Beyond Language Independent Object-Oriented Metrics: Model Independent Metrics

Michele Lanza
lanza@iam.unibe.ch
Software Composition Group
Universitá di Berna, Svizzera

Stéphane Ducasse
ducasse@iam.unibe.ch
Software Composition Group
Université de Berne, Suisse

ABSTRACT

Software Metrics have become essential in software engineering for several reasons, among which quality assessment and reengineering. In the context of the European Esprit Project FAMOOS, whose main goal was to provide methodologies for the reengineering of large industrial software systems, we have developed the Moose Reengineering Environment, based on the language independent FAMIX metamodel. Moose includes a metrics engine which supports language independent metrics, since coping with software written in different implementation languages was one of the project's main constraints. Our current research is pushing us towards the development and implementation of a metametamodel, which would include our metamodel and allow for several extension in different research directions, among which concept analysis, knowledge management and software evolution. In this article we want to present our current and future work for the transition from language independent to domain independent metrics.

Keywords: Software Metrics, Object-Oriented Programming, Language Independence

1. INTRODUCTION AND MOTIVATION

Metrics have become essential in some disciplines of software engineering. In *forward engineering* they are being used to measure software quality and to estimate cost and effort of software projects [15]. In the field of *software evolution*, metrics can be used for identifying stable or unstable parts of software systems, as well as for identifying where refactorings can be applied or have been applied [9], and for detecting increases or decreases of quality in the structure of evolving software systems. In the area of *software reengineering* and *reverse engineering* [7], metrics are being used for assessing the quality and complexity of software systems, as well as getting a basic understanding and providing clues about sensitive parts of software systems.

If we restrict ourselves to the field of object-oriented reengineering and object-oriented metrics, we see that this field is fairly new. In the past decade a great body of research has been conducted [5, 17, 16] and various metric suites have been proposed [21, 6], which cover most of the relevant aspects of object-oriented software.

In the context of the European FAMOOS Esprit Project, whose main goal was to investigate methodologies and create tools to reengineer large software systems, and whose main results have been summarized in [12, 10], we were confronted with several large scale industrial systems written in different languages, among which C++, Java, Smalltalk and Ada.
Therefore we created the Moose Reengineering Environment [13], a repository and tool environment, based on the language independent FAMIX metamodel [11]. Source code in one of the aforementioned languages is mapped to a language independent representation, which we present in more detail in the next section.
The Moose Reengineering Environment included a metrics engine, which was first implemented to perform simple queries and then greatly extended to support the development of CodeCrawler, a software visualization tool which integrates metrics in its visualizations [18, 8]. The extended metrics engine included a suite of language independent metrics, whose computation was based exclusively on the language independent metamodel representation of software artifacts.
As we are now starting the implementation of a metametamodel, where apart from artifacts defined in the FAMIX metamodel, non-software artifacts can as well be accomodated, the need has arisen to extend the language independent metrics suite to become a model independent metrics suite.

In this article we present and discuss our current state of research and a first implementation of a metametamodel, and give an outlook on our future work in this field.

Structure of this Document

This paper is structured as follows: In the next section we present a short overview on our previous work in the field of reengineering and reverse engineering with a special focus on software metrics. We then abstract from this context and come up with model independent metrics, and show that these metrics cover all previously presented metrics. We then present Moose and CodeCrawler, the software applica-

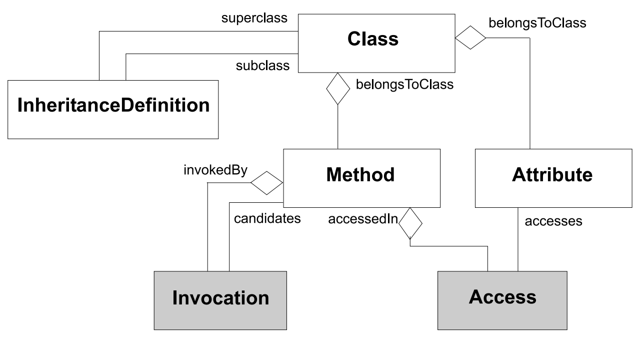


Figure 1: A simplified view of the FAMIX metamodel.

tions we are currently developing to support our research. Next we report on our experiences with a first implementation of model independent metrics. Finally, we conclude by analyzing benefits and limits of our approach and present our future work in this area.

2. LANGUAGE INDEPENDENT OBJECT-ORIENTED SOFTWARE METRICS

In the context of the European FAMOOS project we created the Moose Reengineering Environment, which implements the FAMIX metamodel and serves as repository for software artifacts and common development base for various tools, among which CodeCrawler.

A simplified view of the FAMIX metamodel comprises the main object-oriented concepts - namely class, method, attribute and inheritance - plus the necessary associations between them - namely method invocation and attribute access (see Figure 1). Note that the actual FAMIX metamodel contains nearly all possible software artifacts, including method parameters and local variables. The FAMIX metamodel does not include the actual source code, however it provides easy access to the actual source code by means of source anchors that point to the right location, for example the position within a source file.

In Table 1 and Table 2 we list a selection of metrics which can be computed by our metrics engine. The metrics are divided into class, method and attribute metrics, i.e., these are the entities that the metric measurements are assigned to. Moose includes a metrics engine with more than 50 different metrics, among which ca. 30 language-independent metrics.
In bold we have emphasized the few language-dependent metrics, which depend mainly on language-specific constructs and properties, like modifiers for C++ and Java or method protocols for Smalltalk source code. Note that metrics which are sums of language dependent metrics (for example WLOC and LOC) can be considered language independent.

The computation of the language independent metrics is based solely on our repository of software entities and is therefore by definition language independent.

To give a short example, suppose that Moose contains a class *foo* and two methods *bar* and *tender*. As we have seen in Figure 1 a method entity has a relationship called *belongsToClass*, which points to the corresponding class. If both methods in our example point to the class *foo* we can infer that the number of methods (NOM) metric of *foo* will have the value 2.

Thus the computation of the metrics based solely on the metamodel representation of source code can be conceived as a traversal of the graph depicted in Figure 1. The nodes in the graph are source code artifacts, while the edges represent relationships between those artifacts. In the FAMIX metamodel the main entities are class, method and attribute and the main relationships inheritance, access (from method to attribute) and invocation (between methods). Added to this we have containment relationships, i.e., a class contains a set of methods and attributes.

Added to this we also have a set of properties on the entities which can also be boolean. For example a class has the property *isAbstract* or a method has one of the properties *isAccessor, isOverriding, isExtending, etc.*. The entities

Name	**Description**
	Class Metrics
HNL	Number of classes in superclass chain of class
NAM	Number of abstract methods
NCV	Number of class variables
NIA	Number of inherited attributes
NIV	Number of instance variables
NME	Number of methods extended, i.e., redefined in subclass by invoking the same method on a superclass
NMI	Number of methods inherited, i.e., defined in superclass and inherited unmodified by subclass
NMO	Number of methods overridden, i.e., redefined compared to superclass
NOA	Number of attributes (NOA = NIV + NCV)
NOC	Number of immediate subclasses of a class
NOM	Number of methods
NOMP	Number of method protocols (in Smalltalk)
PriA	Number of private attributes (equivalent for protected and public attributes)
PriM	Number of private methods (equivalent for protected and public methods)
WLOC	Sum of LOC over all methods
WMSG	Sum of message sends in a class
WNMAA	Number of all accesses on attributes
WNOC	Number of all descendant classes
WNOS	Sum of statements in all method bodies of class
WNI	Number of invocations of all methods

Table 1: A list of the class metrics contained in the metrics engine of the Moose Reengineering Environment.

Name	Description
Method Metrics	
LOC	Method lines of code
NMA	Number of methods added, i.e., defined in subclass and not in superclass
MHNL	Class HNL in which method is implemented
MSG	Number of method message sends
NOP	Number of (input) parameters
NI	Number of invocations of other methods within method body
NMAA	Number of accesses on attributes
NOS	Number of statements in method body
Attribute Metrics	
AHNL	Class HNL in which attribute is defined
NAA	Number of times directly accessed

Table 2: A list of the method and attribute metrics contained in the metrics engine of the Moose Reengineering Environment.

have been implemented in an extensible way, i.e., each entity has a field of freely definable properties which we compute by means of operators, i.e., we iterate over the entities of a metamodel, compute properties and add those properties to the entities. Note that the metrics themselves have been modeled as properties.

3. MODEL INDEPENDENT METRICS

In this section we first present the construction of model independent metrics. The construction of these metrics is based on the traversal of the metamodel graph, for example like the one depicted in Figure 1. We do this by defining three generic metrics, namely **NodeCount** , **EdgeCount** *, and* **PathLength** also denoted by the acronyms NC, EC, and PL. Starting from these generic metrics a large number of object-oriented metrics can be defined in a general, flexible and extensible way, as we show by providing concrete examples. To increase the readability and to stay within the limits of this position paper we omit formal definitions of the metrics and provide smalltalk source code examples instead. However, we plan to eventually add the formal definitions in an extended future version of this paper.

3.1 Generic Metrics

3.1.1 NodeCount

NodeCount can be defined as the number of nodes connected to a node over a certain kind of edge, i.e., the number of nodes connected to a certain node over containment relationship edges or invocation edges, etc. Note that each node and edge in the graph knows about its neighboring nodes and edges, therefore after constructing the graph it is easy to get the necessary collections of nodes and edges necessary for our metrics computations.

Examples. The number of methods (NOM) of a class c is equal to the number of method nodes contained in class Node c. The number of accessed attributes by a method m is equal to the number of attribute nodes connected to node m by means of access edges. The number of invoked methods by a method m is equal to the number of method nodes connected to node m by means of invocation edges.

Furthermore, **NodeCount** can be extended to satisfy certain properties, by selecting or rejecting counted nodes depending on whether these nodes fulfill certain properties.

Examples. The number of abstract methods (NAM) of a class c is equal to the number of method nodes contained in class node c and which fulfill the property *isAbstract.* Similarly the number of concrete methods can either be defined by reject abstract methods or by using the property *isConcrete.*

Note that this definition of **NodeCount** will ignore duplicates, i.e., if method m accesses attribute a twice, this will be counted as only one accessed attribute, since the metric measures the number of accessed attributes and not the number of attribute accesses performed by method m. To obtain the latter metric, we need the generic metric **EdgeCount** , that we discuss below.

3.1.2 EdgeCount

Similar to **NodeCount** we can define a generic metric **EdgeCount** for counting edges. **EdgeCount** can be defined as the number of edges attached to a node. The edges can have an arbitrary type.

Examples. The number of attribute accesses performed by method m is equal to number of access edges attached to method node m. The number of method invocations performed by method m is defined accordingly.

3.1.3 PathLength

The last generic metric **PathLength** deals with chains of edges of the same type. **PathLength** can be defined as the length of the chain of edges of a certain type starting in a certain node.

Examples. The depth level (HNL) of a class c within an inheritance hierarchy is equal to the length of the chain of inheritance relationships starting from class node c. The maximum method invocation depth within a class c is equal to the maximum **PathLength** of method invocations computed over all method nodes contained in class node c.

3.2 Derived Generic Metrics

We can extend the generic metrics in order to allow us to compute more complicated metrics, among which *transitive, ratio* and *promoted* metrics.

3.2.1 Transitive Metrics

Transitive metrics can be seen as extensions to the generic metrics, by traversing the metamodel graph over different kinds of nodes and edges and by counting the things relevant to the metric.

Examples. The number of attributes accessed by a class c, e.g., by the methods contained in class c is a transitive **NodeCount** metric which traverses the graph from class node c over its contained method nodes $m_1, m_2, ...$ to the attribute nodes $a_1, a_2, ...$ attached to the method nodes by means of access edges.

3.2.2 *Promoted Metrics*

The generic metrics are absolute metrics, i.e., their values will be positive integers including zero. Promoted metrics are obtained by taking two or more (not necessarily generic) metrics and performing arithmetic operations on them. Prominent examples of promoted metrics are percentage, average and summation metrics.

Examples. The average number of accessed attributes by the methods contained in class c will be defined as the number of attributes accessed by class c divided by the number of methods contained in class c.

4. CODECRAWLER AND MOOSE

In this section we shortly present the Moose Reengineering Environment and CodeCrawler, its most prominent tool, which also was the main cause for our current work in the field of metrics.

4.1 The Moose Reengineering Environment

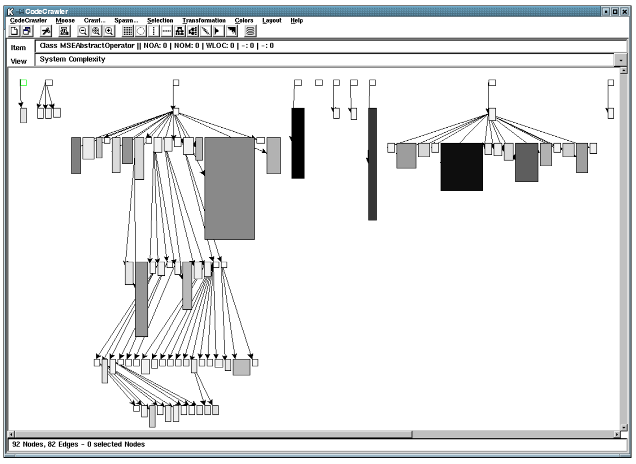
Moose is a language independent reengineering environment written in Smalltalk. It is based on the FAMIX metamodel [11], which provides for a language independent representation of object-oriented sources and contains the required information for the reengineering tasks performed by our tools. It is *language independent*, because we need to work with legacy systems written in different implementation languages. 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, we allow for language plug-ins that extend the model with language-specific features. Next to that, we allow tool plug-ins to extend as well the model with tool-specific information.

4.2 CodeCrawler

CodeCrawler supports reverse engineering through the combination of metrics and visualization [18, 8, 19]. It visualizes entities as nodes and relatioships as edges, and renders up to five metrics on the nodes by using their width, height, color and position on the display. Through these simple visualization enriched with the large metrics suite provided by Moose, it enables the user to gain insights in large systems in a short time. CodeCrawler is a tool which works best when unknown systems are approached and quick insights are needed for reverse engineering. CodeCrawler has been successfully tested on several industrial case studies.

4.3 A Model Independent Metrics Suite

The motivation to implement a metametamodel came mainly from the success of CodeCrawler: as it evolved and got faster and more scalable, it was able to visualize larger and larger systems. At one point, a new implementation of CodeCrawler permitted to easily visualize non-FAMIX entities. In some experiments we visualized for example concepts from the field of concept analysis or prolog facts from a prolog engine. However, as one of CodeCrawler's main aspects is the way it combines software visualization


CodeCrawler
CodeCrawler Moose Crawl... Spawn... Selection Transformation Colors Layout Help

Item Class MSEAbstractOperator || NOA: 0 | NOM: 0 | WLOC: 0 | -: 0 | -: 0

View System Complexity

92 Nodes, 82 Edges - 0 selected Nodes

Figure 2: A screenshot taken from CodeCrawler. In this view we see a display of inheritance hierarchies of classes. The width and height of the class nodes represent the number of attributes, respectively the number of methods of the classes, while the color reflects the number of lines of code of the classes.

with metrics, the need arose for a more general metrics engine. Although the current metrics engine is flexible and easily extensible, it is not general enough, and would require extensions each time new kinds of entities have to be accomodated into CodeCrawler or Moose.

5. DISCUSSION

In this section we first draw some conclusions on our current work and discuss limits and benefits of our approach. We then discuss related work which has been performed in this field and end by giving an outlook on the future work we plan to do in the presented context.

5.1 Conclusion

In this paper we have presented a simple approach to generate model independent metrics. We have started by recapitulating our experiences with language independent object-oriented software metrics and have then abstracted from that context to come up with a general way to create metrics independently from the underlying metamodel. We have reported on some first experiences regarding the implementation of a model independent metrics suite within our metametamodel.

The benefits of our approach are the increased flexibility, i.e., we can introduce new metamodels from arbitrary contexts (for example knowledge management, the financial world, databases, etc.) which provides us with a standard metrics suite without having to implement new metrics each time such a new context is introduced.

The limits of our approach are that currently not all object-oriented software metrics can be defined in terms of our model independent metrics definitions. Certain metrics tend to be very specialized and are thus difficult to define in a generic way. Another problem is that for some metrics there is still no consensus about what is the best way to define them. For these reasons we did not consider *coupling metrics* (such as NCR [21], CF [4], CBO and RFC [6]) and *cohesion metrics* (such as LCOM [14, 6, 20], CR [1] and CAMC [2]). Furthermore, as, due to our research in reengineering, we come from a more pragmatic side, it is not solely the definition of the metrics which interest us. Indeed, it is the creation of concrete metrics, which we can then use for instance in CodeCrawler, which interests us. In this context other problems arise, whose solutions lie mainly at the implementation level: what are the names of those metrics, how many can we generate and when does it make sense to stop generating metrics? Are the generated metrics actually usable and which ones do make sense in the given contexts? We plan to answer some of these questions in our future work.

5.2 Related and Future Work

A great body of research has been performed on the subject of metrics. We limit ourselves to list the most important references.

Metrics have long been studied as a way to assess the quality and complexity of software [15], and recently this has been applied to object-oriented software as well [21, 16]. Metrics profit from their scalability and, in the case of simple ones, from their reliable definition.
In [3] a mechanism is provided for comparing measures and their potential use, integrating existing measures which examine the same concepts in different ways, and facilitating more rigorous decision making regarding the definition of new measures. This paper also provides an excellent state-of-the-art which highlights that many measures have unclear definitions, as should be the case in measurement theory.
In [22] a generic metamodel is used to define metrics that abstract away from the particular metamodel elements. Because of this, the generic metrics are automatically available for all the metamodels (such as UML or FAMIX [11] that are mapped to the generic metamodel.

For reasons of simplicity, we did not provide subtype relationships in our metamodel. However, most metamodeling approaches (such as UML) make use of subtypes. By exploiting this subtype information we can make our framework even more generic.
At this time we only use properties for the nodes. In some cases it could also be useful to use properties on the edges. For example, we can make a distinction between access edges which retrieve or change the value of an attribute.
In the future we plan to extend our metametamodel and its model independent metrics engine. We believe this will be pushed more as soon as we begin to integrate different metamodels in our metametamodel. The first experiences with the domains of concept analysis and logic languages have been promising.

Acknowledgements. We would like to thank Roel Wuyts for comments on drafts of this paper.

6. REFERENCES

[1] N.V. Balasubramanian. Object-oriented metrics. In *Proc. 3rd Int'l Asia-Pacific Software Engineering Conf. (ASPEC '96)*, pages 30–34. IEEE Computer Society Press, 1996.

[2] Jagdish Bansiya, Letha Etzkorn, Carl Davis, and Wei Li. A class cohesion metric for object-oriented designs. *Journal of Object-Oriented Programming*, 11(8):47–52, January 1999.

[3] Lionel C. Briand, John W. Daly, and Jürgen K. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, 1999.

[4] F. Brito e Abreu, M. Goulao, and R. Esteves. Toward the design quality evaluation of object-oriented software systems. In *Proc. 5th Int'l Conf. Software Quality*, pages 44–57, October 1995.

[5] Shyam R. Chidamber and Chris F. Kemerer. Towards a metrics suite for object oriented design. In *Proceedings OOPSLA '91, ACM SIGPLAN Notices*, pages 197–211, November 1991. Published as Proceedings OOPSLA '91, ACM SIGPLAN Notices, volume 26, number 11.

[6] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[7] Elliot J. Chikofsky and James H. Cross, II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, pages 13–17, January 1990.

[8] 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 (6th Working Conference on Reverse Engineering)*. IEEE, October 1999.

[9] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. Finding refactorings via change metrics. In *Proceedings of OOPSLA'2000, ACM SIGPLAN Notices*, pages 166–178, 2000.

[10] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns.* Morgan Kaufmann, 2002. to appear, spring 2002.

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

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

[13] Stéphane Ducasse, Michele Lanza, and Sander Tichelaar. The moose reengineering environment. *Smalltalk Chronicles*, August 2001.

[14] Letha Etzkorn, Carl Davis, and Wei Li. A practical look at the lack of cohesion in methods metric. *Journal of Object-Oriented Programming*, 11(5):27–34, September 1998.

[15] Norman Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach.* International Thomson Computer Press, London, UK, second edition, 1996.

[16] Brian Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity.* Prentice-Hall, 1996.

[17] M. Hitz and B. Montazeri. Chidamber and kemerer's metrics suite; a measurement theory perspective. *IEEE Transactions on Software Engineering*, 22(4):267–271, April 1996.

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

[19] Michele Lanza and Stéphane Ducasse. A categorization of classes based on the visualization of their internal structure: the class blueprint. In *Proceedings of OOPSLA 2001*, pages 300–311, 2001.

[20] W. Li and S. Henry. Object oriented metrics that predict maintainability. *Journal of System Software*, 23(2):111–122, 1993.

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

[22] Vojislav B. Mišić and Simon Moser. From formal metamodels to metrics: An object-oriented approach. In *Proc. Technology of Object-Oriented Languages and Systems (TOOLS-24).* IEEE Computer Society Press, 1998.