Agile Software Assessment with Moose[*]

[Extended Abstract]

Oscar Nierstrasz
Software Composition Group
University of Bern, Switzerland
http://scg.unibe.ch

ABSTRACT

During software maintenance, much time is spent reading and assessing existing code. Unfortunately most of the tools available for exploring and assessing code, such as browsers, debuggers and profilers, focus on development tasks, and offer little to support program understanding. We present a platform for software and data analysis, called Moose, which enables the rapid development of custom tools for software assessment. We demonstrate how Moose supports agile software assessment through a series of demos, we illustrate some of the custom tools that have been developed, and we draw various lessons learned for future work in this domain.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments

1. INTRODUCTION

It is well-acknowledged that developers often spend as much time understanding code as they do writing new code [17]. There are many reasons for this phenomenon. As established by Lehmann and Belady [18], real-world software systems become more complex over time as they are adapted to fulfill new requirements, unless effort is invested to simplify their design.

The needed reengineering effort is made more difficult because critical information is missing from the software artifacts. As an example, we can note that *software architecture is not in the code.* Although software architecture is perhaps the most important technological driver of a complex system, its presence is only implicit in the source code. Worse, written documentation is rarely accurate and often fails to capture many of the important aspects of architecture [6]. Ideally, like in literate programming, the source code would be self-explanatory, but that is rarely the case. For this reason, much effort is often spent in recovering architecture. Many tools have been developed over the years, such as *architectural description languages* [28], tools to detect violations of architectural constraints [3], and clustering tools to recover aspects of architecture from source code [20, 21]. Nevertheless, such tools have not yet entered the mainstream.

Few dedicated tools are available to the developer to support program understanding and analysis. Most tools in the IDE (integrated development environment), such as classical browsers, debuggers and profilers, focus on low-level programming tasks rather than on broader architectural or software engineering issues. Some specialized tools exist, but they are expensive to develop, maintain and integrate into the IDE. (As an example, consider Senseo [27], an Eclipse plugin that integrates run-time information into the classical source code views.)

Program understanding entails *software assessment*, the process of posing specific questions about the software system under study and carrying out specialized analyses to answer these questions. This requires the development of specialized tools. Two examples of such tools are *feature views* [11], which establish a correspondence between software artifacts and the features they support, and *object-flow analysis* [19], which tracks the flow of objects at run-time to identify the source of obscure bugs.

We argue that there is a demonstrated need for *agile software assessment*, that is, a meta-tooling infrastructure and environment that allows developers to rapidly and cheaply develop lightweight, custom tools to support program understanding. Such an environment must go beyond the conventional IDE to integrate various sources of information (*i.e.*, not just source code), construct high-level models of this information, offer querying and navigation facilities, support metrics extraction, and offer means to generate high-level, interactive views to support exploration and *ad hoc* analyses. As step in this direction, we present *Moose*[1], a software and data analysis platform (section 2), and we illustrate some of the ways in which it supports agile software assessment. We draw some key lessons learned (section 3) and conclude with remarks on open problems and future work (section 4).

[*]PREPRINT. Invited keynote presentation at IWRE 2012(3rd India Workshop on Reverse Engineering), SIGSOFT Softw. Eng. Notes, Vol 37, No. 3, pp. 1–5, May 2012. DOI: 10.1145/180921.2180925

[1]http://moosetechnology.org

2. MOOSE: AN ANALYSIS PLATFORM

Moose is a platform for analyzing software and data [23]. The system was originally developed between 1996 and 1999 as part of the FAMOOS[2] project. At the core of Moose is the FAMIX meta-model (Figure 1), which offers a simple, language-independent way to represent object-oriented models.

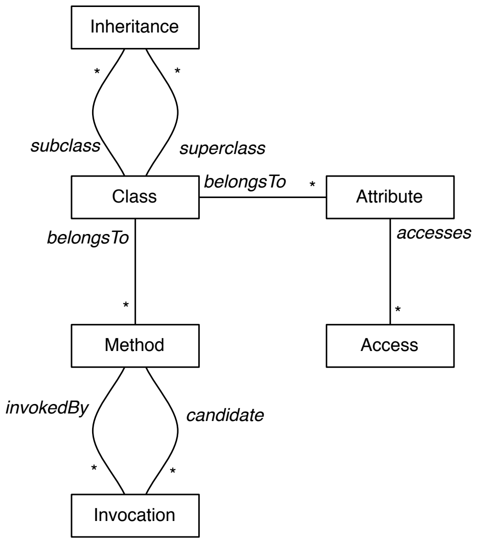


Figure 1: FAMIX core metamodel (extract)

A typical use-case for Moose is to use one of several available importers to parse source code in one of many languages, such as Java or C++, to generate a so-called MSE (Moose exchange) file. Moose then imports this file to build a model that can be queried and manipulated. Moose is implemented in Pharo[3] [4], an open-source Smalltalk implementation. As a consequence, Moose models reside in a Smalltalk image.

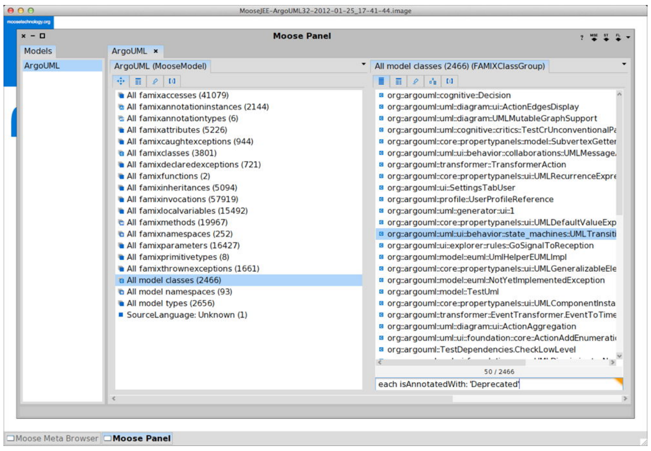
Moose offers a lightweight user interface (Figure 2) for browsing and querying models. A model consists of a group of entities. Properties and metrics over entities can be used to pose *ad hoc* queries and filter groups of interest.

Moose is designed as an open system. The FAMIX meta-model is extensible, thus allowing custom tools to be built that extend the meta-model with new concepts, for example, modeling history [9], semantic clustering [13], and feature views [11].

One of the key features needed for custom tools is a means to generate lightweight visualizations. *Polymetric views* [16] map software metrics to very simple graphical representations. A *system complexity view* (Figure 3), for example, maps the number of methods, number of attributes, and lines of code, respectively, to the height, width and color of the classes in a class hierarchy view, thus offering an easily comprehensible overview of the "big" and "small" classes in a system.

Polymetric views are used for producing many different kinds of visualization, such as clone evolution [2] (showing the evolution of clones over time), class blueprints [8] (showing the relationships between methods and attributes of a class), topic correlation matrices [13] (showing the clusters of semantic topics within a system), distribution map [7] (showing how features relate to artifacts), hierarchy evolution [9] (showing how a system has evolved over time), and ownership maps [10] (showing who has worked on which parts of a system over time).

[2]`http://scg.unibe.ch/archive/famoos/`
[3]`http://pharo-project.org`


MooseJEE-ArgoUML32-2012-01-25_17-41-44.image
Moose Panel
Models
ArgoUML
ArgoUML ×
ArgoUML (MooseModel)
All model classes (2466) (FAMIXClassGroup)
Moose Meta Browser
Moose Panel

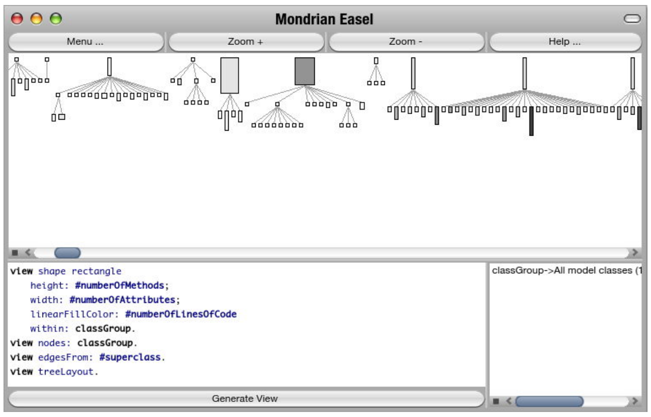
```
All famixaccesses (41079)
All famixannotationinstances (2144)
All famixannotationtypes (6)
All famixattributes (5226)
All famixcaughtexceptions (944)
All famixclasses (3801)
All famixdeclaredexceptions (721)
All famixfunctions (2)
All famixinheritances (5094)
All famixinvocations (57919)
All famixlocalvariables (15492)
All famixmethods (19967)
All famixnamespaces (252)
All famixparameters (16427)
All famixprimitivetypes (8)
All famixthrownexceptions (1661)
All model classes (2466)
All model namespaces (93)
All model types (2656)
SourceLanguage: Unknown (1)
```

```
org::argouml::cognitive::Decision
org::argouml::uml::diagram::ui::ActionEdgesDisplay
org::argouml::uml::diagram::UMLMutableGraphSupport
org::argouml::uml::cognitive::critics::TestCrUnconventionalPa
org::argouml::core::propertypanels::model::SubvertexGetter
org::argouml::uml::ui::behavior::collaborations::UMLMessage.
org::argouml::transformer::TransformerAction
org::argouml::core::propertypanels::ui::UMLRecurrenceExpre
org::argouml::ui::SettingsTabUser
org::argouml::profile::UserProfileReference
org::argouml::uml::generator::ui::1
org::argouml::core::propertypanels::ui::UMLDefaultValueExp
org::argouml::uml::ui::behavior::state_machines::UMLTransiti
org::argouml::ui::explorer::rules::GoSignalToReception
org::argouml::model::euml::UmlHelperEUMLImpl
org::argouml::core::propertypanels::ui::UMLGeneralizableEle
org::argouml::model::euml::NotYetImplementedException
org::argouml::model::TestUml
org::argouml::core::propertypanels::ui::UMLComponentInsta
org::argouml::transformer::EventTransformer.EventToTime
org::argouml::uml::diagram::ui::ActionAggregation
org::argouml::uml::ui::foundation::core::ActionAddEnumerati
org::argouml::TestDependencies.CheckLowLevel
50 / 2466
each isAnnotatedWith: 'Deprecated'
```

Figure 2: Moose User Interface

Polymetric views were introduced in an early version of Moose as a classical object-oriented white box framework, allowing developers to use subclassing to define new visualizations. As is common with such frameworks, considerable experience is required to effectively use the framework.

Mondrian [22] (Figure 3) is a black box component framework that makes it much easier to build such visualizations. Since the host language, Smalltalk, has a very simple syntax, composing new visualization resembles writing scripts in a DSL. A lightweight, interactive UI allows developers to immediately see the result of changes to their Mondrian scripts. Mondrian supports agile software assessment by enabling developers to generate new visualizations in minutes instead of days.



```
view shape rectangle
   height: #numberOfMethods;
   width: #numberOfAttributes;
   linearFillColor: #numberOfLinesOfCode
   within: classGroup.
view nodes: classGroup.
view edgesFrom: #superclass.
view treeLayout.
```

Figure 3: Mondrian visualization engine displaying a scripted System Complexity view

This same principle has been used successfully to develop two related meta-tooling environments. *Glamour* [5] applies this idea to the construction of custom browsers for arbitrary models (not just source code). A browser is built up from a number of components, each of which renders some fragment of a model, and *transmits* information about that fragment to another connected component. Data exploration is a fundamental component of data analysis. Glamour aims at modeling the basic navigation and interaction methods inherent in any information exploration tool. A classical Smalltalk browser with panels displaying packages, classes, methods and source code can be constructed as a compact Glamour script. Such a script can then easily be adapted to incorporate useful visualizations. Dedicated browsers for other kinds of models can similarly be scripted with low effort.

EyeSee [12] also takes the same idea and applies it to the construction of graphical charts. Since the underlying infrastructure is the same, EyeSee charts can also be integrated into Glamour browsers.

3. LESSONS LEARNED

Here follow some of the lessons we have learned based on our experience using and developing Moose over roughly a 15 year period (NB: the first three of these are discussed in more detail in a previous paper [24]):

Less is more. In other words, lightweight techniques go a long way to solving most problems. Simple visualizations, simple techniques, simple and general meta-models, languages with simple syntax, are important enablers for agile software assessment.

You can change a running system. Unlike mainstream programming languages and development environments, Smalltalk does not artificially split compile time and run time. The *only* way to change a Smalltalk system is at run time. This is an important enabler as it allows one to move very quickly when developing new tools and ideas.

Reify everything. By making meta-level concepts (such as the implementation of the underlying system) explicit as first-class model entities, they become queryable and manipulable at run time.

A black box framework is a DSL. A clean, component-oriented design yields an “algebra of objects” that can be composed and configured at run time. If the host language has a simple and readable syntax (as Smalltalk does), the interface of the framework yields a so-called *internal DSL* (domain-specific language), and configurations of components can be read like scripts in this DSL.

Know when to invest in infrastructure. Moose has been re-designed and reimplemented several times over its 15-year history. Although there have been virtually no scientific papers published on Moose itself, and although maintaining, refining, and re-architecting the infrastructure has been expensive, the investment has paid off handsomely as an enabler for research for (literally) hundreds of research papers, and numerous PhD and Masters theses.

4. FUTURE WORK

Although we believe that Moose makes significant steps towards supporting agile software assessment by offering a meta-tooling environment for rapidly developing tools for custom analyses, there remain many open questions, both technical and methodological.

First, we have only limited insight into precisely what exactly developers do. There has recently been increasing interest in performing empirical studies to determine what obstacles developers encounter, and how better to support them. A successful environment to support agile software assessment should be well-grounded in such studies.

Second, there is a tragic disconnect between programming languages and software engineering issues. We have already mentioned the lack of support for encoding architecture and architectural constraints in programming languages. Program source code is similarly disconnected from high-level domain models, requirements models, and user-oriented feature views, to mention just a few examples. Program source code is similarly disconnected from other kinds of software artifacts. An IDE supporting agile software assessment should track all such dependencies and connections. IDEs furthermore focus too much on static views of source code. Senseo [27], mentioned earlier, offers just one example of an attempt to link static and dynamic views, but more needs to be done. An effective IDE should support *continuous monitoring* of source code evolution as well as the running system to support analysis of architecture, design, code smells, and so on.

Third, though by no means finally, a key bottleneck for agile software assessment is the automated construction of software models. Although Moose can readily import models from Java source code, importing models from other languages, such as PHP, Cobol or PL/1, is problematic, since a custom parser must be developed to generate FAMIX models. This can entails days or weeks of effort. We have explored several innovative techniques to speed up this process, such as generating parsers automatically from examples [25], recovering models from the abstract or concrete syntax trees of existing parsers [15, 14], and even using genetic programming to generate grammars [29]. Although modest success was achieved in each case, we are far from having practical techniques to rapidly import models from unknown languages. A related issue is that modern software systems are rarely implemented using a single source language. Java enterprise applications, for example, include not only Java source code but also HTML, JavaScript, XML, SQL and possibly other kinds of artifacts. To analyze such systems, one must construct software models that combine all these sources of information [26, 1].

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project “Synchronizing Models and Code” (SNF Project No. 200020-131827, Oct. 2010 - Sept. 2012). We also thank Tudor Gîrba, Fabrizio Perin and Mircea Lungu for their contributions in preparing the keynote presentation and in reviewing this extended abstract.

5. REFERENCES

[1] Amir Aryani, Fabrizio Perin, Mircea Lungu, Abdun Naser Mahmood, and Oscar Nierstrasz. Can we predict dependencies using domain information? In *Proceedings of the 18th Working Conference on Reverse Engineering (WCRE 2011)*, October 2011. Available from: `http://scg.unibe.ch/archive/papers/Aria11aWCRE11.pdf`, `doi:10.1109/WCRE.2011.17`.

[2] Mihai Balint, Tudor Gîrba, and Radu Marinescu. How developers copy. In *Proceedings of International Conference on Program Comprehension (ICPC 2006)*, pages 56–65, 2006. Available from: `http://scg.unibe.ch/archive/papers/Bali06aHowDevelopersCopy.pdf`, `doi:10.1109/ICPC.2006.25`.

[3] Walter Bischofberger, Jan Kühl, and Silvio Löffler. Sotograph – a pragmatic approach to source code architecture conformance checking. In *Software Architecture*, volume 3047 of *LNCS*, pages 1–9. Springer-Verlag, 2004. `doi:10.1007/b97879`.

[4] Andrew Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example*. Square Bracket Associates, 2009. Available from: `http://pharobyexample.org`.

[5] Philipp Bunge. Scripting browsers with Glamour. Master's thesis, University of Bern, April 2009. Available from: `http://scg.unibe.ch/archive/masters/Bung09a.pdf`.

[6] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Square Bracket Associates, 2008. Available from: `http://scg.unibe.ch/download/oorp/`.

[7] Stéphane Ducasse, Tudor Gîrba, and Adrian Kuhn. Distribution map. In *Proceedings of 22nd IEEE International Conference on Software Maintenance (ICSM '06)*, pages 203–212, Los Alamitos CA, 2006. IEEE Computer Society. Available from: `http://scg.unibe.ch/archive/papers/Duca06cDistributionMap.pdf`, `doi:10.1109/ICSM.2006.22`.

[8] Stéphane Ducasse and Michele Lanza. The Class Blueprint: Visually supporting the understanding of classes. *Transactions on Software Engineering (TSE)*, 31(1):75–90, January 2005. Available from: `http://scg.unibe.ch/archive/papers/Duca05bTSEClassBlueprint.pdf`, `doi:10.1109/TSE.2005.14`.

[9] Tudor Gîrba and Stéphane Ducasse. Modeling history to analyze software evolution. *Journal of Software Maintenance: Research and Practice (JSME)*, 18:207–236, 2006. Available from: `http://scg.unibe.ch/archive/papers/Girb06aHismo.pdf`.

[10] Tudor Gîrba, Adrian Kuhn, Mauricio Seeberger, and Stéphane Ducasse. How developers drive software evolution. In *Proceedings of International Workshop on Principles of Software Evolution (IWPSE 2005)*, pages 113–122. IEEE Computer Society Press, 2005. Available from: `http://scg.unibe.ch/archive/papers/Girb05cOwnershipMap.pdf`, `doi:10.1109/IWPSE.2005.21`.

[11] Orla Greevy, Stéphane Ducasse, and Tudor Gîrba. Analyzing software evolution through feature views. *Journal of Software Maintenance and Evolution: Research and Practice (JSME)*, 18(6):425–456, 2006. Available from: `http://scg.unibe.ch/archive/papers/Gree06bTraceScraperJSME-SCG.pdf`, `doi:10.1002/smr.340`.

[12] Matthias Junker and Markus Hofstetter. Scripting diagrams with eyesee. Bachelor's thesis, University of Bern, May 2007. Available from: `http://scg.unibe.ch/archive/projects/Junk07aJunkerHofstetterEyeSee.pdf`.

[13] Adrian Kuhn, Stéphane Ducasse, and Tudor Gîrba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49(3):230–243, March 2007. Available from: `http://scg.unibe.ch/archive/drafts/Kuhn06bSemanticClustering.pdf`, `doi:10.1016/j.infsof.2006.10.017`.

[14] Daniel Langone. Recycling trees: Mapping Eclipse ASTs to Moose models. Bachelor's thesis, University of Bern, January 2009. Available from: `http://scg.unibe.ch/archive/projects/Lang09a.pdf`.

[15] Daniel Langone and Toon Verwaest. Extracting models from IDEs. In *2nd Workshop on FAMIX and Moose in Software Reengineering (FAMOOSr 2008)*, pages 32–35, October 2008. Available from: `http://scg.unibe.ch/archive/papers/Lang08aModelExtraction.pdf`.

[16] Michele Lanza and Stéphane Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *Transactions on Software Engineering (TSE)*, 29(9):782–795, September 2003. Available from: `http://scg.unibe.ch/archive/papers/Lanz03dTSEPolymetric.pdf`, `doi:10.1109/TSE.2003.1232284`.

[17] Thomas D. LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: a study of developer work habits. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 492–501, New York, NY, USA, 2006. ACM. `doi:10.1145/1134285.1134355`.

[18] Manny Lehman and Les Belady. *Program Evolution: Processes of Software Change*. London Academic Press, London, 1985. Available from: `ftp://ftp.umh.ac.be/pub/ftp_infofs/1985/ProgramEvolution.pdf`.

[19] Adrian Lienhard, Tudor Gîrba, and Oscar Nierstrasz. Practical object-oriented back-in-time debugging. In *Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP'08)*, volume 5142 of *LNCS*, pages 592–615. Springer, 2008. ECOOP distinguished paper award. Available from: `http://scg.unibe.ch/archive/papers/Lien08bBackInTimeDebugging.pdf`, `doi:10.1007/978-3-540-70592-5_25`.

[20] Mircea Lungu and Michele Lanza. Softwarenaut: Exploring hierarchical system decompositions. In *Proceedings of CSMR 2006 (10th European Conference on Software Maintenance and Reengineering)*, pages 351–354, Los Alamitos CA, 2006. IEEE Computer Society Press. `doi:10.1109/CSMR.2006.52`.

[21] Mircea Lungu and Oscar Nierstrasz. Recovering software architecture with softwarenaut. *ERCIM News*, 88, January 2012. Available from: `http://ercim-news.ercim.eu/en88/special/recovering-software-architecture-with-softwarenaut`.

[22] Michael Meyer, Tudor Gîrba, and Mircea Lungu. Mondrian: An agile visualization framework. In *ACM Symposium on Software Visualization (SoftVis'06)*, pages 135–144, New York, NY, USA, 2006. ACM Press. Available from: `http://scg.unibe.ch/archive/papers/Meye06aMondrian.pdf`, `doi:10.1145/1148493.1148513`.

[23] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of Moose: an agile reengineering environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE'05)*, pages 1–10, New York, NY, USA, 2005. ACM Press. Invited paper. Available from: `http://scg.unibe.ch/archive/papers/Nier05cStoryOfMoose.pdf`, `doi:10.1145/1095430.1081707`.

[24] Oscar Nierstrasz and Tudor Gîrba. Lessons in software evolution learned by listening to Smalltalk. In J. van Leeuwen et al., editor, *SOFSEM 2010*, volume 5901 of *LNCS*, pages 77–95. Springer-Verlag, 2010. Available from: `http://scg.unibe.ch/archive/papers/Nier10aSmalltalkLessons.pdf`, `doi:10.1007/978-3-642-11266-9_7`.

[25] Oscar Nierstrasz, Markus Kobel, Tudor Gîrba, Michele Lanza, and Horst Bunke. Example-driven reconstruction of software models. In *Proceedings of Conference on Software Maintenance and Reengineering (CSMR 2007)*, pages 275–286, Los Alamitos CA, 2007. IEEE Computer Society Press. Available from: `http://scg.unibe.ch/archive/papers/Nier07aExampleDrivenMR.pdf`, `doi:10.1109/CSMR.2007.23`.

[26] Fabrizio Perin, Tudor Gîrba, and Oscar Nierstrasz. Recovery and analysis of transaction scope from scattered information in Java enterprise applications. In *Proceedings of International Conference on Software Maintenance 2010*, September 2010. Available from: `http://scg.unibe.ch/archive/papers/Peri10aTransactionRecovery.pdf`, `doi:10.1109/ICSM.2010.5609572`.

[27] David Röthlisberger, Marcel Härry, Alex Villazón, Danilo Ansaloni, Walter Binder, Oscar Nierstrasz, and Philippe Moret. Exploiting dynamic information in ides improves speed and correctness of software maintenance tasks. *Transactions on Software Engineering*, 2011. To appear (preprint online). Available from: `http://scg.unibe.ch/archive/papers/Roet11aSenseoTSE.pdf`, `doi:10.1109/TSE.2011.42`.

[28] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[29] Sandro De Zanet. Grammar generation with genetic programming — evolutionary grammar generation. Master's thesis, University of Bern, July 2009. Available from: `http://scg.unibe.ch/archive/masters/DeZa09a.pdf`.