The Moose Reengineering Environment

Stéphane Ducasse, Michele Lanza and Sander Tichelaar
Software Composition Group, University of Berne
{ducasse,lanza,tichel}@iam.unibe.ch

1. Introduction

This article presents the Moose Reengineering Environment, a language independent tool environment to reverse engineer, i.e., understand, and reengineer software systems, as well as the tools which have been developed around it and the experience, both academic and industrial, we have obtained.

Moose is implemented in VisualWorks 3.0/Envy 4.0. It consists of a repository to store models of software systems, and provides facilities to analyse, query and navigate them. 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 analysing and storing multiple models, for refactoring and by providing support for analysis methods such as metrics and the inference of properties of source code elements.
- With its *fully object-oriented implementation*, Moose provides a complete description of the metamodel elements in terms of objects that are easily parameterised, extended or manipulated.

These properties make Moose an ideal foundation for reengineering tools such as CodeCrawler [DDL99] or Supremo [KN01]. These, and other, tools are discussed in section 8.

Moose is based on the FAMIX metamodel [DDT99] [DTD01]. 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). It is *extensible*, since we cannot know in advance all information that is needed in

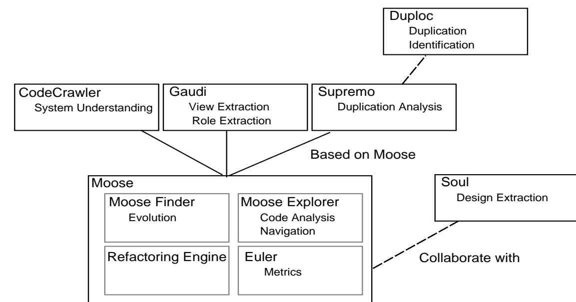

Figure 1 Moose collaborations with other tools

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++). Therefore 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.

The outline of this article is the following. We start by listing the main characteristics that we expect from a reengineering environment. We then present Moose in depth: first we discuss its overall architecture, followed by a detailed presentation of the different parts. We present some tools that make use of Moose and we finish with a validation of Moose against the requirements we have set and a conclusion.

2. Requirements for a Reengineering Environment

Based on our experiences and on the requirements reported in the literature [MN97] [HEH+96] [Kaz96], these are our main requirements for a reengineering environment:

- **Support for reengineering tasks.** An obvious requirement which determines the focus of the tool. It determines the information to store and which services the environment provides. Typical reengineering tasks are metrics, grouping, visualisation and refactoring.
- **Extensible.** An environment for reverse engineering and reengineering should be extensible in many aspects:

- The metamodel 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 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 long latency times.
- **Information Exchange and Tool Integration.** A reengineering effort is typically a cooperation of a group of specialised tools [DDT99]. Therefore, a reengineering environment needs to be able to integrate with external tools, either by exchanging information or ideally by providing a platform for tools for runtime integration.

In addition to these general requirements, the context of the FAMOOS project [DD99] forced us to have an environment that is able to support multiple languages.

3. Architecture

Moose has a layered architecture (see figure 2). We describe the architecture starting from the bottom.

Import/Export Framework. There are several ways to import information about software systems into Moose.

- In the case of VisualWorks Smalltalk — the language in which Moose is implemented — sources can be directly extracted via the metamodel of the Smalltalk language or via the built-in parser.
- For other source languages Moose provides an import interface for CDIF or XMI files based on the FAMIX metamodel. CDIF [Com94] and XMI [OMG98] are industry-standard interchange formats for exchanging models via files or streams. Over this interface Moose uses external parsers for source languages other than VisualWorks Smalltalk. Currently C++, Java, Ada and some other Smalltalk dialects are supported. Information exchange is discussed in more detail in section 8

Repository and Model Management. Information is transformed from source code into a source code model. Moose can maintain and access several models in memory at the same time. The models are based on the FAMIX metamodel [DTD01]. Every model contains elements representing the software artifacts of the target system. The information in this model can then be analysed, manipulated and used to trigger code transformations by means of refactorings.

Services. Moose provides several services that make the life of a reengineer easier.

- *Querying and Navigation.* Every element in a model is represented by an object, which allows direct interaction of elements, and consequently an easy way to query and navigate a model. The query and navigation support is discussed in detail in section 4
- *Metrics and other Analysis support.* Moose's analysis services are mostly implemented as operators that can be run over a model to compute additional information regarding the software elements. For example, metrics can be computed and associated with the software entities, entities can be annotated with additional information such as inferred type information, analysis of the polymorphic invocations, etc.
- *Grouping.* Moose has grouping mechanisms, with which it is easy to group several entities into one group entity, which is treated from then on as an entity itself. This is useful when a reengineer wants to reduce the amount of information by looking at the subject system from a higher level of abstraction, i.e., instead of looking at several hundreds of classes, he can group them

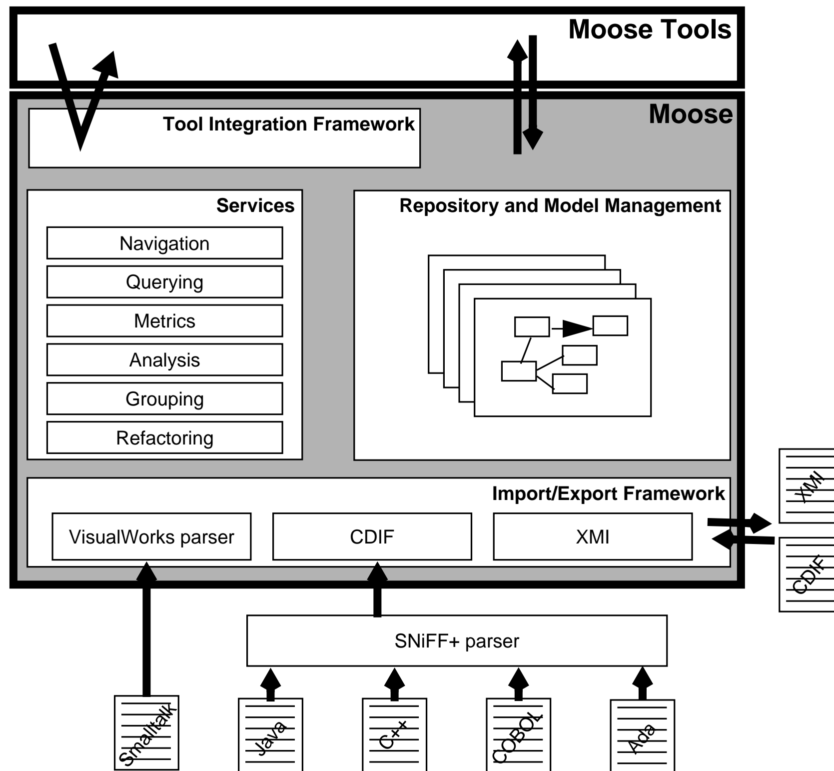

Figure 2 Architecture of Moose

into a few dozen applications and thus obtain a better view of the whole system.

- *Refactoring.* The Moose Refactoring Engine implements language-independent refactorings as defined in [TDDN00]. Section 7 describes the engine in more detail.

Tools Layer and Tools Integration Framework. The functionality which is provided by Moose is to be used by tools. This is represented by the top layer of figure 2. Tools can use the repository and services of Moose and use the Tools Integration Framework to find each other and integrate. The Tools Integration Framework and examples of tools based on Moose are described in section 8.2.

The following sections discuss the different parts of Moose in more detail.

4. Querying and Navigation

One of the challenges when dealing with large complex metamodels 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.

4.1 Programming Queries

The fact that the metamodel 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 (WNOC is a metric and means Weighted Number Of Children).

```
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 [KC99].

Moose Finder

The Moose Finder is a tool that allows one 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 supports multiple models in the context of software evolution.

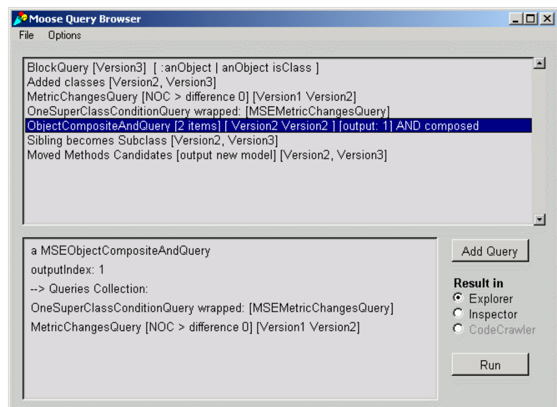

Figure 3 Moose Finder

4.2 Querying and navigating 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 metamodel, 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 a uniform way to represent model information (see figure 4). 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 classes of the current model. The bottom left pane represents all the possible ways to access other entities from the currently selected ones. 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.

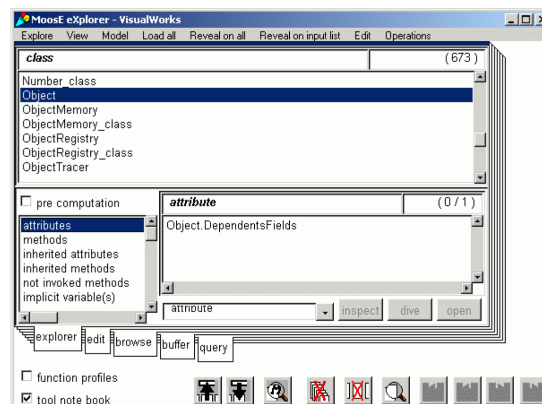

Figure 4 Moose Explorer

5. Metrics and other analysis support

Moose includes a metrics engine for the computation and storage of metric measurements. We support so called Design Metrics [LK94], i.e., metrics which are *extracted from the source entities themselves*. These metrics are used to assess the size and in some cases the quality and complexity of software. The current implementation of the metrics engine includes language-independent as well as language-specificmetrics. The language independent metrics can be extracted from our metamodel. Examples are the number of methods or the number of attributes of class. Examples for language- specific metrics are the number of method protocols of a class in Smalltalk or the number of private attributes of a class in C++ or Java.

6. Grouping

Moose supports grouping of any entities. Groups can be nested. They can also described by intention and by extention. The following code shows how all classes of the system under analysis can be grouped into group representing the categories they belong to.

```
MSEModel currentModel allClassesDo:
  [:aMooseClass |
  | group category|
  aMooseClass isStub
    ifFalse: [
      category :=
        MSEUtilities extractCategory:
                        aMooseClass sourceAnchor.
        group := MSEModel currentModel
                enumeratedGroupWithName:category.
        group add: aMooseClass ]]
```

For all classes, we check if the class is not a stub i.e., a class that is outside of the analysed code, then the method groupWithName: returns an existing group representing a category or a new one to which the class is added.

Note that in this code we are not manipulating Smalltalk classes directly but Moose representation of these classes which are language independent. However, this example illustrates that while the Moose meta-model is language independent the interpretation of information has sometimes to be language dependent. Here the category concept only exists in Smalltalk.

7. Moose Refactoring Engine

Refactoring [FBB+99] is about making changes to code to improve its structure, simplicity, flexibility, understandability or performance [Bec99] without changing the external behaviour of the system. The Moose Refactoring Engine provides support for twelve low-level refactorings. The functionality is similar to the Refactoring Browser [RBJ97] for Smalltalk, but for multiple implementation languages, currently Smalltalk and Java.

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 [DTD01]. 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 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 [RBJ97] 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.

8. Information Exchange and Tool Integration

Interoperability between reengineering tools is supported in two ways. First, there is the possibility to exchange information in text files using industry standard exchange formats. Second, tools written in VisualWorks Smalltalk can interoperate with the Moose repository, its services and each other at runtime.

8.1 Information Exchange with CDIF and XMI

To exchange FAMIX-based information between different tools, Moose provides two textual formats. One is CDIF [Com94], an industrial standard for transferring models created with different tools. The main reasons for adopting CDIF are, that it is an industry standard and 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.

Recently, we have adopted XMI (XML Metadata Interchange [OMG98]) as a second storage and exchange format [Sch01] [Fre00]. XMI is an OMG standard for exchanging models based on the MOF (Meta-Object Facility [OMG00]) and uses XML (Extensible Markup Language [BPSM98]) as the underlying technology to save this information.

The main reason to support a second standard is that CDIF did not succeed in becoming a widely used standard. XMI seems to stand a better chance, especially because it is based on XML, which is likely to become the de facto standard for transferring information between applications and allows the use of XML-based technologies such as XSL. Secondly, XMI is based on the MOF, which is likely to become the de facto standard to describe metamodels and offers excellent integration to MOF-based metamodels such as UML.

As shown in figure 2 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+ [Tak96] [TD99]. Next to parsers we also have integrations with external environments such as the Nokia Reengineering Environment [DD99].

A third format we plan to support is the Graph eXchange Language (GXL) [HWS00]. GXL is a collaborative effort from several academic and industrial research institutes to come up with an exchange format and a set of metamodels for information exchange for reengineering tools.

8.2 Tool Integration Framework and Tools

As mentioned earlier, 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. To enable tools to interact, Moose provides a simple tool registration and lookup mechanism.

At this point in time the following tools have been developed:

- CodeCrawler supports reverse engineering through the combination of metrics and visualization [Lan99] [DDL99] (see figure 5). Through simple visualisations 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.

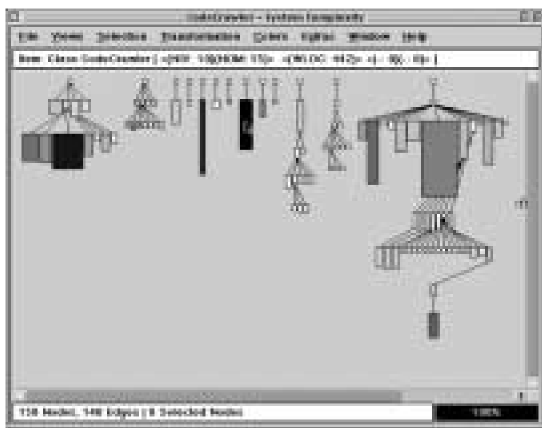

Figure 5 CodeCrawler

- Gaudi [RD99] combines dynamic with static information (see figure 6). 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.

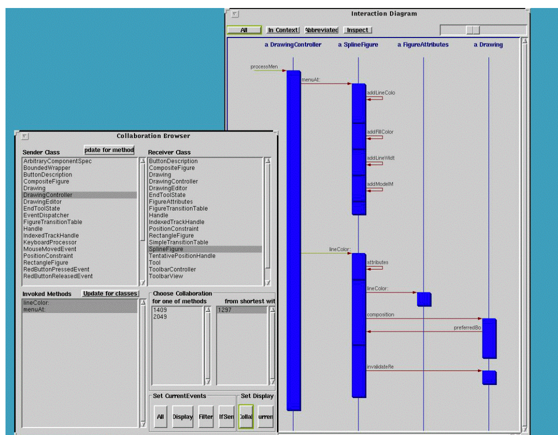

Figure 6 Gaudi

- Supremo [KN01] uses the Moose repository and the duplication detection tool Duploc [DRD99] to put duplication in context. Figure 7 shows an example: the colored nodes in the class inheritance tree represent the distribution of a recurring code sequence.

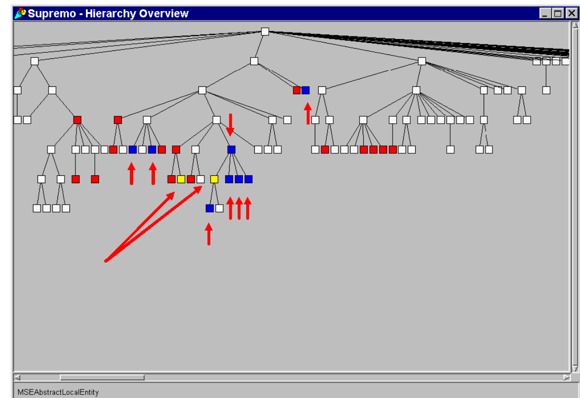

Figure 7 Supremo

Except for providing the foundation for our own tools, Moose also interfaces with external tools. Examples are the Nokia Reengineering Environment [DD99] and Soul [Wuy01].

9. Validation and Evaluation

Moose and its tools have been validated in several academic and industrial experiences, some of which we list in more detail below. 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. A 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 with 128 MB of RAM.

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 [DDL99]. 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 700'000. The workability of Moose then depends largely on the amount of RAM of the computer it is running on. In the industrial context we reached 300'000 entities, which due to the small amount of RAM (128 MB) of the computer made the VisualWorks Smalltalk 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 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 heavily 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 the main requirements for a reengineering environment.We now discuss how Moose evaluates in that context.

1. **Support for reengineering tasks.** Moose supports all major reengineering tasks, mainly because it has grown while being constantly validated in industrial contexts. Therefore, although coming from an academic background, the whole environment has strong roots in industry.
2. **Extensible.** The extensibility of Moose is inherent to the extensibility of its metamodel. 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.
3. **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.
4. **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 700,000 entities and could still be held completely in memory without any notable performance penalties.
5. **Information Exchange and Tool Integration.** The integration with several external tools has been repeatedly done without major problems. The information can be exchanged with other tool platforms using either CDIF, XMI or GXL. Especially GXL may open up a whole new set of possible cooperations with other tools.

10. Conclusion and Future Work

In this article 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 metamodel 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 repeatedly 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, system evolution, steering of refactorings based on code duplication detection or other kinds of analysis.

11. Some Links

In this section we want to provide you with a set of links regarding Moose, related tools, as well as research in this context.

- SCG P.U.R.E. (**S**oftware **C**omposition **G**roup **P**rogram **U**nderstanding and **R**e**E**ngineering) (http://www.iam.unibe.ch/~pure/) are the members of the Software Composition Group (http://www.iam.unibe.ch/~scg/) doing research in the context of program understanding and reengineering. All the tools mentioned in this article can be downloaded for free.
- Conferences:

  WCRE (Working Conference on Reverse Engineering) http://www.reengineer.org/

  ICSM (International Conference on Software Maintenance) http://tcse.org/

References

[Bec99] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

BPSM98] Tim Bray, Jean Paoli, and C.M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0 - w3c recommendation 10-february-1998. Technical Report REC-xml-19980210, World Wide Web Consortium, February 1998.

[Com94] CDIF Technical Committee. Cdif framework for modeling and extensibility. Technical Report EIA/IS-107, Electronic Industries Association, January 1994. See http://www.cdif.org/.

[DD99] Stéphane Ducasse and Serge Demeyer, editors. *The FAMOOS Object-Oriented Reengineering Handbook*. University of Berne, October 1999. See http://www.iam.unibe.ch/~famoos/handbook.

[DDL99] Serge Demeyer, Stéphane Ducasse, and Michele Lanza. A hybrid reverse engineering platform combining metrics and program visualization. In Francoise Balmas, Mike Blaha, and Spencer Rugaber, editors, *Proceedings WCRE'99*. IEEE, October 1999.

[DDT99] Serge Demeyer, Stéphane Ducasse, and Sander Tichelaar. Why unified is not universal. UML shortcomings for coping with round-trip engineering. In Bernhard Rumpe, editor, *Proceedings UML'99*, LNCS 1723, October 1999. Springer-Verlag.

[DRD99] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In Hongji Yang and Lee White, editors, *Proceedings ICSM'99*, pages 109–118. IEEE, September 1999.

[DTD01] Serge Demeyer, Sander Tichelaar, and Stéphane Ducasse. FAMIX 2.1 - the FAMOOS information exchange model. Technical report, University of Berne, 2001. to appear.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[Fre00] Michael Freidig. XMI for FAMIX. Informatikprojekt, University of Berne, June 2000.

[HEH+96] 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.

[HWS00] Richard C. Holt, Andreas Winter, and Andy Schürr. GXL: Towards a standard exchange format. In *Proceedings WCRE'00*, November 2000.

[Kaz96] R. Kazman. Tool support for architecture analysis and design, 1996. Proceedings of Workshop (ISAW-2) joint Sigsoft.

[KC99] R. Kazman and S.J. Carriere. Playing detective: Reconstructing software architecture from available evidence. *Automated Software Engineering*, April 1999.

[KN01] Georges Golomingi Koni-N'sapu. A scenario based approach for refactoring duplicated code in object oriented systems. Diploma thesis, University of Berne, June 2001.

[Lan99] Michele Lanza. Combining metrics and graphs for object oriented reverse engineering. Diploma thesis, University of Bern, October 1999.

[LK94] Mark Lorenz and Jeff Kidd. *Object-Oriented Software Metrics: A Practical Guide*. Prentice-Hall, 1994.

[MN97] Gail Murphy and David Notkin. Reengineering with reflexion models: A case study. *IEEE Computer*, 8:29–36, 1997.

[OMG98] Object Management Group. XML Metadata Interchange (XMI). Technical Report ad/98-10-05, Object Management Group, February 1998.

[OMG00] Object Management Group. Meta Object Facility (MOF) specification (version 1.3). Technical report, Object Management Group, March 2000.

[RBJ97] Don Roberts, John Brant, and Ralph E. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems (TAPOS)*, 3(4):253–263, 1997.

[RD99] Tamar Richner and Stéphane Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In Hongji Yang and Lee White, editors, *Proceedings ICSM'99*, pages 13–22. IEEE, September 1999.

[Sch01] Andreas Schlapbach. Generix XMI support for the MOOSE reengineering environment. Informatikprojekt, University of Bern, Jun 2001.

[Tak96] TakeFive Software GmbH. *SNiFF+*, 1996.

[TD99] Sander Tichelaar and Serge Demeyer. SNiFF+ talks to Rational Rose – interoperability using a common exchange model. In *SNiFF+ User's Conference*, January 1999.

[TDDN00] Sander Tichelaar, Stéphane Ducasse, Serge Demeyer, and Oscar Nierstrasz. A meta-model for language-independent refactoring. In *Proceedings ISPSE 2000*. IEEE, 2000.

[Wuy01] Roel Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, 2001.