MOOSE: an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems

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

Abstract

Surprising as it may seem, many of the early adopters of the object-oriented paradigm already face a number of problems typically encountered in large-scale legacy systems. The reengineering of those systems often poses problems because of the considerable size and complexity of such systems. In the context of the FAMOOS project we have developed a language independent environment called Moose which can deal with that complexity. This paper describes the architecture of Moose, the tools which have been developed around it and the industrial experiences we have obtained.

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

1 Introduction

Legacy systems are not limited to the procedural paradigm or languages such as COBOL. Although the object-oriented paradigm promised increased flexibility of systems and the ease in their evolution, even these systems get hard to maintain over time and need to be adapted to new requirements. The goal of the FAMOOS Esprit project was to support the evolution of such object-oriented legacy systems towards frameworks [6].

During the FAMOOS project we built a tool environment called MOOSE to reverse engineer and reengineer object-oriented systems. It consists of a repository to store models of software systems, and provides query and navigation facilities. Models consist of entities representing software artifacts such as classes, methods, etc. MOOSE has the following characteristics:

- It supports reengineering of applications developed in different object-oriented languages, as its core model is *language independent* which, if needed, can be *customized* to incorporate language specific features.
- It is *extensible*. New entities like measurements or special-purpose relationships can be added to the environment.
- It supports reengineering by providing facilities for analyzing and storing multiple models, for refactoring and by providing support for analysis methods such as metrics and the inference of properties of source code entities.
- Its implementation 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.

These properties make MOOSE an ideal foundation for reengineering tools [3].

The outline of this paper is the following: Before presenting the specific aspects of MOOSE, we list the main characteristics that we expect from a reengineering environment. After presenting the architecture of MOOSE, we give an overview of its underlying meta-model and interchange format. We present how a modelled system can be navigated and queried. Then we show how MOOSE supports code refactorings. To give a more dynamic perception of MOOSE we show a typical use in the form of a short scenario. Finally we evaluate the environment regarding the requirements we previously listed and conclude.

2 Requirements for a Reengineering Environment

Based on our experiences and on the requirements reported in the literature [12, 8, 9], these are our main requirements for a reengineering environment:

Extensible. An environment for reverse engineering and reengineering should be extensible in many aspects:

- The meta-model should be able to represent and manipulate entities other than the ones directly extracted from the source code (e.g. measurements, associations, relationships, etc.).
- To support reengineering in the context of software evolution the environment should be able to handle several source code models simultaneously.
- It should be able to use and combine information from various sources, for instance the inclusion of tool-specific information such as run-time information, metric information, graph layout information, etc.
- The environment should be able to operate with external tools like graph drawing tools, diagrammers (e.g. Rational Rose) and parsers.

Exploratory. The exploratory nature of reverse engineering and reengineering demands that a reengineering environment does not impose rigid sequences of activities. The environment should be able to present the source code entities in many views, both textual and graphical, in little time. It should be possible to perform several types of actions on the views the tools provide such as zooming, switching between different abstraction levels, deleting entities from views, grouping entities into logical clusters, etc. The environment should as well provide a way to easily access and query the entities contained in a model. To minimize the distance between the representation of an entity and the actual entity in the source code, an environment should provide every entity with a direct linkage to its source code. A secondary requirement in this context is the possibility to maintain a history of all steps performed by the reengineer and preferably allow him to return to earlier states in the reengineering process.

Scalable. As legacy systems tend to be huge, an environment should be scalable in terms of the number of entities being represented, i.e. at any level of granularity the environment should provide meaningful information. An additional requirement in this context is the actual performance of such an environment. It should be possible to handle a legacy system of any size without incurring long latency times.

In addition to these general requirements, the context of our work [6] forces us to have an environment that is able to support multiple languages.

3 Architecture

MOOSE uses a layered architecture (see Figure 1). Information is transformed from source code into a source code model. The models are based on the FAMIX meta-model [4, 5] which is described in section 4. 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.

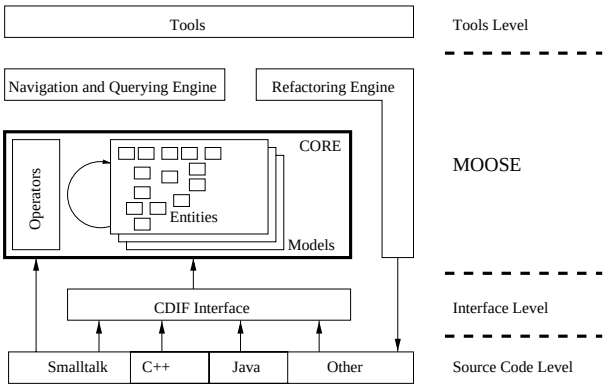

Figure 1. Architecture of Moose.

- *Extraction/Import.* MOOSE supports multiple languages. Source code can be imported into the meta-model in two different ways:
  1. In the case of VisualWorks Smalltalk – the language in which MOOSE is implemented – sources can be directly extracted via the meta-model of the SMALLTALK language.
  2. For other source languages MOOSE provides an import interface for CDIF files based on our FAMIX meta-model. CDIF is an industry-standard interchange format which enables exchanging models via files or streams. Over this interface MOOSE uses external parsers for source languages other than SMALLTALK. Currently C++, JAVA, ADA and other SMALLTALK dialects are supported.
- *Storage and Tools.* The models are stored in memory. 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 a whole model. MOOSE can maintain and access several models in memory at the same time.

  Additionally the core of MOOSE contains the following functionality:

  – *Operators.* Operators can be run on a model to compute additional information regarding the

software entities. For example, metrics can be computed and associated with the software entities, or entities can be annotated with additional information such as inferred type information, analysis of the polymorphic calls, etc. Basically any kind of information can be added to an entity.

  - *Navigation facilities.* On top of the MOOSE core we have included querying and navigation support. This support is discussed in section 5.
  - *Refactoring Engine.* The MOOSE REFACTORING ENGINE defines language-independent refactorings. The analysis for a code refactoring is based on model information. The code manipulation which a refactoring entails, is being handled by language-specific front-ends. Section 6 describes the engine in more detail.

- *Tools Layer.* The functionality which is provided by MOOSE can be used by tools. This is represented by the top layer of figure 1. Some examples of tools based on MOOSE are described in section 7.

4 A Language Independent Meta-model

MOOSE is based on the FAMIX meta-model [4, 5]. FAMIX provides for a language-independent representation of object-oriented sources and contains the required information for the reengineering tasks performed by our tools. It is *language independent*, because we need to work with legacy systems in different implementation languages (C++, JAVA, SMALLTALK, ADA). And it is *extensible*: since we cannot know in advance all information that is needed in future tools, and since for some reengineering problems tools might need to work with language-specific information (e.g. to analyse include hierarchies in C++), we allow for language plug-ins that extend the model with language-specific features. Next to that, we allow tool plug-ins to extend the model to store, for instance, analysis results or layout information for graphs. Figure 2 shows the core of the FAMIX model. It consists of the main object-oriented entities, namely Class, Method and Attribute. In addition there are the associations InheritanceDefinition, Invocation and Access. An Invocation represents a Method calling another Method and an Access represents a Method accessing an Attribute. These abstractions are needed for reengineering tasks such as dependency analysis, metrics computation and reorganisation operations. The complete model consists of much more information, i.e. more entities such as functions and formal parameters, and additional relevant information for every entity. The model does not contain any source code. The complete specification of the model can be found in [5].

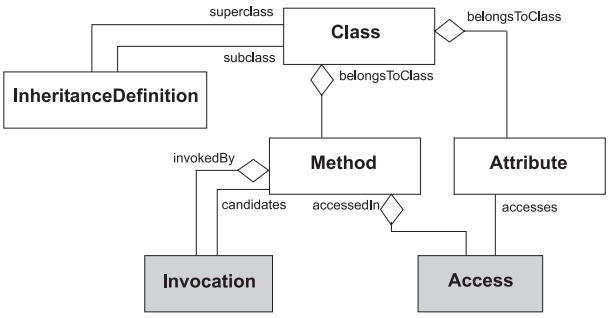

Figure 2. Core of the FAMIX model.

Information exchange with CDIF

To exchange FAMIX-based information between different tools we have adopted CDIF [2] . CDIF is an industrial standard for transferring models created with different tools. The main reasons for adopting CDIF are, that firstly it is an industry standard, and secondly it has a standard plain text encoding which tackles the requirements of convenient querying and human readability. Next to that the CDIF framework supports the extensibility we need to define our model and plug-ins. As shown in Figure 1 we use CDIF to import FAMIX-based information about systems written in JAVA, C++ and other languages. The information is produced by external parsers such as SNiFF+ [15, 16]. Next to parsers we also have integrations with external environments such as the Nokia Reengineering Environment [6].

5 Navigation and Querying

One of the challenges when dealing with complex meta-models is how to support their navigation and facilitate easy access to specific entities. In the following subsections we present two different ways of querying and inspecting source code models in MOOSE.

5.1 Programming Queries

The fact that the meta-model in MOOSE is fully object-oriented together with the facilities offered by the Smalltalk environment, it is simple to directly query a model in MOOSE. We show two examples. The first query returns all the methods accessing the attribute name of the class Node.

```
(MSEModel currentModel
   entityWithName: #'Node.name')
      accessedByCollect:
         [ :each | MSEModel currentModel
             entityWithName: each accessedIn ]
```

The second query select all the classes that have more than 10 descendants.

```
MSEModel currentModel allClassesSelect:
  [ :each | each hasProperties and:
     [ (each hasPropertyNamed: #WNOC) ifTrue:
        [(each getNamedPropertyAt: #WNOC) > 10]]]
```

Note that these queries resemble SQL queries on model information stored in a database [10]

5.2 Querying using the MOOSE EXPLORER

Reengineering large systems brings up the problem of how to navigate large amounts of complex information. Well-known solutions are code browsers such as the Smalltalk one, which have been sufficient to support code browsing, editing and navigating a system by the way of senders and implementers. However, for reengineering these approaches are not sufficient because:

- The number of potentially interesting entities and their interrelationships is too large. A typical system can have several hundreds of classes which contain in turn several thousands of methods, etc.
- All entities need to be navigable in a *uniform way*.
  - In the context of reengineering no entity is predominant. For example, attribute accesses can be extremely important to analysis methods but in other cases completely irrelevant.
  - In presence of an extensible meta-model, the navigation schema should take into account the fact that new entities and relationships can be added and should be navigable as well.

MOOSE EXPLORER proposes an uniform way to represent model information (see figure 3). All entities, relationships and newly added entities can be browsed in the same way. From top to bottom, the first pane represents a current set of selected entities. Here we see all the attributes of the current model. The bottom left pane represents all the possible ways to access other entities from the currently selected ones. Here, from the selected attribute name of the class Node the methods that access it are requested. The resulting entities are displayed in the right bottom pane and can then be further browsed. 'Diving' into the resulting entities puts them as the current selection in the top pane again, which allows for further navigation through the model.

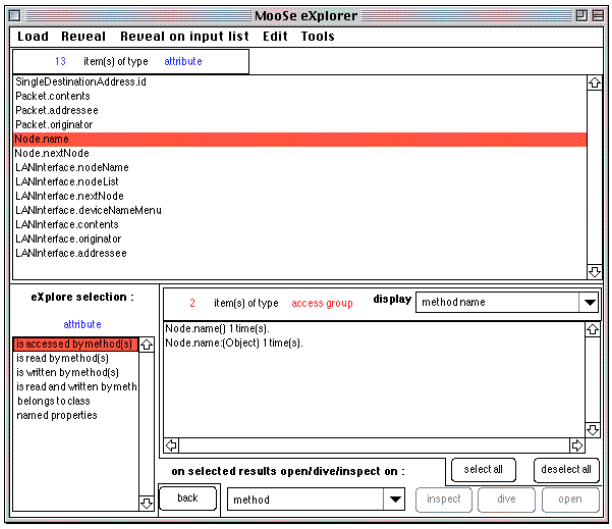

Figure 3. MOOSE EXPLORER: navigating a meta-model in an uniform way.

6 Refactoring

The MOOSE REFACTORING ENGINE closes the reengineering circle. While the MOOSE core provides for a repository and querying and navigation support, the MOOSE REFACTORING ENGINE provides support for doing actual code changes. Refactoring [7] is about making changes to code to improve its structure, simplicity, flexibility, understandability or performance [1] without changing the external behaviour of the system. The MOOSE REFACTORING ENGINE provides functionality similar to the Refactoring Browser [14] for Smalltalk, but for multiple implementation languages.

The MOOSE REFACTORING ENGINE does virtually all of the analysis — needed to check the applicability of a refactoring and to see what exactly has to be changed — using the language-independent FAMIX model. The language dependence can be kept on a minimal level, because firstly the refactorings are very similar for the different languages, and secondly, FAMIX is designed to capture these commonalities as much as possible. For instance, FAMIX supports multiple inheritance, which covers Smalltalk's single inheritance, C++'s multiple inheritance and Java's classes and interfaces. Language extensions (see section 4) cover most of the remaining issues, for instance, to figure out if a class entity in MOOSE represents a class or an interface in Java.

Of course, changing the code is language-specific. For every supported language a component has to be provided that performs the actual code changes directly on the source code. Currently the MOOSE REFACTORING ENGINE is a prototype with language front-ends for Smalltalk and Java. For Smalltalk we use the Refactoring Browser [14] to change the code, and for Java we currently use a text-based approach based on regular expressions. Although the text-based approach is more powerful than we initially expected, we plan to move to an abstract syntax tree based approach in the future.

A set of language-independent refactorings together with

the analysis support of MOOSE itself provides for a powerful combination of using analysis to drive (semi-)automated code improvements. This is illustrated by the scenario in section 8.

7 Foundation for other tools

MOOSE serves as a foundation for other tools. It acts as the central repository and provides services such as metric computation and refactorings to the reengineering tools built on top of MOOSE. At this point in time the following tools have been developed:

- CODECRAWLER supports reverse engineering through the combination of metrics and visualization [11, 3] (see Figure 4). Through simple visualizations which make extensive use of metrics, it enables the user to gain insights in large systems in a short time. CODECRAWLER is a tool which works best when we approach a new system and need quick insights to get information on how to proceed. CODECRAWLER has been successfully tested on several industrial case studies.
- GAUDI [13] combines dynamic with static information. It supports an iterative approach creating views which can be incrementally refined by extending and refining queries on the repository, while focusing on dynamic information.

The following tools are currently under development:

- The MOOSE REVEALER is used to detect entities which fulfill certain properties. At the basic level these may be abstract classes, empty methods, etc. At a higher level of complexity it addresses design problems such as unused attributes or big classes which could be split by identifying clusters of methods or attributes.
- The MOOSE FINDER is a tool that allows to compose queries based on different criteria like entity type, properties or relationships, etc. A simple query finds entities that meet certain conditions. Such a query can in turn be combined with other queries to express more complex ones. The MOOSE FINDER is currently being extended in order to handle multiple models in the context of software evolution.
- The MOOSE DESIGN FILTER can use the meta-model information to communicate with Rational Rose, in order to generate design views on the code.

Not only does MOOSE serve as the base for all those applications providing them a number of functionalities like the metrics framework, the repository also serves as common interface between those tools.

Except for providing the foundation for our own tools, MOOSE also interfaces with external tools. One example is the Nokia Reengineering Environment [6].

8 Scenario

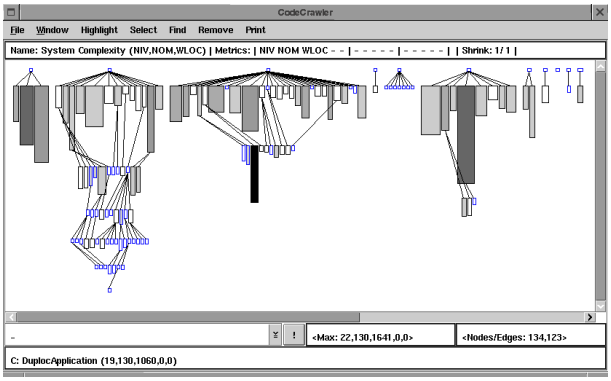

Figure 4. CODECRAWLER showing an inheritance tree view of a system. The width of the nodes represents the number of methods, the height represents the number of instance variables.

In this section we present a typical scenario of how the MOOSE environment can be used. It shows three different tools based on MOOSE, and their interaction to detect a problem, analyze it and finally resolve it by changing the code. Note that the scenario is partly hypothetical in the sense that the MOOSE REVEALER is in its early stages of development and that its capabilities are not yet tested in real world cases.

We start with CODECRAWLER. Figure 4 shows a screenshot of this tool. In this case the bigger boxes denote bigger classes in the inheritance hierarchy. The classes are bigger, in terms of number of methods (x-axis) and number of attributes (y-axis). In this way CODECRAWLER points us to possible problems in a software system, as big classes might imply a wrong distribution of responsibilities. We will focus on the tall gray class on the right side of the drawing.

In the second phase we use the MOOSE REVEALER to analyze our possible problem. In this case the MOOSE REVEALER finds out that the class can be split in two pieces, because it finds two groups of methods that have a strong internal cohesion, but do not really depend on the other group. The MOOSE REVEALER proposes the user to split the class in a superclass and a subclass, both with one group of methods. If the user decides that the proposed solution

is a good idea, he or she can trigger the MOOSE REFACTORING ENGINE to implement the proposed change. The MOOSE REFACTORING ENGINE initiates a series of refactorings: it creates a new superclass, and pulls up the methods of one of the groups into this new class, while updating all the references to these methods and checking if the changes do not have any unwanted effect on the system (the changes should be behaviour preserving).

The scenario shows how powerful the combination of metrics, visualization, FAMIX-based analysis and refactorings can be. Of course, not every big class can be nicely split (and quite often there is a good reason to have a specific big class). Currently we are researching how far we can get in finding possible solutions to potential problems. In the end, however, only the developer can decide if a potential problem is really a problem and if the proposed solution is indeed a good and viable solution.

The fact that most of the analysis is based on the language-independent representation of software in MOOSE, makes the scenario applicable for every language supported by MOOSE and the MOOSE REFACTORING ENGINE.

9 Validation and Evaluation

MOOSE and its tools have been validated in a few industrial experiences. The idea was that members of our team went to work on the industrial applications in a 'let's see what they can tell us about our system' way. There was no training of the developers with our tools. The common point about those experiences was that the subject systems were of considerable size and that there was a narrow time constraint for all experiences we describe below:

1. A very large legacy system written in C++. The size of the system was of 1.2 million lines of code in more than 2300 classes. We had four days to obtain results.

2. An medium-sized system written in both C++ and JAVA. The system consisted of about 120,000 lines of code in about 400 classes. The time frame was again four days.

3. A large system written in SMALLTALK. The system consisted of about 600,000 lines of code in more than 2500 classes. This time we had less than three days to obtain results. Parsing and storing the complete system took less than 5 minutes on a PC Pentium III 500Hz.

The fact that all the industrial case studies where under extreme time pressure lead us to mainly get an understanding of the system and produce overviews [3]. We were also able to point out potential design problems and on the smallest case study we even had the time to propose a possible redesign of the system. Taking the time constraints into account, we obtained very satisfying results. Most of the time, the (often initially sceptical) developers were surprised to learn some unknown aspects of their system. On the other hand, they typically knew already about many problems we found.

We learnt that, in addition to the views provided by our tools, code browsing was needed to get a better understanding of specific parts of the applications. Combining metrics, graphical analysis and code browsing proved to be an successful approach to get the results described above. The obvious conclusion is that tools are necessary but not sufficient.

Memory issues

Up to now we did not have problems regarding the number of entities we loaded into the code repository. The maximum number of entities we loaded was around 250,000 in the third industrial case, which was the limit on the available computers. Surpassing 300,000 entities made the environment swap information to the hard disk and back. The code repository might run into problems with multi-million line projects. For that reason we have designed the code repository to support a possible database mapping easily. In that sense the design of the code repository is more database-oriented (with, for instance, a global entity manager than object-oriented.

In addition, the following considerations have to be taken into account when speaking about memory problems. First, the amount of available memory on the used computer system is, of course, an important factor. Secondly, we have never even tried to optimize our environment neither in access speed nor in memory consumption, because so far we did not really have problems in these areas. Therefore, there is some room for improvement, would it be needed in the future. A third aspect is that tools that make use of the repository need some memory of their own as well. For instance, CODECRAWLER needs to create a lot of additional objects (representing nodes and edges) for the purpose of visualization.

The requirements revisited

In section 2, we listed three properties which a reengineering environment should possess. We will now list those properties and discuss how MOOSE evaluates in that context. In section 2 we stated that such an environment should be:

1. **Extensible.** The extensibility of MOOSE is inherent to the extensibility of its meta-model. Its design allows for extensions for language-specific features and

for tool-specific information. We have already built several tools which use the functionalities offered by MOOSE.

2. **Exploratory.** MOOSE is an object-oriented framework and offers as such a great deal of possible interactions with the represented entities. We implemented several ways to handle and manipulate entities contained in a model, as we have described in the previous sections.

3. **Scalable.** The industrial case studies presented at the beginning of this section have proved that MOOSE can deal with large systems in a satisfactory way: we have been able to parse and load large systems in a short time. Since we keep all entities in memory we have fast access times to the model itself. So far we have not encountered memory problems: the largest system loaded contained more than 250,000 entities and could still be held completely in memory without any notable performance penalties.

10 Conclusion and Future Work

In this paper we have presented the MOOSE reengineering environment. First, we have defined our requirements for such an environment and afterwards we have introduced the architecture of MOOSE, its meta-model and the different tools that are based on it.

The facilities of MOOSE for storing, querying and navigating information and its extensibility make it an ideal foundation for other tools, as shown by tools such as GAUDI and CODECRAWLER. Next to that, the environment has proven its scalability and usability in an industrial setting.

Future work includes further development of our MOOSE-based tools, using them to explore in more detail topics such as design extraction, steering of refactorings based on code duplication detection or other kinds of analysis, and evaluating system evolution. Furthermore, we are working on providing extended support for fine-grained analysis by means of composed queries. Next to that, we plan to introduce *classifications* or *groupings* of entities to support higher level views of systems.

Acknowledgements

This work has been funded by the Swiss Government under Project no. NFS-2000-46947.96 and BBW-96.0015 as well as by the European Union under the ESPRIT programme Project no. 21975 (FAMOOS).

References

[1] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[2] C. T. Committee. Cdif framework for modeling and extensibility. Technical Report EIA/IS-107, Electronic Industries Association, Jan. 1994. See http://www.cdif.org/.

[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. Demeyer, S. Ducasse, and S. Tichelaar. Why unified is not universal. UML shortcomings for coping with round-trip engineering. In B. Rumpe, editor, *Proceedings UML'99 (The Second International Conference on The Unified Modeling Language)*, LNCS 1723, Kaiserslautern, Germany, Oct. 1999. Springer-Verlag.

[5] S. Demeyer, S. Tichelaar, and P. Steyaert. FAMIX 2.0 - the FAMOOS information exchange model. technical report, University of Berne, Aug. 1999.

[6] 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.

[7] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[8] J.-L. Hainaut, V. Englebert, J. Henrard, J.-M. Hick, and D. Roland. Database reverse engineering: From requirements to CARE tools. *Automated Software Engineering*, 3(1-2), June 1996.

[9] R. Kazman. Tool support for architecture analysis and design, 1996. Proceedings of Workshop (ISAW-2) joint Sigsoft.

[10] R. Kazman and S. J. Carriere. Playing detective: Reconstructing software architecture from available evidence. *Automated Software Engineering*, Apr. 1999.

[11] M. Lanza. *Combining Metrics and Graphs for Object Oriented Reverse Engineering*. Diploma thesis, University of Bern, Oct. 1999.

[12] G. Murphy and D. Notkin. Reengineering with reflexion models: A case study. *IEEE Computer*, 8:29–36.

[13] T. Richner and S. Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In H. Yang and L. White, editors, *Proceedings ICSM'99 (International Conference on Software Maintenance)*, pages 13–22. IEEE, Sept. 1999.

[14] D. Roberts, J. Brant, and R. E. Johnson. A refactoring tool for smalltalk. *Theory and Practice of Object Systems (TAPOS)*, 3(4):253–263, 1997.

[15] TakeFive Software GmbH. *SNiFF+*, 1996.

[16] S. Tichelaar and S. Demeyer. Sniff+ talks to rational rose – interoperability using a common exchange model. In *SNiFF+ User's Conference*, Jan. 1999. Also appeared in the "Proceedings of the ESEC/FSE'99 Workshop on Object-Oriented Re-engineering (WOOR'99)" – Technical Report of the Technical University of Vienna (TUV-1841-99-13).