



Ingeniare. Revista Chilena de Ingeniería

ISSN: 0718-3291

facing@uta.cl

Universidad de Tarapacá

Chile

Scalise P., Eugenio G.; Favre, Jean-Marie; Zambrano, Nancy
MODEL-DRIVEN REVERSE ENGINEERING AND PROGRAM COMPREHENSION: AN EXAMPLE
Ingeniare. Revista Chilena de Ingeniería, vol. 18, núm. 1, 2010, pp. 76-83
Universidad de Tarapacá
Arica, Chile

Available in: <http://www.redalyc.org/articulo.oa?id=77218811009>

- How to cite
- Complete issue
- More information about this article
- Journal's homepage in redalyc.org

redalyc.org

Scientific Information System
Network of Scientific Journals from Latin America, the Caribbean, Spain and Portugal
Non-profit academic project, developed under the open access initiative

MODEL-DRIVEN REVERSE ENGINEERING AND PROGRAM COMPREHENSION: AN EXAMPLE

INGENIERÍA REVERSA Y COMPRENSIÓN DE PROGRAMAS DIRIGIDA POR MODELOS: UN EJEMPLO

Eugenio G. Scalise P.¹ Jean-Marie Favre² Nancy Zambrano¹

Recibido 3 de abril de 2009, aceptado 16 de abril de 2010

Received: April 3, 2009 Accepted: April 16, 2010

RESUMEN

En este artículo se presenta un ejemplo que describe cómo la Ingeniería Dirigida por Modelos (IDM) puede ser aplicada al desarrollo de herramientas para la ingeniería en reverso y comprensión de programas. Se seleccionó como caso de estudio la herramienta CodeCrawler, específicamente la funcionalidad denominada polymetric view, desarrollada mediante tecnologías y técnicas de la IDM. Para ello, se emplearon dos metamodelos (entrada y salida) y transformaciones a nivel de los metamodelos que permiten generar la información de un polymetric view asociado a un proyecto de software, utilizando la información extraída del código fuente. Las tecnologías utilizadas para el desarrollo del ejemplo, las relacionadas con el Eclipse Modeling Project, específicamente el lenguaje de transformación ATL y el lenguaje KM3. El enfoque seguido para obtener la implementación resultante puede ser utilizado para desarrollar una herramienta de ingeniería en reverso y comprensión de software mediante técnicas de la IDM, incorporando el vocabulario del dominio en la implementación.

Palabras clave: Ingeniería dirigida por modelos, ingeniería en reverso, comprensión de programas, modelos, metamodelos, transformaciones entre metamodelos, CodeCrawler, ATL, KM3.

ABSTRACT

This paper presents an example of how Model-Driven Engineering (MDE) can be applied to the development of tools for reverse engineering and program comprehension. The tool CodeCrawler was selected as an example; in particular, the polymetric view feature was computed using MDE techniques. To this end, two metamodels were proposed (source and target) and meta-level transformations that were used to deduce the information of a polymetric view associated to any software project (source code). The technologies selected to develop the example were those related with the Eclipse Modeling Project, specifically the ATL and the KM3 languages. The approach used in this paper to obtain the implementation can be used to develop all the features of a MDE-oriented software comprehension tool, obtaining a domain-oriented implementation.

Keywords: Model-driven engineering, MDE, reverse engineering, program comprehension, models, metamodels, transformations between metamodels, CodeCrawler, ATL, KM3.

INTRODUCTION

Software evolution is one of the most significant and recurrent problems in software engineering. This is a consequence of the fast changes over the last decades, at technological level and in software development methods and techniques; subsequently, techniques that support software evolution are a necessity, especially for big software (million lines of code).

Today, as technological changes are so fast, some technologies become obsolete quickly, even before they reach an appropriate maturity level. The Object-Oriented (OO) paradigm was considered to be a solution for several software evolution and maintenance problems, but the empirical evidence is proving that OO is creating new evolution problems and must be used with care to ensure that the complexity of the maintenance is not greater than the complexity of traditional systems.

¹ Universidad Central de Venezuela. Facultad de Ciencias. Escuela de Computación. Centro ISYS. Caracas, Venezuela.

E-mail: eugenio.scalise@ciens.ucv.ve; nancy.zambrano@ciens.ucv.ve

² Universidad Joseph Fourier (Grenoble 1). Equipo Adele. Laboratorio de Informática de Grenoble (LIG). Grenoble, Francia.

E-mail: Jean-Marie.Favre@imag.fr

In addition, the legacy software concept has changed over time. In the production systems of the industrial world, where the systems in production are, the legacy term has become common. Over recent years, it has become normal to use the “legacy” term to qualify Java or C++ software, and hence it can be said that the evolution of OO software and traditional software (non structured code in COBOL or similar) have the same level of importance. In the past, legacy software were monolithic systems, whereas nowadays almost all software is a combination of distributed components, using standard middleware technologies and enterprise frameworks, which causes problems in the evolution and maintenance process.

In the 80s techniques were proposed for the comprehension and evolution of big software products like reengineering, reverse engineering and restructuring. Chikofsky and Cross [5] declared that reverse engineering is the process of analyzing a subject system to identify the system’s components and their interrelationships and create representations of the system in another form or at a higher level of abstraction. Reverse engineering is a process of examination, not a process of change or replication. The key objective of reverse engineering a software system is to increase the comprehensibility of the system, to facilitate the maintenance and new developments. Reverse engineering is also useful to cope with the complexity of a system with the help of alternative views, recover lost information, synthesize higher abstractions, facilitate reuse, deduce software metrics and detect side effects.

At present there are several tools for software comprehension that can be used to apply reengineering, reverse engineering and other techniques. These tools can be used to visualize artifacts, compute software metrics and work in other properties. Examples of these tools are: SHriMP (Simple Hierarchical Multi Perspective) [19], Rigi [20], Portable BookShelf (PBS) [11], Moose [7], CodeCrawler [14-15]. More tools are listed in [17].

In this paper we show how the model driven engineering (MDE) approach can be used to develop software comprehension tools. In particular, we describe how a functionality of the CodeCrawler tool can be developed using an alternative approach; this approach proposed could be applied to any other tool.

This paper is divided in four major parts. The first section describes the general aspects of the CodeCrawler tool, including the format used to represent the information, and the Polymetric View concept, which is used in the example (study case) of the paper. The next section describes the study case by means of the details of the feature to be implemented, the proposed metamodels (source and target) and the transformation between them. After the study case there is a brief discussion of our findings and related work and finally, there is a section with the conclusions of the paper.

THE CODE CRAWLER TOOL

CodeCrawler [14-15] is a lightweight software visualization tool, the first implementation of which dates back to 1998. It has evolved into an information visualization framework and has been customized to work in several contexts (website reengineering, concept analysis and more) keeping a strong focus on software visualization (SV).

CodeCrawler is considered a language independent SV tool, because it uses the Moose reengineering environment [6] that implements the FAMIX metamodel [21], which models software written in C++, Java, Smalltalk, Ada, Python, COBOL, and others.

Figure 1 shows the general architecture of CodeCrawler where the core acts as a bridge between the visualization engine and the metamodel. This architecture which separates the three main parts (core, metamodel, visualization

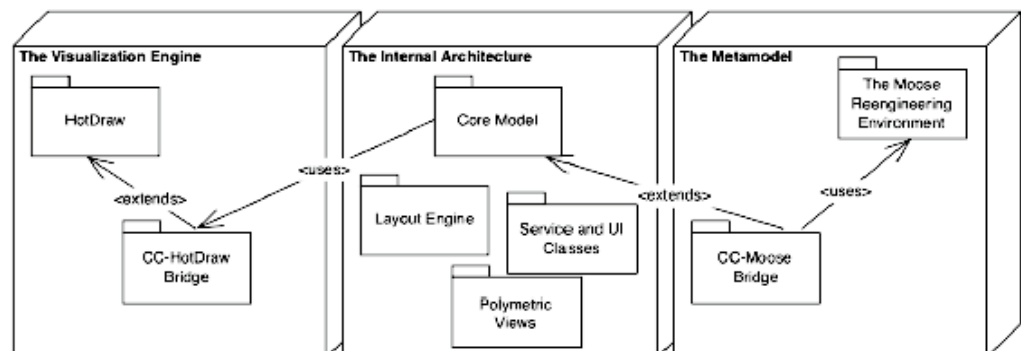


Figure 1. Architecture of CodeCrawler.

engine) of a software visualization tool allows for higher flexibility and greater extensibility.

The next sub-sections describe two elements of CodeCrawler that are use in this paper: the FAMIX metamodel as the standard format that represents information extracted from source code and the Polymetric View concept, which will be used in the study case.

The FAMIX Metamodel

FAMIX [21] is a metamodel for modeling object-oriented software. The main goal of FAMIX is to support reengineering activities in a language-independent way. The aim is not to cover all aspects of all languages, but rather to capture the common features that we need for reengineering activities, so tools can be easily reused for multiple target languages.

FAMIX was proposed as a part of the Moose system [6] and afterwards was used in the CodeCrawler system. The requirements considered for the design of the FAMIX metamodel included support for multiple languages, support for the whole reengineering life cycle, extensibility, scalability and information exchange.

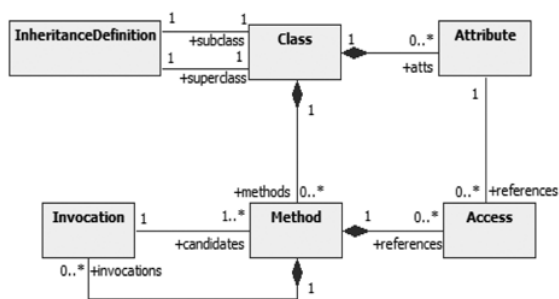


Figure 2. FAMIX core (adapted from [14]).

The FAMIX metamodel defines a set of classes associated to the source code entities available in a software project: packages, classes, methods, attributes, variables, formal parameters, calls, access, etc. Figure 2 shows the core main classes of the FAMIX metamodel and their relationships. The complete metamodel is available in [14] and a draft of the version 2.1 is available in [8].

Polymetric View

A polymetric view [14] is an approach to the construction of lightweight visualizations enriched with software metrics. A polymetric view is a two-dimensional visualization that uses nodes (rectangles) to display software entities or abstractions of them and uses edges

to represent relationships between the entities. This is a widely used practice in information visualization and software visualization tools. This basic visualization technique is enriched by rendering up to 5 metric measurements on a single node simultaneously:

- **Node Size:** The width and the height of a node can each render one metric measurement. The larger these measurements are, the larger the node is in one or both of the dimensions.
- **Node Color:** The color interval between white and black can be used to render another metric measurement. The convention is that the higher the metric value is, the darker the node is. Thus light gray represents a smaller metric measurement than dark gray.
- **Node Position:** The X and Y coordinates of the position of the node can also reflect two metrics measurements. Not all layouts can exploit position metrics, as some of them implicitly dictate the position of the nodes (for example, a tree layout).

An actual polymetric view depends on three ingredients: a layout (the choice of the displayed entities and their relationships), a set of metrics (up to 5 metrics for each node), and a set of entities (parts of the system that will be visualized).

A MDE APPROACH TO POLYMETRIC VIEWS

In the previous section we described the Polymetric View (PV) concept that is used to display graphs where each node is related to an source-code entity (class, method, variable, package, etc.) and can have up to five associated metrics (position x-y, width, height and color).

We selected the Inheritance Classification, a PV that shows a tree layout of nodes enriched with metrics information. In this PV the nodes are classes, while the edges represent the inheritance relationships between them. The size of the nodes reflects the number of methods added –NMA– (width) and the number of methods overridden –NMO– (height) of the classes, while the color tone represents the number of methods extended –NME– of the class. In this PV the position of the nodes does not reflect metric measurements, as the nodes' position is implicitly given by the tree layout.

The study case presented in this paper aims to show how we can use MDE techniques to develop a functionality of a well known tool in the program comprehension domain. We chose to compute the information of a PV starting

from a standard representation of a software system (in FAMIX, or a similar format) using a different approach than the traditional process used in the common tools available. In CodeCrawler the information of a PV is stored in a graph, which is the input of a visualization engine (see Figure 1).

The traditional approach used to develop this kind of tool is usually *ad-hoc* and a mixture of control code and domain logic; on the other hand, the approach proposed in this paper promotes a programming style that takes advantage of the domain concepts (in this case, source-code entities, graph elements or PV information, described in the metamodels) and the control is the responsibility of the transformation language engine and the host language used to implement the GUI. In this programming style, the models became more active in software development, unlike traditional software development where the models are used as documentation artifacts or implementation guidelines.

In order to reach the main objective, first, we must define the source and target metamodels, which can be used in a program written in a model-transformation language, to produce a model that conforms to the target metamodel (graph) starting from a model that conforms to the source metamodel (source code, FAMIX).

The next sub-sections describe the source metamodel (source code), the target metamodel (graph with metrics info) and the transformation that computes the PV information starting from the source code model.

Source Metamodel: Source Code

In order to apply MDE techniques, in this example we need a metamodel to define the structure of the information extracted from source code. For this example, a simplified metamodel of Java programs is proposed. This metamodel includes the information of the classes and their methods. Figure 3 shows the metamodel expressed in UML and it includes the main entities (Package, Class and Method) and the relationships between them. For the sake of

simplicity we have selected just the relationships needed to compute the metrics of the chosen PV. We chose this simple metamodel instead of FAMIX, because FAMIX could be too big for the example.

The metamodel expressed in UML notation is useful to understand the entities modeled and their relationships; nevertheless, a representation in a language more suitable to be processed by a program is required. In order to do this, we propose an alternative representation in the KM3 language.

KM3 [12] is a textual language to specify metamodels, and it can be used to derive metamodels represented in the standard Ecore [9]. KM3 is available as a part of the Atlas Model Management Architecture (AMMA) project [13] developed as a part of the Eclipse Modeling Project, which also includes Atlas Transformation Language (ATL) [1]. The aim of ATL is to provide a language and an environment to define transformations between metamodels.

Figure 4 shows the source code metamodel of Figure 3, expressed in KM3.

Target Metamodel: Graph

The visualization engine of CodeCrawler requires a graph that contains all the information to be displayed. Additionally it requires a layout that is deduced from the kind of PV to be used. The visualization graph contains the information about all the nodes and edges, which are associated with source-code entities in the definition of the PV. The graph metamodel used in this example is presented in Figure 5 (UML) and Figure 6 (KM3).

In this example, each node is associated with a source-code class and the edges represent inheritance relationships. The graph elements also contain the attributes of the PV. Inheritance Classification just needs the width, height and color of each node, the values of which are respectively the metrics NMA, NMO and NME.

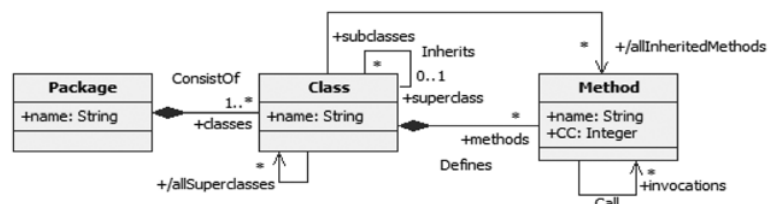


Figure 3. Source code metamodel (UML).

```

package SourceCode {
  class Package {
    attribute name : String ;
    reference classes[1-*] container :
      Class oppositeOf "package" ;
  }
  class Class {
    attribute name : String ;
    reference "package" :
      Package oppositeOf classes ;
    reference superclass[0-1] :
      Class oppositeOf subclasses ;
    reference subclasses[*] :
      Class oppositeOf superclass ;
    reference methods[*] container :
      Method oppositeOf "class" ;
    reference allSuperclasses[*] : Class ;
    reference allInheritedMethods[*] : Method ;
  }
  class Method {
    attribute name : String ;
    reference "class" :
      Class oppositeOf methods ;
    reference invocations[*] : Method ;
  }
}
package PrimitiveTypes {
  datatype String;
}

```

Figure 4. Source Code Metamodel (KM3).

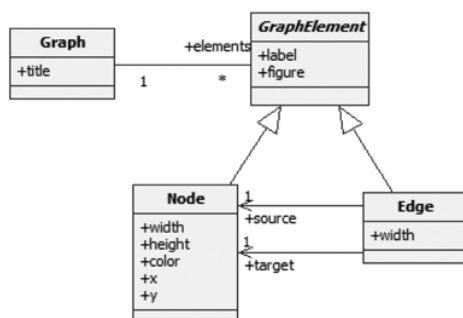


Figure 5. Graph metamodel (UML).

The definitions of each metric used are the following:

- Number of Methods Added by a subclass (NMA): A method is defined as an added method in a subclass if there is no method of the same name in any of its superclasses [16].
- Number of Methods Overridden by a subclass (NMO): Number of methods defined in a subclass with the same name of a method in one of its superclasses [16].
- Number of Methods Extended (NME): Number of methods redefined in subclass by invoking the same method on a superclass [14].

```

package GraphData {
  class Graph {
    attribute title : String ;
    reference elements[*] container :
      GraphElement ;
  }
  abstract class GraphElement {
    attribute label[0-1] : String ;
    attribute figure[0-1] : String ;
  }
  class Node extends GraphElement {
    attribute width[0-1] : Integer ;
    attribute height[0-1] : Integer ;
    attribute color[0-1] : Integer ;
    attribute x[0-1] : Integer ;
    attribute y[0-1] : Integer ;
  }
  class Edge extends GraphElement {
    attribute width[0-1] : Integer ;
    reference source : Node ;
    reference target : Node ;
  }
}
package PrimitiveTypes {
  datatype String;
  datatype Integer
}

```

Figure 6. Graph Metamodel (KM3).

The metrics are computed using the attributes and relationships of the source-code metamodel. An instance of the graph metamodel is derived from the source-code metamodel by means of a transformation. The next sub-section describes the transformations related to this example.

Transformations between the metamodels

In order to generate a graph model with the computed metrics of the PV, we need a transformation that takes as input the source-code metamodel, the graph metamodel and a model conform to the source-code metamodel. This transformation must generate a model that conforms to the target metamodel (graph).

We chose the Atlas Transformation Language (ATL) [1] to write a transformation that checks all the entities of the source model (a representation of a program that conforms to the source-code metamodel) and produces the entities of the target model. Additionally, the transformation uses a set of helpers to compute every metric needed for the PV of the example (Inheritance Classification).

Figure 7 contains the main transformation called Package2Graph and Figure 8 contains the rule Class2Node.

The rule Package2Graph (Figure 7) applies to each package of the source code (line 2 to 3) and generates a graph (line 5 to 11) with the name of the package (line 6) and a sequence of elements (lines 7 to 10): nodes (line 8) and edges (line 9).

```

1 rule Package2Graph {
2   from
3     p: SourceCode!Package
4   to
5     g: GraphData!Graph (
6       title <- p.name,
7       elements <- Sequence {
8         p.classes->collect
9         (c|thisModule.resolveTemp(c, 'class_node')),
10        p.classes->collect
11      }
12    )
13  }

```

Figure 7. ATL Transformation: rule Package2Graph.

The rule Class2Node (Figure 8) applies to every class of the source-code model (line 2 to 3) and generates one graph node per class (line 5 to 13) and as many edges as the number of subclasses of the current class (lines 14 to 21). The node attributes width, height and color are associated with their respective metrics (lines 8 to 10).

```

1 rule Class2Node {
2   from
3     c: SourceCode!Class
4   to
5     class_node: GraphData!Node (
6       label <- 'Class: ' + c.name,
7       Figure <- 'default_node',
8       width <- c.NMA(),
9       height <- c.NMO(),
10      color <- c.NME(),
11      x <- 0,
12      y <- 0
13    ),
14    class_edges: distinct GraphData!Edge
15    foreach (subclass in c.subclasses) (
16      label <- c.name+'->'+subclass.name,
17      Figure <- 'default_edge',
18      width <- 0,
19      source <- c,
20      target <- subclass
21    )
22 }

```

Figure 8. ATL Transformation: rule Class2Node.

The metrics are computed by three helpers that are shown in Figure 9. The attributes x and y are not computed by the transformation because this PV does not use this information (the position of each node is deduced from the tree layout). Some edge attributes, in particular the width, are not needed in this PV.

To generate another PV, the process to follow is similar to that applied in this example. The main differences are the values (metrics) used to the attributes of a node. This process can be generalized and automated to any PV, using an environment to configure PVs, selecting the metrics from a list. The metrics could be in a repository. The ATL transformation to generate the information about a PV could be automatically generated by means of a GUI, a domain specific language (DSL) or a high-order transformation.

RELATED WORK

There are different research groups using MDE techniques in the software evolution domain. Bézivin and his colleagues [3] show how to increase the tool interoperability; in particular, a general metamodel for the data of bugtracking tools is defined and an example is presented where the data of several tools (Bugzilla, Mantis and Excel) is translated between them. In [2], Antoniol and his colleagues combine data extracted from source code, CVS repositories and bugtracking tools using metamodels. In [18] MDE techniques were used to visualize bugtracking data.

Bull and his colleagues [4] use MDE techniques to assist with the creation of highly customizable interfaces for software visualization; in particular, an example is presented where some Java source code metrics are computed

```
-- Number of Methods Added (NMA) metric, for a given class
helper context SourceCode!Class def : NMA() : Integer =
  let setOfClassMethodsNames : Set(String) =
    self.methods->collect(m | m.name).asSet() in
    let setOfClassInheritedMethods : Set(String) =
      self.allInheritedMethods->collect(m | m.name).asSet() in
      (setOfClassMethodsNames - setOfClassInheritedMethods).size();

-- Number of Methods Overridden (NMO) metric, for a given class
helper context SourceCode!Class def : NMO() : Integer =
  let setOfClassMethodsNames : Set(String) =
    self.methods->collect(m | m.name).asSet() in
    let setOfClassInheritedMethods : Set(String) =
      self.allInheritedMethods->collect(m | m.name).asSet() in
      setOfClassMethodsNames.intersection(
        setOfClassInheritedMethods).size();

-- Number of Methods Extended (NME) metric, for a given class
helper context SourceCode!Class def : NME() : Integer =
  self.methods->iterate( m ; count : Integer = 0 |
    count + m.invocations->select( inv | inv.name = m.name and
      m.class.allSuperclasses->includes(inv.class))->size()
  );
```

Figure 9. Helpers to compute metrics.

and they are visualized with trees, bar diagrams and directed graphs. The approach followed is totally based on metamodels and transformations.

Girba [10] proposed a generic metamodel for software evolution that takes into account the history of a software artifact as a first order entity. A history is an ordered set of versions. Both historial and versions are generic concepts that can be applied to any kind of artifact, like classes or packages.

CONCLUSION

The study case used in this paper is useful to show that MDE techniques could be applied in the reverse engineering and program comprehension domain.

Metamodels promote the use of domain-concepts instead of programming concepts; consequently, this domain-oriented engineering promotes the definition of domain-specific languages (DSL). Additionally, using metamodels facilitates the domain comprehension and can even be used to share a common structure between different applications, increasing the interoperability. To attain this, we require standards to define and represent metamodels (such as KM3 and Ecore) and languages to define transformations between metamodels (like ATL).

As we state in the study case, the process used in this example can be generalized and used to define a setup environment for polymetric views, where the configuration of metrics can be selected from a list (library) and the code of the ATL transformation can be automatically generated. Additionally, the steps followed to develop the polymetric view feature can be applied in a similar way to develop the functionalities of a software comprehension tool.

ACKNOWLEDGEMENTS

Thanks to the Consejo de Desarrollo Científico y Humanístico (Universidad Central de Venezuela, Caracas, Venezuela) for its support to the development of this research work.

REFERENCES

- [1] F. Allilaire, J. Bézivin, M. Didonet, F. Jouault, D. Touzet et P. Valduriez. "AMMA: vers une plateforme générique d'ingénierie des modèles". *Génie Logiciel*. Vol. 73, pp. 8-15. 2005.
- [2] G. Antoniol, M. Di Penta, H. Gall and M. Pinzger. "Towards the Integration of CVS Repositories, Bug Reporting and Source Code Meta-models". *Electronics Notes in Theoretical Computer Science*, pp. 87-99. 2005.

- [3] J. Bézivin, H. Brunelière, F. Jouault and I. Kurtev. "Model Engineering Support for Tool Interoperability". 4th Workshop in Software Model Engineering, WiSME 2005 (Satellite event of MoDELS 2005). Montego Bay, Jamaica. October, 2005.
- [4] R.I. Bull, M.-A. Storey, J.-M. Favre and M. Litou. "An Architecture to Support Model Driven Software Visualization". International Conference on Program Comprehension, ICPC 2006. IEEE. Athens. June 14-16, 2006.
- [5] E.J. Chikofsky and J.H. Cross. "Reverse Engineering and Design Recovery: A Taxonomy". IEEE Software. Vol. 7 N° 1, pp. 13-17. IEEE Computer Society. January 1990.
- [6] S. Ducasse, M. Lanza and S. Tichelaar. "Moose: an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems". In Proceedings of the Second International Symposium on Constructing Software Engineering Tools (CoSET 2000). June 2000.
- [7] S. Demeyer, S. Tichelaar and P. Steyaert. "Famix 2.0". Date of visit: January 2009. URL: <http://scg.unibe.ch/archive/famoos/FAMIX/Famix20/Html/famix20.html>
- [8] Moose Technology: FAMIX. Software Composition Group. Date of visit: January 2009. URL: <http://moosetechnology.org/docs/famix>
- [9] The Eclipse Modeling Framework (EMF) Overview. Date of visit: January 2009. URL: <http://www.eclipse.org/emf>
- [10] T. Girba and S. Ducasse. "Modeling History to Understand Software Evolution". International Journal on Software Maintenance and Evolution: Research and Practice (JSME). Vol. 18, pp. 207-236. 2006.
- [11] R. Holt. "Software Bookshelf: Overview and Construction". March 1997.
- [12] F. Jouault and J. Bézivin. "KM3: a DSL for Metamodel Specification". Proceedings of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems. Bologna, Italy. 2006.
- [13] F. Jouault, J. Bézivin, C. Consel, I. Kurtev and F. Latry. "Building DSLs with AMMA/ATL: a Case Study on SPL and CPL Telephony Languages". Proceedings of the first ECOOP Workshop on Domain-Specific Program Development (DSPD). Nantes, France. July 3rd, 2006.
- [14] M. Lanza. "CodeCrawler-Lessons Learned in Building a Software Visualization Tool". CSMR 2003 Proceedings (7th European Conference on Software Maintenance and Reengineering). IEEE Computer Society, pp. 409-418. 2003.
- [15] M. Lanza. "CodeCrawler - Polymetric views in action". Proceedings of ASE 2004 (19th IEEE International Conference on Automated Software Engineering). IEEE CS Press, pp. 394-395. 2004.
- [16] M. Lorenz and J. Kidd. "Object-oriented Software Metrics". Prentice Hall Object-Oriented Series, 1994.
- [17] H. Müller, J. Jahnke, D. Smith, M.-A. Storey, S. Tilley and K. Wong. "Reverse Engineering: A Roadmap". International Conference on Software Engineering (Proceedings). Limerick, Ireland, pp. 47-60. ISBN: 1-58113-253-0. 2000.
- [18] E. Scalise, J.-M. Favre and N. Zambrano. "Desarrollo de una Herramienta para la Ingeniería en Reverso Dirigida por Modelos: Visualización de Artefactos de Implementación". XXXIII Conferencia Latinoamericana de Informática. San José, Costa Rica. 9-12 de octubre de 2007.
- [19] M.A. Storey, C. Best, J. Michaud, D. Rayside, M. Litoiu and M. Musen. "SHriMP views: an interactive environment for information visualization and navigation. Proceedings of CHI 2002". Conference Extended Abstracts on Human Factors in Computer Systems. Minneapolis, Minnesota, USA, pp. 520-521. April 20-25, 2002.
- [20] M.A. Storey, K. Wong and H.A. Müller. "Rigi: a visualization environment for reverse engineering". International Conference on Software Engineering (ICSE'97), pp. 606-607. May 1997.
- [21] S. Tichelaar. "Modeling Object-Oriented Software for Reverse Engineering and Refactoring". PhD thesis. University of Berne. December 2001.