Electronic Notes in Theoretical Computer Science 72 No. 2 (2002)
URL: `http://www.elsevier.nl/locate/entcs/volume72.html` 12 pages

A Graph-Based Metamodel for Object-Oriented Software Metrics

Tom Mens [1], [2]

Programming Technology Lab
Vrije Universiteit Brussel, Belgium

Michele Lanza [3]

Software Composition Group
University of Bern, Switzerland

Abstract

Metrics are essential in object-oriented software engineering for several reasons, among which quality assessment and improvement of development team productivity. While the mathematical nature of metrics calls for clear definitions, frequently there exist many contradicting definitions of the same metric depending on the implementation language. We suggest to express and define metrics using a language-independent metamodel based on graphs. This graph-based approach allows for an unambiguous definition of generic object-oriented metrics and higher-order metrics. We also report on some prototype tools that implement these ideas.

1 Introduction

Metrics are essential in several disciplines of software engineering. In *forward engineering* they are used to measure software quality, to pinpoint anomalies and to estimate cost and effort. In *software reengineering*, metrics are useful for getting a basic understanding and providing higher level views of the software, and for finding violations of good software design. In the context of *software evolution*, metrics can identify stable and unstable parts of the software, identify where refactorings should be applied [18] or have been applied [7], and identify increases or decreases in the quality of software.

Many object-oriented metric suites have been proposed [3,5,12,15], and new metrics or variants of existing ones are invented every day. The many,

[1] Tom Mens is a PostDoctoral Fellow of the Fund for Scientific Research - Flanders
[2] Email: `tom.mens@vub.ac.be`
[3] Email: `lanza@iam.unibe.ch`

partly contradicting, definitions of the metrics make it difficult to write general metric tools, since they need to be updated every time we want to incorporate new metrics. To tackle this problem, we propose a *language-independent metrics framework* based on a graph representation of the software. Since the metrics are defined in a generic way, independent of the particularities of the user-specified metamodel, they can be applied to any object-oriented programming language.

This paper is structured as follows. In the next section we represent object-oriented software in terms of graphs. Based on this underlying representation, the subsequent section introduces a schema to define generic metrics and shows how typical object-oriented metrics can be defined using that schema. We do the same for a number of higher-order metrics. Next we discuss the current limitations of the metrics framework, and present some related work. Finally, we discuss tool support issues and conclude.

2 Representing Object-Oriented Software as Graphs

As an underlying software representation we use *graphs*, because they can capture the basic structure in a straightforward and generic way: nodes represent software entities and edges represent relationships between those entities.

More precisely, we will use *typed, directed and attributed multigraphs.* The use of *multigraphs* indicates that there can be more than one edge, possibly of the same type, between two nodes. We use a *type graph* to specify the object-oriented metamodel. Nodes can be nested in other nodes, which is indicated by a *contains* edge type. We use *directed graphs*, implying that each edge has exactly one source node and target node. We use *attributed graphs* to attach any number of domain-specific properties to the nodes and edges, although in this paper we only attach properties to nodes.

2.1 An Object-Oriented Metamodel

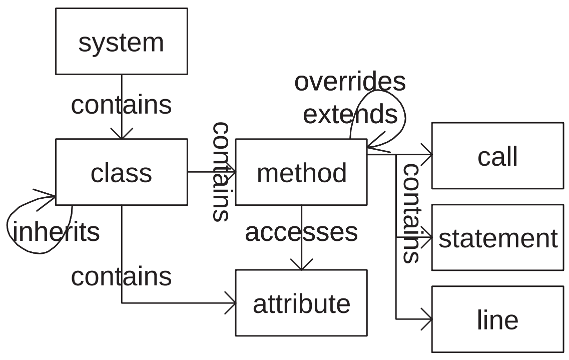

Fig. 1. An object-oriented metamodel.

In order to apply the graph representation to a particular object-oriented language, a metamodel needs to be provided that specifies the kind of software entities that are allowed and the relationships between them. For the sake of readability, Figure 1 shows a simplified subset of the metamodels that we used during our experiments. For example, it does not contain information about method parameters, and does not distinguish between read and write accesses. However, it is important to note that the approach presented in this paper is not specific to a particular metamodel. In fact, it is generally applicable to any representation of the source code whose metamodel can be expressed as a type graph that specifies the types of nodes and edges that are allowed and how they are related.

In the remainder of this text, η_i represents an arbitrary node type (such as *class*, *method* or *attribute*), and ε_i represents an arbitrary edge type (such as *contains* or *accesses*). As an abbreviation, we call a node of type η_i a η_i-node, and an edge of type ε_i an ε_i-edge. For example, a *system*-node represents the entire software system, and a *call*-node represents a method call or message send. A *contains*-edge represents an aggregation or composition relationship, directed from container to contained element. We will use the over-left-arrow notation to denote edge types that are followed in the opposite direction. For example, $\overleftarrow{contains}$ denotes a *contains*-edge that is followed from contained element to container.

Since we use attributed graphs, we can attach specific properties to nodes. For example, a *class*-node can have properties $abstractness(abstract \mid concrete)$ and *visibility*(*private* | *protected* | *public*). A *method*-node can have the same properties as well as the following ones: *scope*(*instance* | *class*) and $defType(overrides \mid extends \mid defines)$. An *attribute*-node can have properties *scope*(*instance* | *class*) and *visibility*(*private* | *protected* | *public*).

Based on the semantics of the used metamodel, additional constraints have to be specified between the node types and the edge types. For example:

- A *method*-node with property *defType(overrides)* (resp. *defType(extends)*) must be the source of an *overrides*-edge (resp. *extends*-edge).
- A *method*-node with property *defType(defines)* cannot be the source of any *overrides*-edge or *extends*-edge.
- *Overrides*-edges and *extends*-edges between *method*-nodes are only allowed if there is an *inherits*-edge between the *class*-nodes that contain these *method*-nodes.

In the remainder of the paper, we will assume to be working with a concrete graph G that satisfies the type graph of Figure 1, i.e., it is a valid instance of the metamodel. The node set of G will be called N and its edge set E. For any given edge $e \in E$ we use $source(e)$ to retrieve its source node, $target(e)$ for the target node, and *type*(e) for the edge type. For any given node $n \in N$ we use *type*(n) to retrieve the node type, and *property*(n, p) to get the value for a specific property p.

3 Representing Object-Oriented Metrics

In this section we present our graph-based schema for generic metrics. We then combine this schema with the object-oriented metamodel of Figure 1 to express typical object-oriented metrics in terms of the generic ones.

3.1 Generic Metrics

Based on our idea of representing software as graphs we can define three generic metrics (***NodeCount***, ***EdgeCount*** and ***PathLength***). Starting from these generic metrics a large number of object-oriented metrics can be defined in a general, flexible and extensible way, as we will show in the next subsection. In the remainder of this paper we use the acronyms ***NC***, ***EC*** and ***PL*** to denote these generic metrics.

The generic node count metric $NC(id, \eta_1, \eta_2, \Phi)$ computes the number of η_2-nodes that satisfy predicate Φ and that are contained in η_1-node id. The predicate $\Phi : N \rightarrow Boolean$ provides a filter when counting the number of nodes. It is an optional argument. If it is not provided we use the predicate that always returns true, which means that all nodes are counted. # stands for "number of".

Definition 3.1 (NodeCount) Let $n \in N$ and $type(n) = \eta_1$.
$NC(n, \eta_1, \eta_2, \Phi) := \#\{m \in N \mid type(m) = \eta_2 \wedge \Phi(m) \wedge$
$\quad \exists e \in E : source(e) = n, target(e) = m, type(e) = contains\}$

Example 3.2 $NC(c, class, method)$ = the number of methods of a class c; $NC(m, method, calls)$ = the number of method calls within a method m.

The NC metric has two useful refinements.
The first refinement, $NC_1(n, \eta_1, \eta_2, \varepsilon, \Psi)$, computes the number of η_2-nodes that have (resp. have not) outgoig ε-edges, and that are contained in η_1-node n. Ψ indicates whether adjacency of ε-edges is *required* or *prohibited.* Formally, Ψ is represented by a logic quantifier $\exists$ or $\nexists$.
The second refinement, $NC_2(n, \eta_1, \eta_2, \Pi)$, computes the number of η_2-nodes satisfying property Π and contained in η_1-node n. Π specifies which property needs to be fulfilled for the nodes that are being counted.

Definition 3.3 (NodeCount Refinements)
$NC_1(n, \eta_1, \eta_2, \varepsilon, \Psi) :=$
$\quad NC(n, \eta_1, \eta_2, \Phi)$ where $\Phi(m) = \Psi e \in E : type(e) = \varepsilon \wedge source(e) = m$
$NC_2(n, \eta_1, \eta_2, \Pi) :=$
$\quad NC(n, \eta_1, \eta_2, \Phi)$ where $\Phi(m) = property(m, \Pi)$

Example 3.4 $NC_1(c, class, method, overrides, \exists)$ = the number of methods in a class c that override a method in one of its superclasses;

$NC_2(c, class, method, abstractness(abstract))$ = the number of abstract methods of the class c.

We can also define a generic metric EC for counting edges. $EC(n, \eta, \varepsilon, \theta, \Phi)$ computes the number of ε-edges starting in η-node n. The optional predicate $\Phi : E \to Boolean$ provides a filter when counting the number of edges. θ designates whether multiple occurrences of edges between the same two nodes are counted once or more than once. For example, a method can perform multiple accesses to the same attribute, but sometimes we are only interested in the existence of such an access, while sometimes we need to know the exact number of accesses. In the former case we use $\theta = single$, in the latter case we use $\theta = multiple$.[4]

Definition 3.5 (EdgeCount) Let $n \in N$ and $type(n) = \eta$.
$EC(n, \eta, \varepsilon, multiple, \Phi) := \#FanOut(n)$ where
$FanOut(n) = \{e \in E \mid source(e) = n \wedge type(e) = \varepsilon \wedge \Phi(e)\}$.
$EC(n, \eta, \varepsilon, single, \Phi) := \#\{(n, m) \mid \exists e \in FanOut(n) : source(e) = n \wedge target(e) = m\}$

Example 3.6 $EC(c, class, \overleftarrow{inheritance}, single)$ = the number of immediate children of a class c; $EC(a, attribute, \overleftarrow{accesses}, multiple)$ = the number of times an attribute a is directly accessed by any method in the system. To calculate the number of times a is *locally* accessed by methods belonging to its containing class, we have to specify an extra predicate Φ that captures this constraint:
$EC(a, attribute, \overleftarrow{accesses}, multiple, \Phi)$ where $\Phi(e) =$
$\exists c \in N : type(c) = class \wedge$
$\exists e_1 \in E : source(e_1) = c \wedge target(e_1) = target(e) \wedge type(e_1) = contains$
$\exists e_2 \in E : source(e_2) = c \wedge target(e_2) = a \wedge type(e_2) = contains$

The last generic metric, PL (PathLength), deals with chains of edges of the same type. $PL(n, \eta, \varepsilon, \lambda, \Phi)$ computes the length of a chain of ε-edges starting in η-node n. λ denotes whether we have to calculate the *minimal*, *maximal* or *average* length if there is more than one edge chain originating from the node. $\Phi : E \to Boolean$ is an optional predicate that provides an extra filter over the allowed kind of edges when calculating the path length.

Example 3.7 $PL(c, class, inheritance, maximal)$ = the depth level of the class c within the inheritance hierarchy. This metric computes the length of the longest path in the inheritance tree from the root class to class c. It also works if multiple inheritance is used.

[4] This is the reason why we need multigraphs: in the source code there can be more than one access edge from the same *method*-node to the same *attribute*-node.

3.2 Object-Oriented Software Metrics

We will now show how the generic metrics of the previous section can be used to generate a metrics suite for the object-oriented metamodel specified in Figure 1. We categorize the metrics according to their scope: *system*, *class*, *method* or *attribute*.

First we list some concrete **system metrics** and show how they can be defined in terms of our generic metrics schema.

- #classes in the system $s = NC(s, system, class)$
- #leaf classes in the system $s = NC_1(s, system, class, \overleftarrow{inheritance}, \nexists)$
- #abstract classes in the system $s =$ $NC_2(s, system, class, abstractness(abstract))$

Next we express some typical **class metrics** in terms of the generic metrics.

- #immediate ancestors in class $c = EC(c, class, inheritance, single)$
- #methods in c (a.k.a. NM [15]) $= NC(c, class, method)$
- #immediate descendants in c (a.k.a. Number Of Children [5]) $=$ $EC(c, class, \overleftarrow{inheritance}, single)$
- #public methods in c (a.k.a. PM [15]) $=$ $NC_2(c, class, method, visibility(public))$
- #methods that *override* methods in the superclasses of class $c =$ $NC_1(c, class, method, overrides, \exists)$
- #instance variables in class c (a.k.a. NIV [15]) $=$ $NC(c, class, attribute, scope(instance))$
- #attributes of a class that are being accessed $=$ $NC_1(c, class, attribute, \overleftarrow{accesses}, \exists)$
- depth of class c in the inheritance tree (a.k.a. DIT [5] or HNL [15]) $=$ $PL(c, class, inheritance, maximal)$.

Below we express some **method metrics** in terms of the generic metrics.

- #statements in method m (a.k.a. NOS [15]) $= NC(m, method, statement)$
- #message sends in method m (a.k.a. MSG [15]) $= NC(m, method, calls)$
- #attribute accesses performed by method $m =$ $EC(m, method, accesses, multiple)$
- different number of attributes being accessed by method $m =$ $EC(m, method, accesses, single)$

Finally we express some **attribute metrics** in a generic way.

- #times attribute a is directly accessed $= EC(a, attribute, \overleftarrow{accesses}, multiple)$
- #methods (in the entire software system) accessing attribute $a =$ $EC(a, attribute, \overleftarrow{accesses}, single)$

3.3 Transitive Edges

Sometimes we need to take the *transitive closure* of edges of a certain type into account in order to calculate a particular metric. Therefore, we introduce the + notation to denote transitive edges, i.e., chains of edges of the same type. This straightforward extension of our formalism allows us to define a number of new interesting metrics, such as:

- #ancestors of a class (a.k.a. NOA [5]) = $EC(c, class, inheritance^{+}, single)$
- #number of descendants of a class (a.k.a. NOAC [5]) = $EC(c, class, \overleftarrow{inheritance^{+}}, single)$

4 Higher-Order Metrics

In this section we define some higher order metrics that allow us to capture some more object-oriented metrics in a generic way.

4.1 Ratio Metrics

For the node count metric *NC* we can define a relative counterpart that returns a value between 0 and 1, i.e., a ratio (or a percentage if multiplied by 100). In this way we can generate a whole range of new object-oriented metrics.

Definition 4.1 (Node Count Ratio)
$Ratio(NC(n, \eta_1, \eta_2, \Phi)) := \frac{NC(n,\eta_1,\eta_2,\Phi)}{NC(n,\eta_1,\eta_2)}$

This relative metric has two obvious refinements that correspond to the absolute counterparts NC_1 and NC_2, respectively. $Ratio(NC_1(n, \eta_1, \eta_2, \varepsilon, \Psi))$ computes the ratio of η_2-nodes contained in a η_1-node n for which ε-edges are required/prohibited. $Ratio(NC_2(n, \eta_1, \eta_2, \Pi))$ computes the ratio of η_2-nodes contained in a η_1-node n for which property Π is satisfied.

Definition 4.2 (Ratio Refinements)
$Ratio(NC_1(n, \eta_1, \eta_2, \varepsilon, \Psi)) := \frac{NC_1(n,\eta_1,\eta_2,\varepsilon,\Psi)}{NC(n,\eta_1,\eta_2)}$
$Ratio(NC_2(n, \eta_1, \eta_2, \Pi)) := \frac{NC_2(n,\eta_1,\eta_2,\Pi)}{NC(n,\eta_1,\eta_2)}$

Here are some concrete examples of how to use the ratio metric.

- Ratio of abstract classes in the system s = $Ratio(NC_2(s, system, class, abstractness(abstract)))$
- Ratio of leaf classes in the system s = $Ratio(NC_1(s, system, class, \overleftarrow{inheritance}, \nexists))$
- Method visibility factor (ratio of public methods) of class c = $Ratio(NC_2(c, class, method, visibility(public)))$

- Attribute visibility factor (ratio of public attributes) of class $c =$ $Ratio(NC_2(c, class, attribute, visibility(public)))$

As an alternative, [3] defines the *method hiding factor* as the ratio of non-public methods of a class, and the *attribute hiding factor* as the ratio of non-public attributes of a class.

4.2 Promoted metrics

When there is a containment relationship between entities, one can express so-called *promoted metrics*, i.e., metrics defined by making use of summation or average. $Sum(n, \eta_1, \eta_2, \mu)$ computes the sum of the metric $\mu(m)$ for all η_2-nodes m contained in η_1-node n, while $Average(n, \eta_1, \eta_2, \mu)$ computes the average over this metric.

Definition 4.3 (Promoted Metrics)
$Sum(n, \eta_1, \eta_2, \mu) := \sum_{m \in M} \mu(m)$, where
$M = \{m \in N \mid type(m) = \eta_2 \wedge$
$\exists e \in E : type(e) = contains \wedge source(e) = n \wedge target(e) = m\}$
$Average(n, \eta_1, \eta_2, \mu) := \frac{Sum(n,\eta_1,\eta_2,\mu)}{\#M}$

These promoted metrics allow us to express more object-oriented metrics. For example, we can promote the following method-level and attribute-level metrics to class level:

- Total number of method calls of all the methods of class $c =$ $Sum(c, class, method, NMC)$ where $NMC(m) = NC(m, method, calls)$
- Total number of statements of all the methods of class $c =$ $Sum(c, class, method, NOS)$ where $NOS(m) = NC(m, method, statement)$
- Average method size in class c (a.ka. AMS [15]) $=$ $Average(c, class, method, LOC)$

Similarly we can promote class-level metrics to system level:

- Total number of methods in the system $s =$ $Sum(s, system, class, NOM)$ where $NOM(c) = NC(c, class, method)$
- Average number of methods per class in the system $s =$ $Average(s, system, class, NOM)$
- Total lines of code in the system $s =$ $Sum(s, system, class, CLOC)$ where $CLOC(c) = Sum(c, class, method, LOC)$

5 Discussion and Related Work

This section discusses the current limitations of our approach and presents an overview of related work.

Other metrics. While our framework captures a wide range of object-oriented software metrics, there are still many specialised metrics that cannot be defined in a generic way. Another problem is that, for some metrics, there is still no consensus about what is the best alternative. This is the reason why we did not consider *coupling metrics* (such as CBO [5], CF [3], NCR [15], RFC [5]) and *cohesion metrics* (such as LCOM [5,14], CR [1], CAMC [2]). Some research to define cohesion metrics in a more generic way has already been carried out. The LCOM metric (lack of cohesion in methods) of [14] is defined in terms of graphs in [13] as the number of connected components of a graph. This makes it a generic metric that coincides with the original definition if the graph nodes represent methods, and the graph edges represent attribute accesses. One can also define a generic cohesion measure based on the fact that the similarity between two entities relates to the collection of their shared properties [4]. Let, for any entity x, $p(x)$ be the set of considered properties for a specific situation. For example, if x is a class, $p(x)$ could be the set of all superclasses, the set of all abstract methods, the set of all public attributes, and so on. Then the generic distance metric $dist(x, y)$ supports the measurement of cohesion: parts with low distance are considered cohesive (with respect to the considered properties), while parts with higher distances are less cohesive.

$$dist(x, y) = 1 - \frac{\mid p(x) \cap p(y) \mid}{\mid p(x) \cup p(y) \mid}$$

Subtypes. For reasons of simplicity, we did not provide subtype relationships in our metamodel. However, most metamodelling approaches (such as UML [11] and FAMIX [8]) make use of subtypes. For example, in UML, *class* and *interface* are both subtypes of *classifier*, and *method* and *attribute* are both (indirect) subtypes of *feature*. By exploiting this subtype information we can define another generic metric (assume that η_{sub} is a subtype of η_{super}):

$$SubtypeRatio(id, \eta, \eta_{sub}, \eta_{super}) = \frac{NC(id, \eta, \eta_{sub})}{NC(id, \eta, \eta_{super})}$$

With this generic metric we can calculate the method-to-attribute ratio in a class c as $SubtypeRatio(c, class, method, feature)$, and the class-to-interface ratio in a Java system as $SubtypeRatio(c, system, class, classifier)$.

Edge properties. In this paper we only attached properties to nodes. In many cases, it is also useful to attach properties to edges. For example, we can make a distinction between *accesses*-edges based on whether they *read* or *write* the attribute's value. For a *contains*-edge we can distinguish between *aggregation* and *composition* depending on whether or not the contained element can be shared between different containers. For *inheritance*-edges we can distinguish between *specialisation* and $interface$ inheritance depending on how the parent class is reused in the subclass. Using these extra properties defined on edge types, we can refine the generic metric EC to take edge properties into account, in a similar way as we have refined the generic metric NC to NC_2.

Generic metamodel. [17] uses a generic metamodel 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 [11] or FAMIX [8]) that are mapped to the generic metamodel.

6 Tool Support

Clearly, a metrics tool should not be used as a stand-alone tool, but should be integrated in a (meta-) CASE tool or be part of the development environment. To illustrate this, we have integrated two research prototypes of our metrics framework in existing software engineering tools.

Our first prototype was integrated in Moose [8], a reverse engineering framework, implemented in VisualWorks Smalltalk that comes with a language-independent metamodel FAMIX. It supports import from C++, Java, Smalltalk and Ada source code. We implemented more than 30 language-independent metrics based on the metamodel information only. We integrated these metrics into CodeCrawler [6], a powerful software visualisation tool defined on top of Moose. An obvious benefit we experienced was the fact that we were able to perform metric measurements on the case studies independently from their size, implementation language and general condition (reengineering case studies find themselves seldom in a coherent and ordered manner).

A second prototype of our generic metrics approach was specified in SOUL [20,16], a logic metaprogramming language built on top of – and tightly integrated with – the VisualWorks Smalltalk programming environment. SOUL offers a wide scale of logic rules to reason about the underlying object-oriented base language. We extended SOUL to express the generic metrics as a collection of logic rules. This could be done in a language-independent way, since SOUL provides support for Smalltalk as well as Java through the SOULJava extension [10]. The generic metrics are also applicable to language-specific features, such as Java interfaces.

7 Conclusion

We use type graphs as a metamodel to define a set of generic metrics that allow us to express a large number of object-oriented metrics in a general, flexible and extensible way. The metrics that can be expressed using our framework have definitions that are *unambiguous*, *simple* and *language independent*. They are unambiguous, as their definition relies on the accurate formalism of graphs. They are simple, as their computation is mainly achieved using graph traversal techniques.

Furthermore, the framework allows us to automate measurement tools to a large extent by relying on the metamodel information to generate language-independent metrics suites. The main advantage is that we do not have to rewrite a tool each time a new metric (or a variant of an existing one) is introduced, or when new language features are included. The approach also allows us to experiment to find out which metrics are most appropriate in a particular situation.

Acknowledgement

We thank Kim Mens, Roel Wuyts and the anonymous reviewers for their comments on drafts of this paper.

References

[1] Balasubramanian, N., *Object-oriented metrics*, in: *Proc. Int. Asia-Pacific Conf. Software Engineering* (1996), pp. 30–34.

[2] Bansiya, J., L. Etzkorn, C. Davis and W. Li, *A class cohesion metric for object-oriented designs*, J. Object-Oriented Programming **11** (1999), pp. 47–52.

[3] Brito e Abreu, F., M. Goulao and R. Esteves, *Toward the design quality evaluation of object-oriented software systems*, in: *Proc. Int. Conf. Software Quality*, 1995, pp. 44–57.

[4] Bunge, M., "Treatise on Basic Philosophy. Ontology I: The Furniture of the World." Boston, Riedel, 1977.

[5] Chidamber, S. R. and C. F. Kemerer, *A metrics suite for object-oriented design*, IEEE Trans. Software Engineering **20** (1994), pp. 476–493.

[6] Demeyer, S., S. Ducasse and M. Lanza, *A hybrid reverse engineering approach combining metrics and program visualization*, in: *Proc. Working Conf. Reverse Engineering (WCRE '99)* (1999).

[7] Demeyer, S., S. Ducasse and O. Nierstrasz, *Finding refactorings via change metrics*, in: *Proc. Int. Conf. OOPSLA 2000* (2000).

[8] Ducasse, S., M. Lanza and S. Tichelaar, *Moose: an extensible language-independent environment for reengineering object-oriented systems*, in: *Proc. 2nd Int'l Symp. Constructing Software Engineering Tools (CoSET 2000)*, 2000.

[9] Ducasse, S., M. Rieger and S. Demeyer, *A language independent approach for detecting duplicated code*, in: H. Yang and L. White, editors, *Proc. Int'l Conf. Software Maintenance*, 1999, pp. 109–118.

[10] Fabry, J. and T. Mens, *Language-independent detection of oo patterns using logic meta programming*, Technical report, Programming Technology Lab, Vrije Universiteit Brussel (2002), submitted to PADL 2003.

[11] Group, O. M., *OMG Unified Modeling Language specification version 1.3*, formal/2000-03-01 (2000).

[12] Henderson-Sellers, B., "Object-Oriented Metrics: Measures of Complexity," Prentice-Hall, 1996.

[13] Hitz, M. and B. Montazeri, *Measuring coupling and cohesion in object-oriented systems*, in: *Proc. Int. Symp. Applied Corporate Computing*, 1995, pp. 25–27.

[14] Li, W. and S. Henry, *Object-oriented metrics that predict maintainability*, J. Systems and Software **23** (1993), pp. 111–122.

[15] Lorenz, M. and J. Kidd, "Object-Oriented Software Metrics: A Practical Approach," Prentice-Hall, 1994.

[16] Mens, K., I. Michiels and R. Wuyts, *Supporting software development through declaratively codified programming patterns*, Journal on Expert Systems with Applications (2002).

[17] Mišić, V. B. and S. Moser, *From formal metamodels to metrics: An object-oriented approach*, in: *Proc. Technology of Object-Oriented Languages and Systems (TOOLS-24)* (1998).

[18] Simon, F., F. Steinbrückner and C. Lewerentz, *Metrics based refactoring*, in: *Proc. European Conf. Software Maintenance and Reengineering* (2001), pp. 30–38.

[19] Tichelaar, S., S. Ducasse, S. Demeyer and O. Nierstrasz, *A meta-model for language-independent refactoring*, in: *Proc. Int. Symp. Principles of Software Evolution* (2000), pp. 157–169.

[20] Wuyts, R., *Declarative reasoning about the structure of object-oriented systems*, in: *Proc. Int'l Conf. TOOLS USA'98* (1998), pp. 112–124.