisualWorks Smalltalk the language in which Moose is implemented models can be directly extracted via the meta-model and the parser of the Smalltalk language.For other source languages Moose provides an import interface for CDIF and XMI files based on our FAMIX meta-model. Over this interface Moose uses external parsers for languages other than Smalltalk. Currently C++, Java, COBOL, and other Smalltalk dialects are supported.

Core. In the center of MOOSE is the FAMIX meta-model. Every model contains entities representing the software artifacts of the target system. Every entity is represented by an object, which allows direct interaction and querying of entities, and consequently an easy way to query and navigate the model. MOOSE can maintain and manipulate several models in memory at the same time via a model repository. Every entity is described by a meta-description, which is then used by the environment to display user interfaces or load/save entities. These meta-descriptions are extensible by other tools and are used by different tools. Examples of the supported meta-descriptions are:

- Navigation. Given an entity we describe what are the associated entities. For example, a class has multiple methods.
- Menu. Every entity has a menu attached to it, and the tools can register menu actions to a particular kind of entity and this action can be triggered from everywhere in the environment.
- Properties. Every entity is annotated with the properties that can be computed on that entity. For example, given a class we can compute its number of methods, or denote whether the class is abstract, etc.
- Load/Save. Every entity can be saved in flat file format (such as CDIF or XMI). This conversion from complex entity with relation-ships into a flat representation is based on the entity meta-description.

MOOSE also provides basic tools that use the are generic by using the meta-descriptions (see Figure 2):

- Browser. With the browser we can navigate the contents of the model.
- Entity Inspector. The Inspector shows all properties of a given en-tity.
- Filtering Tool. It selects all entities that conform to a certain rule specified by an expression. The Filtering Tool is part of the Browser, but it can also be used as a stand-alone tool.

CodeCrawler displaying a System Complexity View.

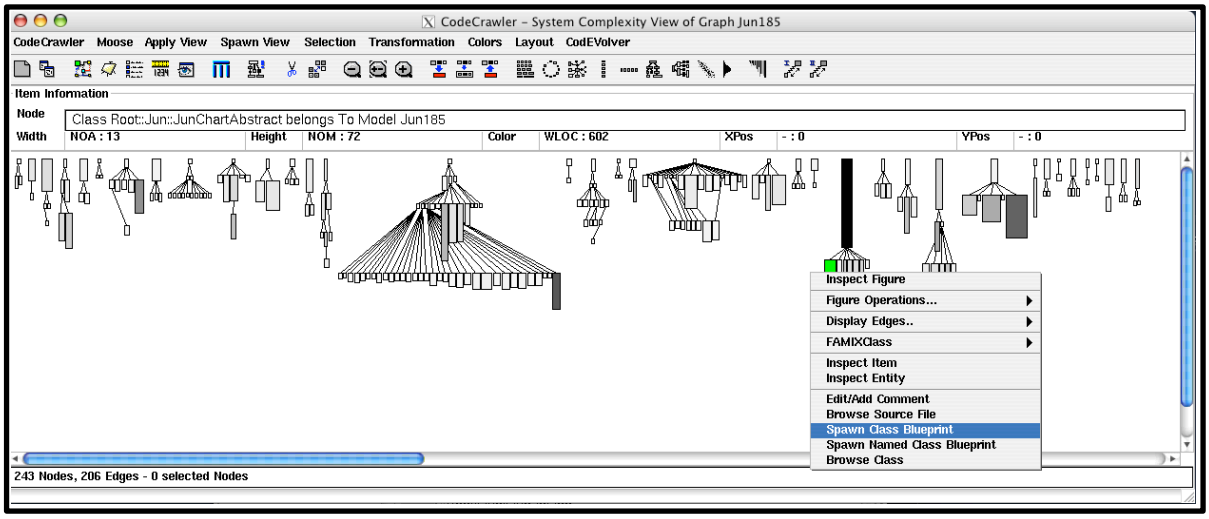

Figure 3: CodeCrawler

3. TOOLS BUILT ON TOP OF MOOSE

CodeCrawler. CodeCrawler is a visualization tool implementing polymetric views [10, 5] which is based on a graph notion where the nodes and edges in the graph can wrap the entities in the model. For example, in Figure 3 we see a screenshot of CodeCrawler displaying a hierarchy of a system called Jun. Although CodeCrawler was first developed as a visualization tool for software systems, in its latest implementation it turned into a general-purpose visualization tool, which can accommodate different needs. For example, in Figure 1 it is shown that CodeCrawler is used by different tools for different visualizations.

ConAn. ConAn is a concept analysis tool that manipulates concepts as first class entities [1, 2]. Its target is to detect different kinds of patterns in the model based on combining elements and properties. ConAn uses CodeCrawler for visualization purposes and supports analyses like: X-Ray views for understanding the internal of classes, identification of recurring code patterns, and views for hierarchy understanding.

Van. Van is a tool for analyzing the evolution of systems. At its core, it defines the Hismo meta-model which is based on the notion of history [4]. Hismo is independent from FAMIX, but it works closely with it. Based on Hismo different evolution analyses are defined. For example, in Figure 4 we show how Van uses CodeCrawler to display the evolution of the class hierarchies in the Jun system [8]. In the same figure we see two other tools: the Diagram Viewer and the History Inspector. Van offers other analyses like: history measurements, changes characterization, hidden dependencies detection based on change information, past refactorings detection.

Chronia. Chronia is a tool that bridges Van with versioning systems like CVS. It enables analyses of how developers change the system [7].

TraceScraper. TraceScraper analyzes the dynamic traces from different perspectives. For example it offers measurements and visualizations for dynamic traces [9]. TraceScraper also uses Van to analyze the evolution of dynamic traces.

Baobab. Baobab is a tool to understand dependencies between modules. It extends FAMIX with the notion of dependency between different parts of the system and provides various measurements for these dependencies.

Hapax. Hapax is a semantic analysis tool. It makes use of the comments and names of the identifiers from the code to recover the domain information. It also offers clustering of different parts of the system based on how they use the same terms.

4. MOOSE AVAILABILITY

Moose is completely implemented in Smalltalk under the BSD license: it is free and open source software. MOOSE runs on every major platform (Windows, Mac OS, Linux, Unix). MOOSE is freely available for download. The current webpage of MOOSE is located at:

http://www.iam.unibe.ch/ scg/Research/Moose/.

Moreover, MOOSE is also available as free goodie on the VisualWorks Smalltalk CD, a professional, commercial development environment developed and sold by the Cincom company which also exists in a non-commercial version freely available for download at:

http://www.cincomsmalltalk.com/

5. CONCLUSIONS

Reverse engineering is a complex task and requires combining different techniques and tools. We presented Moose, a reengineering environment designed to be: (i) extensible - to support unexpected needs, (ii) exploratory - to allow for flexible sequence of actions, and (iii) scalable, to cope with large systems.

On top of Moose several tools were built, each of them having its own focus: dependency analysis, visualization, concept analysis, version analysis, semantic analysis. These

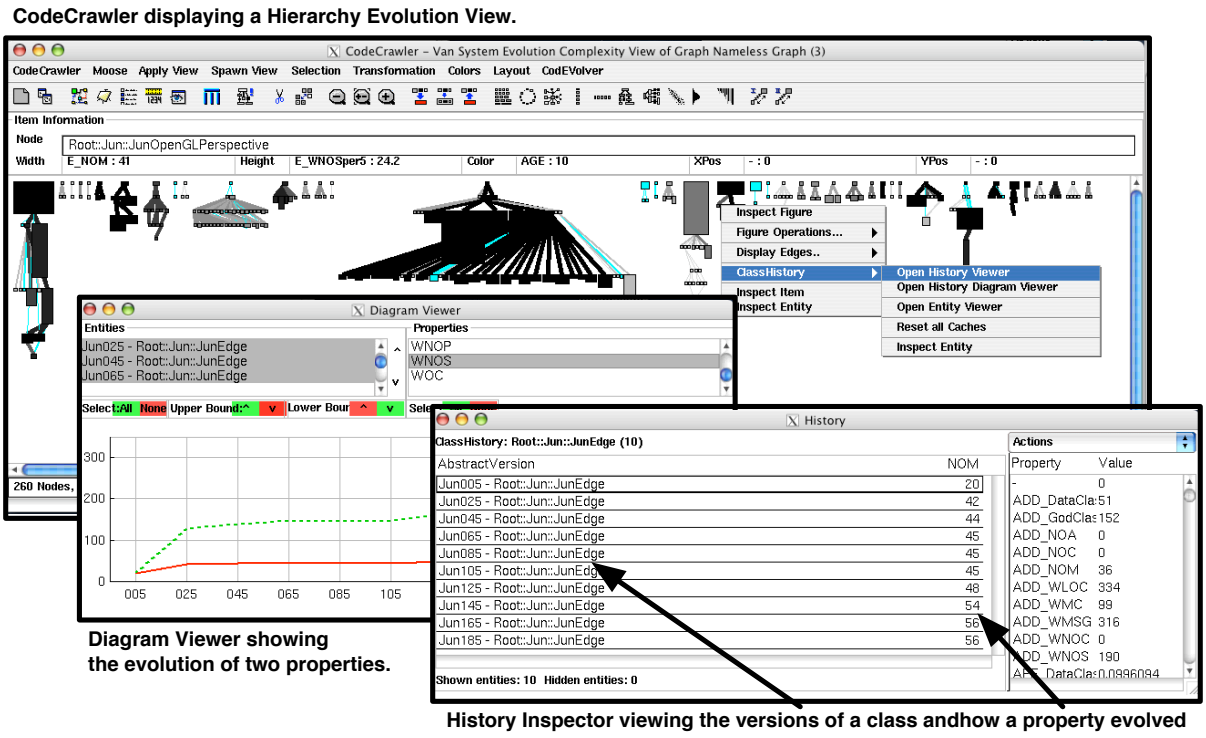

Figure 4: Van and CodeCrawler

tools extend Moose and collaborate with each other using meta-descriptions.

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project and "RECAST: Evolution of Object-Oriented Applications" (SNF Project No. 620-066077, Sept. 2002 - Aug. 2006).

6. REFERENCES

[1] G. Arévalo, F. Buchli, and O. Nierstrasz. Detecting implicit collaboration patterns. In *Proceedings of WCRE '04 (11th Working Conference on Reverse Engineering)*, pages 122–131. IEEE Computer Society Press, Nov. 2004.

[2] G. Arévalo, S. Ducasse, and O. Nierstrasz. Discovering unanticipated dependency schemas in class hierarchies. In *Proceedings of CSMR '05 (9th European Conference on Software Maintenance and Reengineering)*, pages 62–71. IEEE Computer Society Press, Mar. 2005.

[3] S. Demeyer, S. Tichelaar, and S. Ducasse. FAMIX 2.1 — The FAMOOS Information Exchange Model. Technical report, University of Bern, 2001.

[4] S. Ducasse, T. Gîrba, and J.-M. Favre. Modeling software evolution by treating history as a first class entity. In *Workshop on Software Evolution Through Transformation (SETra 2004)*, pages 71–82, 2004.

[5] S. Ducasse and M. Lanza. The class blueprint: Visually supporting the understanding of classes. *IEEE Transactions on Software Engineering*, 31(1):75–90, 2005.

[6] S. Ducasse and S. Tichelaar. Dimensions of reengineering environment infrastructures. *International Journal on Software Maintenance: Research and Practice*, 15(5):345–373, Oct. 2003.

[7] T. Gîrba, A. Kuhn, M. Seeberger, and S. Ducasse. How Developers Drive Software Evolution. In *Proceedings of International Workshop on Principles of Software Evolution (IWPSE)*. IEEE Computer Society Press, 2005. to appear.

[8] T. Gîrba, M. Lanza, and S. Ducasse. Characterizing the evolution of class hierarchies. In *Proceedings of European Conference on Software Maintenance (CSMR 2005)*, 2005.

[9] O. Greevy and S. Ducasse. Correlating features and code using a compact two-sided trace analysis approach. In *Proceedings of CSMR 2005 (9th European Conference on Software Maintenance and Reengineering*. IEEE Computer Society Press, 2005.

[10] M. Lanza and S. Ducasse. Polymetric views — a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29(9):782–795, Sept. 2003.