A Formal Approach to Building a Polymorphism Metric in Object-Oriented Systems

Claudia Pons* Luis Olsina♦ Máximo Prieto*

* Lifia, Universidad Nacional de La Plata
Calle 50 esq.115, 1er.Piso,
(1900) La Plata, Argentina
E-mail: cpons@sol.info.unlp.edu.ar

♦GIDIS, Facultad de Ingeniería,
UNLPam, Calle 9 y 110
(6360) Gral.Pico, LP, Argentina
E-mail: olsinal@ing.unlpam.edu.ar

Key words: Object-oriented software quality metrics, formal methods, polymorphism, dynamic logic.

Abstract

Although quality is not easy to evaluate since it is a complex concept compound by different aspects, several properties that make a good object-oriented design have been recognized and widely accepted by the software engineering community. We agree that both the traditional and the new object-oriented properties should be analyzed in assessing the quality of object-oriented design. However, we believe that it is necessary to pay special attention to the *polymorphism* concept and metric, since they should be considered one of the key concerns in determining the quality of an object-oriented system.

In this paper, we have given a rigorous definition of polymorphism. On top of this formalization we propose a metric that provides an objective and precise mechanism to detect and quantify dynamic polymorphism. The metric takes information coming from the first stages of the development process giving developers the opportunity to early evaluate and improve the quality of the software product. Finally, a first approach to the theoretical validation of the metric is presented.

1. Introduction

Object-oriented (O-O) software engineers need a better understanding of the desirable and non-desirable properties of O-O systems design, and their effect onto the quality factor. The desirable properties must represent those characteristics that ultimately lead to more efficient, reusable, maintainable and extensible software products.

The key issues related to the quality assessment of O-O systems are:

- It is necessary to determine the desirable and non-desirable properties of systems.
- There must be a formal definition of these properties.
- It is necessary to provide mechanisms to detect (and quantify) the presence of these properties. These mechanisms must be formal and objective.

Although quality is not easy to evaluate since it is a complex concept integrated by different aspects, several properties that make a good O-O design have been recognized and widely accepted by the community. Despite the fact that the properties of coupling, cohesion, modularity, complexity and size, generally used to characterize quality in traditional structural design are also important in O-O designs, traditional metrics are not suitable for O-O designs. This problem is due to the presence of additional properties that are inherent to the O-O paradigm, such as abstraction, inheritance and polymorphism. These new concepts are vital for the construction of reusable, flexible and adaptable software products, and they must be taken into consideration by O-O software quality metrics.

The applicability problem of traditional techniques has been analyzed in the works of Chidamber and Kemerer (1994), Tegarden et al. (1992) and Wilde and Huitt (1992) among others. Special metrics for O-O systems have been investigated, see for example the works of Chen and Lu (1993), Kim et al. (1994) and

Li and Henry (1993). There are numerous proposals addressing the assessment of the traditional properties into O-O systems; for example the works of Poulin (1997), Briand et al. (1997), Price and Demurjian (1997), Benlarbi (1997). But less work has been done in the field of the specific O-O properties; see for example the works of Moore (1996), Bansiya (1997; 1999-a; 1999-b), Benlarbi and Melo (1999), Abreu and Carapuça (1994) and Zuse (1998).

An additional problem is that many of the currently available metrics can be applied only when the product is finished or almost finished, since data is taken from the implementation. This makes the quality-weakness problems be detected too late. It is desirable to have a tool that takes information coming from the first stages of the development process (i.e. requirement analysis phases); this will give developers the opportunity to early evaluate and improve the quality of the product in the development process.

In this paper, new metrics to measure the quality of an O-O design are defined. These metrics are applied to the conceptual model of a system expressed in the Unified Modeling Language UML (1999), thus permitting an early analysis of the system quality. Although we agree that both the traditional and the specific O-O properties or attributes should be analyzed in assessing the quality of O-O design, our purposes are not to define a complete quality evaluation mechanism (in the sense that it considers every system property), but only to characterize some aspects of the *polymorphism* attribute.

Polymorphism concept can be considered one of the key concerns in order to determine the quality of an O-O design. Regarding the literature, e.g., Benlarbi et al. (1999), different kinds of polymorphisms have been classified, namely: pure, static, and dynamic ones. For instance, considering the latter, for two methods to be polymorphic, they need to have the same name and signature (parameter types and return type) and also the same effects (changing the state of the receiver in the same way and raising the same messages to other objects in the system). Dynamic binding lets one substitute objects that are polymorphic for each other at run-time. This substitutability is a key concept in O-O systems. Polymorphic systems have several advantages. They simplify the definition of clients, since as long as a client only uses the polymorphic interface, it can substitute an instance of one class for another instance of a class that has the same interface at run-time. Because all instances behave the same way.

We formally define the polymorphism concept, giving foundations for its detection and quantification. Thus, the polymorphism measure should be combined with the measures of the rest of the properties (such as coupling, cohesion, entropy, etc.) with the aim of determining the total quality of the system. However, this metrics combination task is beyond the scope of this work.

2. The formal domain

We introduce the M&D-theory, a proposal for giving formal semantics to the Unified Modeling Language UML (1999). The basic idea behind this formalization is the definition of a semantic domain integrating both the model level and the data level. In this way, both static aspects and dynamic aspects of either the model or the modeled system, can be described within a first order formal framework.

Dichotomy of entities

The entities defined by the M&D-theory are classified in two disjoint sets: Modeling entities and Modeled entities. Figure 1, shows this dichotomy of entities. Modeling entities correspond to concrete syntax of the UML, such as Classes or StateMachine. In contrast, modeled entities, such as Object or Link represent run-time information, i.e. instances of classes and processes running on a concrete system.

Structure of the theory

The M&D-theory is a first-order order-sorted dynamic logic theory[1], consisting of three sub-theories:

M&D-theory = UML-theory + SYS-theory + JOINT-theory

[1] A first-order order-sorted dynamic logic therory **Th** consists of a signature Σ that defines the language of the theory, and a set of Σ-axiomas ϕ: **Th** = (Σ, ϕ)

A signature Σ consists of a a set of sort symbols S, a partial order relation between sorts $\leq$, a set F of function symbols, a set P of predicate symbols , and a set A of Action symbols : $\Sigma = ((S, \leq), F, P, A)$

The language of the theory intentionally follows the notation of the UML metamodel (UML, 1999) and the Object Constraint Language OCL (UML, 1999).

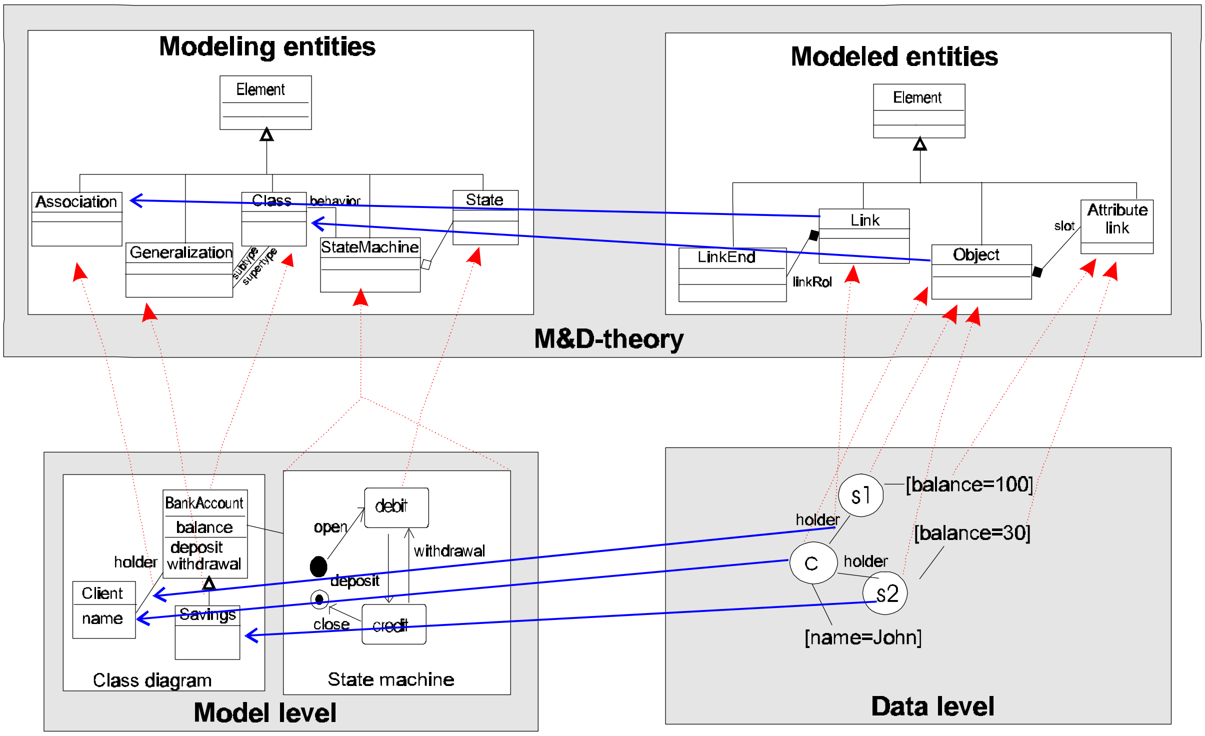

Figure 1: dichotomy of entities in the M&D-theory

The sub theory UML-theory:

The theory describes modeling entities (i.e. models). In the UML, Class Diagrams model the structural aspects of the system. Classes and relationships between them, such as Generalizations, Aggregations and Associations constitute Class Diagrams. On the other hand, the dynamic part of the system is modeled by Sequence and Collaboration diagrams that describe the behavior of a group of instances in terms of message sendings, and by State Machines that show the intra-object dynamics in terms of state transitions.

Modeling entities are related to other modeling entities*.* Consider for example the association between Class and StateMachine by the relation labeled ‘behavior’. This association indicates that StateMachines can be used for the definition of the behavior of the instances of a Class. Other example is given by the relation existing between StateMachine and State, that specifies that a StateMachine is composed by a set of States. It is important to formally define how the different UML diagrams are related to one another, to be able to maintain the consistency of the model. Moreover, it is important to specify the effect of modifications of these diagrams, showing what is the impact on other diagrams, if a modification to one diagram is made.

The theory consists of a signature $\Sigma_{UML} = ((S_{UML}, \leq), F_{UML}, P_{UML}, A_{UML})$ and a formula ϕ_{UML} over Σ_{UML}:

$$\text{UML-theory} = (\Sigma_{UML}, \phi_{UML})$$

The set S_{UML} contains sort symbols representing modeling elements, such as Class and StateMachine. The order relation between sorts allows for the hierarchical specification of the elements.

The sets of symbols F_{UML} and P_{UML} define functions and predicates on modeling entities.

The set A_{UML} consists of action symbols representing evolution of specifications over their life cycle. One of the most common forms of evolution involves structural changes such as the extension of an existing specification by addition of new classes of objects or the addition of attributes to the original classes of objects. At the other hand, evolution at this level might reflect not only structural changes but also behavioral changes of the specified objects. Behavioral changes are reflected for example in the modification of sequence diagrams and state machines.

The formula ϕ_{UML} is the conjunction of two disjoint sets of formulas, ϕ_S and ϕ_D of static and dynamic formulas respectively. The former consists of first-order formulas which have to be valid in every state the system goes through (they are invariants or static properties or well-formedness rules of models). These rules are used to perform schema analysis and to report possible schema design errors. The latter consists of modal formulas defining the semantics of actions, that is to say, the evolution of models.

***The sub-theory SYS-theory*:**

This theory describes the modeled entities (i.e. data and process). The elements in the data level are basically instances (data value and objects) and messages. At the data level a system is viewed as a set of related objects collaborating concurrently. Objects communicate each other through messages that are stored in semi-public places called mailboxes. Each object has a mailbox where other objects can leave messages.

Modeled entities are related to other modeled entities. For example the relationship named 'slot' between Object and AttributeLink, denotes the connection between an Object and the values of its attributes.

The theory consists of a signature $\Sigma_{SYS} = ((S_{SYS}, \leq), F_{SYS}, P_{SYS}, A_{SYS})$ and a formula γ_{SYS} over Σ_{SYS}:

$$\text{SYS-theory} = (\Sigma_{SYS}, \gamma_{SYS})$$

The set S_{SYS} contains sort symbols representing the data in the system and its relationships, such as objects, links, messages, etc. The sets of symbols F_{SYS} and P_{SYS} define functions and predicates on data.

The set A_{SYS} consists of action symbols representing evolution of data at run time, such as object state changes. The formula γ_{SYS} is the conjunction of two disjoint sets of formulas, γ_S and γ_D of static and dynamic formulas respectively. The former consists of first-order formulas which have to be valid in every state the system goes through (they are invariants or static properties or well-formedness rules of data). Whereas, the latter consists of modal formulas defining the semantics of actions, that is to say the possible evolution of the data.

The sub-theory JOINT-theory:

This part of the theory describes the connection between model and data levels. Modeling entities are related to modeled entities. There is a special relationship among some modeled entities with their corresponding modeling entity. This relationship denotes 'instantiation', for example, an Object is an instance of a Class, whereas Links are instances of Associations (see figure 1).

Finally, ϕ_{JOINT} is a formula constructed in the extended language $\Sigma_{M\&D}$, and thus it can express at the same time data properties (e.g. behavioral properties of objects), model properties (e.g. properties about the specification of the system) and properties relating both aspects.

Figure 2, shows a sample of the M&D-theory. More details of the theory can be found in (Pons et al., 1999 and Pons, 1999).

Advantages of the integration

The integration of modeling entities and modeled entities into a single formalism allows us to express both static and dynamic aspects of either the model or the modeled system within a first order framework. The validity problem (i.e. given a sentence ϕ of the logic, to decide whether ϕ is valid) is less complex for first-order formalisms than for higher order formalisms. Although first order logic is undecidable, computer systems satisfy certain properties (e.g. systems are interpreted over arithmetic structures, the state of a program is given by a finite set of values) that allow us to calculate the validity of formulas in an effective way. The four different dimensions: static aspects of models, static aspects of data, dynamic aspects of models, dynamic aspects of data, are highlighted in figure 2.

The integrated formalism is suitable for the definition of a variety of properties of O-O systems, structural properties as well as behavioral properties.

The logic allows us to define structural properties such as: depth of class hierarchy, size of class interface, number and type of associations between classes, etc. These properties can be expressed because classes, associations, generalization, etc. are first-class citizens in the logic. On the other hand, instances and their behavior are also first-class citizens of the logic, as a consequence, it is possible to define behavioral properties such as pre/post conditions of operations, equivalence of behavior, among others.

Specification of Classifier

Sorts Classifier

Taxonomy Classifier≤GeneralizableElement

Updatable functions

features: Classifier→Seq of Feature

associationEnds: Classifier→Set of AssociationEnd

behavior: Classifier →StateMachine

....

Updatable predicates

PartOf: Classifier x Classifier

Actions

addFeature: Classifier x Feature → ModelEvolution

deleteFeature: Classifier x Feature → ModelEvolution

Axioms $\forall$c:Classifier $\forall$f:Feature $\forall$e:AssociationEnd

Static axioms

[1]No Attributes may have the same name within a Classifier

$\forall f,g \in attributes(c)\ (name(f) = name(g) \rightarrow f=g\ )$

[2] symmetry rules

$f \in features(c) \leftrightarrow owner(f)=c\ \wedge\ e \in associationEnds(c) \leftrightarrow type(e)=c$

....

Dynamic axioms

$\langle addFeature(c,f)\rangle \mathit{true} \rightarrow f \notin allFeatures(c)$

$[addFeature(c,f)]\ (Exists(f) \wedge f \in features(c) \wedge owner(f)=c)$

....

End specification of Classiffier

Static aspects of models

Dynamic aspects of models

Specification of Instance

Sorts Instance

Taxonomy Instance≤DataElement

Updatable functions

slots: Instance → Set of AttributeLink

linkEnds: Instance → Set of LinkEnd

classifier: Instance → Classifier

mailBox: Instance → Seq of Message

currentStates: Instance →State

....

Actions

-.-: Instance, Message → Modification

Axioms $\forall$i:Instance $\forall$l:LinkEnd

Static axioms

[1] the AttributeLinks matches the declarations in the Classifier.

$\forall l \in slots(i)\ (attribute(l) \in allAttributes(classifier(i))\ )$

[2] symmetry

$l \in linkEnds(i) \leftrightarrow instance(l)=i$

....

Dynamic axioms

[1] effect of call actions:

$previousStates=currentStates(o)\ \wedge$

$firing=firingTransitions(behavior(classifier(o)),m) \rightarrow$

$[o.m]\ (currentStates(o)=(previousStates - \{source(t) \mid t \in firing\}\ )$

$\cup\ \{target(t) \mid t \in firing\}$

$\wedge\ \forall\ t \in firing\ \forall a \in effect(t)\ sent(a)$

$\wedge\ m \notin\ mailBox(o)\ \ )$

....

End specification of Instance

Static aspects of data

Dynamic aspects of data

Figure 2: sample of the M&D-theory

3. Using the M&D-theory to formalize an O-O model

We formally define the semantics of the UML using a two-step approach:

1- interpretation (or translation) of the UML to the M&D-theory.

2- semantics interpretation of the M&D-theory.

That is to say, **UML-constructions** $\rightarrow^{\textit{translation}}$ **M&D-theory** $\rightarrow^{\textit{semantics}}$ **Semantics-domain**

The semantics mapping **Sem** is the composition of both functions, **Sem**=***semantics*** o ***translation***

The first step converts an UML model instance to the modal logic theory, the conversion provides a set of formulas that serves as an intermediate description for the meaning of the UML model instance. The key components of this step are rules for mapping the graphic notation onto the formal kernel model.

The second step is the formal interpretation of this set of formulas. The semantics domain where dynamic logic formulas are interpreted is the set of transition systems. A transition system, $U=(S^U,w_o,m_U)$, is a set of possible worlds or states with a set of transition relations on worlds; For details about semantics of dynamic logic, see Harel et al (1999) or Wieringa and Broersen (1998). Formally, let Σ=((S, $\leq$), F, P, A) be a first-order dynamic logic signature and let Σ_N=((S, $\leq$), F_N, P_N) be the non-updatable part of Σ. Let $U=(A,m_U)$ be a Σ_N-algebra, providing a domain for the interpretation of static terms. Formulas of the language are interpreted on Kripke-frames as follows: $U=(S^U,w_o,m_U)$. Where:

- ✓ S^U is the set of states. Each state $w\in S^U$, is a function that maps terms to the algebra U, in the following way:
  - if $f\in F_N$ then $w(f)=f^U$ (i.e, the static interpretation given by U).
  - if $f\in F_U$ y $f:s_1,..,s_n\rightarrow s$ then $w(f): U_{s1},..,U_{sn} \rightarrow U_s$.
  - if $p\in P_N$ then $w(p)=p^U$ (i.e, the static interpretation given by U).
  - if $p\in P_U$ y $p:s_1,..,s_n$ then $w(p): U_{s1},..,U_{sn}$
  - if x is a variable of sort s, then $w(x)\in U_s$.
  - if $\alpha\in A$ then $w(\alpha)=\alpha^U$ (i.e, the static interpretation given by U).
- ✓ $w_0\in S^U$ is the initial state.
- ✓ m_U associates each action α to a binary relation called the input/output relation of α: $m_U(\alpha) \subseteq S^U \times S^U$

The domain for states is an heterogeneous algebra (a Σ_N-algebra) whose elements are both model elements (such as classes) and data elements (such as objects).

Interpretation of formulas

The interpretation of a term t in a state w given v (written as $int_w(t)$) is defined in the usual way.

The satisfaction of a closed formula in a structure U and a state w is defined as follows:

$U,w \models (t1=t2)$ iff $int_w(t1)= int_w(t2)$

$U,w \models \neg\phi$ iff *not*$(U,w\models\phi)$

$U,w \models \phi\wedge\gamma$ iff $U,w \models\phi$ and $U,w \models\gamma$

$U,w \models [a]\phi$ iff $\forall w'$, if $(w,w')\in m_U(\alpha)$, then $U,w'\models\phi$.

A model for a specification sp=(S,F,P,A, ϕ) is a structure U such that $U,w_0\models\phi$.

4. Formal Definition of Polymorphism

In this section, we give a rigorous definition of polymorphism in the framework of the M&D-theory. Main definitions of the polymorphism concept can be read in (Woolf, 1997). Let M be an UML model of an O-O system. Let U be the formal semantics of that model, i.e. U =***Sem***(M).

Definition 1: Polymorphic Methods

For two methods to be polymorphic, they need to have the same name and signature (parameter types and return type) and also the same effects (changing the state of the receiver in the same way and raising the same messages to other objects in the system).
Let m be a method name. Let C_1 and C_2 be two classes existing in the model M.
The methods named m are polymorphic in C_1 and C_2 in the model U if the following formula holds:

$$U \models \mathrm{Polymorphic}(m,C_1,C_2)$$

Where the predicate Polymorphic is defined as follows:

Def 1.1:
$\forall m{:}\mathrm{Name}\ \forall C_1,C_2{:}\mathrm{Class} \bullet$
$\mathrm{Polymorphic}(m,C_1,C_2) \leftrightarrow \exists m_1,m_2 \bullet (\ m_1 \in C_1.\mathrm{operations} \wedge m_2 \in C_2.\mathrm{operations} \wedge$
$m_1.\mathrm{name}=m \wedge m_2.\mathrm{name}=m \wedge m_1.\mathrm{visibility}=m_2.\ \mathrm{visibility}$
$\wedge\ \mathrm{hasSameSignature}(m_1,m_2)$
$\wedge\ \mathrm{hasSameBehavior}(m,C_1,C_2)\ )$

The predicate hasSameSignature applied on two methods is true if both methods have the same signature. It is defined in the M&D-theory as follows:

Def 1.2:
$\forall b,b'{:}\mathrm{BehavioralFeatures} \bullet$
$\mathrm{hasSameSignature}(b,b') \leftrightarrow (b.\mathrm{name}=\ b'.\mathrm{name} \wedge \mathrm{areEquivalent}(b.\mathrm{parameters},\ b'.\mathrm{parameters}\ )\ )$

Where areEquivalent is defined as follows:

Def 1.3:
$\mathrm{areEquivalent}(\lambda,\lambda)=\mathrm{true}$
$\mathrm{areEquivalent}(p{:}ps,\ \lambda)=\mathrm{false}$
$\mathrm{areEquivalent}(\lambda,p{:}ps)=\mathrm{false}$
$\mathrm{areEquivalent}(p_1{:}ps,p_2{:}ps')=\mathrm{equivalent}(p_1,p_2) \wedge \mathrm{areEquivalent}(ps,ps')$

Where λ denotes the empty sequence and p:ps denotes a non-empty sequence made up from a head (denoted by p) and a tail (denoted by ps).

And finally, the predicate equivalent is applied on two single parameters determining their equivalence:

Def 1.4:
$\forall p_1,p_2{:}\mathrm{Parameter} \bullet \mathrm{equivalent}(p_1,p_2) \leftrightarrow\ (p_1.\mathrm{defaultValue}=p_2.\mathrm{defaultValue} \wedge$
$p_1.\mathrm{kind}=\ p_2.\mathrm{kind} \wedge p_1.\mathrm{type}=p_2.\mathrm{type}\ )$

Two methods named m have the same behavior in C_1 and C_2 if they are indistinguishable, i.e., for every two objects o_1 and o_2 (being o_1 instance of C_1 and o_2 instance of C_2), the effect of executing *o*.m is the same as the effect of executing *o*.m, where o.m denotes that object o receives and execute the method named m. The predicate is defined as follows:

Def 1.5:

$U,w \models \mathrm{hasSameBehavior}(m,C_1,C_2)$ iff

$\forall \boldsymbol{o}{:}\mathrm{Object} \bullet \boldsymbol{o}.\mathrm{classifier}=C_1$ then

$\forall w_1 \bullet (w,w_1) \in m_U(\boldsymbol{o}.m)$ then

$\exists\ w',w'_1 \bullet (\ (w,w') \in m_U(\boldsymbol{o}.\mathrm{migrates}(C_2))$

$\wedge\ (w',w'_1) \in m_U(\boldsymbol{o}.m)$
$\wedge\ (w'_1,w_1) \in m_U(\boldsymbol{o}.\ \mathrm{migrates}(C_1))\ )$

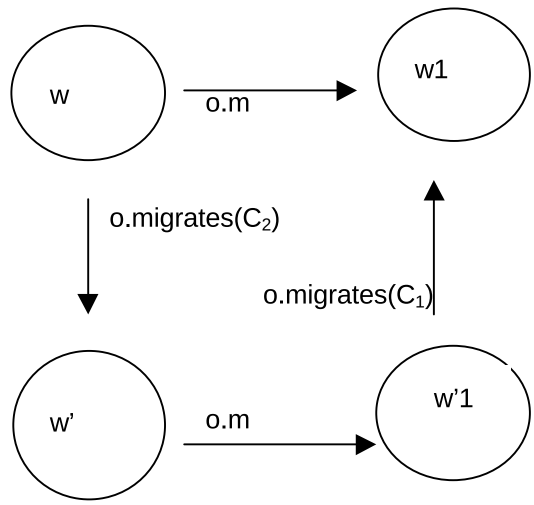

Figure 3: commutativity of polymorphic methods

That is to say, the diagram in figure 3 commutes, where the action ***o***.migrates(C) represents that object ***o*** switches its class to C (see definition 1.6).

Def 1.6: [***o***.migrates(C)] ***o***.classifier=C

Corollary: for every formula ϕ , the following schema is valid in the class of models satisfying that the method named m is polymorphic in classes C_1 and C_2:

$\forall o \in instances(C_1)$ $[o.m]\phi \leftrightarrow [o.migrates(C_2)]$ $[o.m]$ $[o.migrates(C_1)]$ ϕ

Definition 2: Polymorphic Classes

Two classes are polymorphic if they define the same methods, and these methods are polymorphic. Two objects belonging to polymorphic classes are polymorphic objects. Dynamic binding lets you substitute objects that are polymorphic for each other at run-time. This substitutability is a key concept in O-O systems.

Formally, two classes C_1 and C_2 are polymorphic in model U, if the following condition holds:

$U \models Polymorphic(C_1,C_2)$

Where the predicate Polymorphic is defined in the following way:

Def 2.1:

$\forall C_1,C_2$: Class $\bullet$ Polymorphic$(C_1,C_2) \leftrightarrow$
interface(C_1)=interface$(C_2) \wedge \forall n \in$ interface$(C_1) \bullet$ polymorphic(n,C_1,C_2)

Function interface returns the set of names of public methods of a class. It is defined in the following way:

Def 2.2: $\forall$C:Class $\bullet$ $\forall$a:Name $\bullet$ (n$\in$interface(C) $\leftrightarrow$ $\exists$f$\in$C.allFeatures $\bullet$ (f.name=n $\wedge$ f.visibility=#public))

Definition 3: Polymorphic Hierarchy

The concept of polymorphic class that was previously defined is too strong, because in general only some of the methods of a class are polymorphic, not all of them. Therefore, a more flexible concept of polymorphism has been defined (see Woolf, 1997), named *core interface*. A *core interface* is a set of polymorphic methods that several classes share. For a hierarchy to be polymorphic all its classes must share a core interface.

Polymorphic hierarchies have several advantages. They simplifies the definition of clients, since as long as a client only uses the core interface, it can substitute an instance of one class for other instance of a class that has the same core interface at run-time. Because all of instances behave the same, one works just as well as another, with regard to the core interface.

Formally, let H be a set of classes in an O-O hierarchy. Hierarchy H is polymorphic in the model U if there exists a *core class* (this core class might not belong to the set H). The formula below defines the concept of polymorphic hierarchy:

$U \models Polymorphic(H)$

Where the predicate Polymorphic is defined in the following way:

Def 3.1:

$\forall$H:Set of Class $\bullet$ (Polymorphic(H) $\leftrightarrow$ $\exists$C:Class $\bullet$ isCore(C,H))

Where:

Def 3.2:

$\forall$ H:Set of Class $\bullet$ $\forall$C:Class $\bullet$ (isCore(C,H) $\leftrightarrow$
(interface(C)$\neq\emptyset$ $\wedge$ $\forall$S$\in$H $\bullet$ (interface(C)$\subseteq$interface(S) $\wedge$ $\forall$n$\in$interface(C) $\bullet$ polymorphic(n,C,S))))

The degree of polymorphism depends on the size of the core interface. The larger the core is, the higher the polymorphism degree is.

5. The Polymorphism metric

In the previous sections, we have given a rigorous definition of polymorphism in the framework of the M&D-theory (Pons et al. 99) . On top of this formalization we propose a metric for measuring polymorphism, that provides an objective and precise mechanism to detect and quantify polymorphism.

We define the following functions on O-O UML models.

Let S be a UML Model, that is to say, a Package containing a hierarchy of modelElements that together describe an abstraction of a physical system (UML, 1999).

- hierarchies(S) returns a collections containing every non-trivial, pair-wise disjoint hierarchy defined in S.

Then we define the following functions on a class hierarchy h (we restrict h to be a tree):

- classes(h) returns a set containing all of the classes belonging to hierarchy h.
- methods(h) returns a bag containing all of the methods defined in the hierarchy, as follows:

  methods(h)= $\bigoplus_{c\in \text{classes(h)}}$ interface(c), where $\bigoplus$ represents bags union(with repetitions).

- core(h) returns the largest core in the hierarchy h.
- width(h) returns the width of the hierarchy, it is defined as follows:

  width(h) = #(interface(core(h))) , where the symbol # denotes set cardinality.

- children(h) returns the set of direct sub-hierarchies of hierarchy h.

Let S be an UML model, the polymorphism metric of S is the average polymorphism measure of every pairwise disjoint hierarchies in S. The polymorphism metric function is defined as follows:

***polymorphism_metric*:** System $\rightarrow$ [0..1]

$$\textbf{\textit{polymorphism\_metric}}(S) = \frac{\sum_{h\in \textbf{hierarchies(S)}} \text{polymorphism\_measure}(h)}{\#\text{hierarchies}(S)}$$

polymorphism_measure: Hierarchy $\rightarrow$ [0..1]

<u>Trivial Case</u>:

if #classes(h)=1 *then* polymorphism_measure(h)=0

<u>General Case</u>:

if #classes(h)>1 *then* polymorphism_measure(h)=polymorphic_methods(h) / #methods(h)

Where:

$$\text{polymorphic-methods}(h) = \text{width}(h) * \#\text{classes}(h) + \sum_{hi\in \text{children}(h)} \text{polymorphic-methods}(hi - \text{core}(h))$$

Where (hi – core(h)) stands for the hierarchy resulting after removing from hi all of the methods belonging to core(h). Let us remark that the result of the function ***polymorphism_metric*** is in the interval [0...1], where the number zero represents the absence of polymorphism, while the number 1 represents the highest degree of polymorphism (i.e., all of the methods in the hierarchy are polymorphic). In the case of trivial hierarchies (i.e., hierarchies made up from a single class), the metrics returns zero.

6. Examples

6.1 Identifying polymorphism

The hierarchy of collections has been defined and implemented in numerous O-O languages. Figure 4, shows a part of the Collection hierarchy of the Smalltalk language (Lalonde 1994). Using the formal definitions of the M&D-theory we can identify the presence of polymorphic methods in this hierarchy, for example: *size, includes:, select:, collect: reject:, remove:, add*:,etc. As an example, we show the formal proof of that the method named *remove:* is polymorphic for classes Set and Bag. The statement aCollection.remove:anElement denotes that the element equal to anElement is removed from the receiver collection. In the case the receiver is a Set, at most one occurrence can belong to the set, but in the case the receiver is a Bag, several occurrences of the same object may belong to it. In the same way, it is possible to proof that method add: is not polymorphic for classes Set and Bag (i.e. $\models \neg$Polymorphic(*add:*, Set, Bag)) due to the detection of duplicated elements.

Classes	Public Instance Methods
Collection	size, includes:, select:, collect:, reject:, detect:
NotIndexedCollection	remove:
Set	add:
Bag	add:
IndexedCollection	last, first, at:, at:put:
FixedSizeCollection	
Array	
String	< , >
VariableSizeCollection	remove:
OrderedCollection	add:
SortedCollection	add:

Figure 4: part of the Collection hierarchy in Smalltalk[2]

<u>*Theorem 1*</u>: *remove* is polymorphic for classes Set and Bag: $\models$ Polymorphic(*remove*, Set,Bag)

<u>*Hypothesis*</u>:

Let Set and Bag the classes in the Smalltalk hierarchy. There exists an operations removeInSet belonging to Set.interface, and there exists an operation removeInBag belonging to Bag.interface, such that:

[h0] removeInSet$\in$Set.interface $\wedge$ removeInBag$\in$Bag.interface
[h1] removeInSet.name=*remove* $\wedge$ removeInSet.visibility=public
[h2] removeInSet.parameters= <p1>
[h3] p1.defaultValue=nullElement
[h4] p1.kind=in
[h5] p1.type=Object
The specification of the operation is given by the following dynamic logic formula:
[h6] $\forall$s,e:Object $\bullet$ (s.classifier=Set $\rightarrow$ [s.*remove*(e)]e$\notin$s)
[h7] removeInBag.name=*remove* $\wedge$ removeInBag.visibility=public
[h8] removeInBag.parameters= <p2>
[h9] p2.defaultValue= nullElement
[h10] p2.kind = in
[h11] p2.type=Object
The specification of the operation is given by the following dynamic logic formula:
[h12] $\forall$b,e:Object (b.classifier=Bag $\wedge$ occurrences(b,e)=n $\rightarrow$ [b.*remove*(e)]occurrences(b,e)=n-1)

Lets first prove the following lemmas:

***<u>Lemma 1</u>*:** both operations have the same signature: $\models$ hasSameSignature(removeInSet,removeInBag)

<u>Proof of lemma 1</u>:

[l1] p1.defaultValue=p2.defaultValue (from [h3] and [h9])
[l2] p1.kind=p2.kind (from [h4] and [h10])
[l3] p1.type=p2.type (from [h5] and [h11])
[l4] equivalent (p1,p2) (applying def 1.4 to [l1], [l2] and [l3])
[l5] AreEquivalent(<p1>,<p2>) (applying def 1.3 to [l4],
[l6] AreEquivalent(removeInSet.parameters, removeInBag.parameters) (from [h2] and [h8])
[l7] HasSameSignature(removeInSet, removeInBag) (applying def 1.2 to [h1], [h7] and [l6])

***<u>Lemma 2</u>*:** the method *remove*: has the same behavior in both classes:

---

[2] In the chart, every polymorphic method appears only once in the hierarchy of classes. For example, every class of the hierarchy has a polymorphic method named *select:*. Instead of including the method in each class, we only include it once in the root of the hierarchy.

$\models$ hasSameBehavior(remove, Set, Bag) $\wedge$ hasSameBehavior(remove, Bag, Set)

<u>Proof *of lemma 2*</u>:

We have to prove that the corollary holds. Since this dynamic logic has a minimal change semantics, only it is necessary to analyze the post-conditions of the method remove, because no other change is allowed to happen. That is to say, the instance of the corollary we have to prove is:

$\forall s \in$ Set.instances $\forall$e:Object $\bullet$ [s.remove(e)] e$\notin$s $\leftrightarrow$[s.migrates(Bag)] [s.remove(e)] [s.migrates(Set)] e$\notin$s

$\wedge\forall b \in$ Bag.instances $\forall$e:Object $\bullet$

[b.remove(e)] occurrences(b,e)=n-1$\leftrightarrow$[b.migrates(Set)] [b.remove(e)] [b.migrates(Bag)] occurrences(b,e)=n-1

Where n is equal to occurrences(b,e) before the removing.

The proof of the first implication of the first part of the conjunction is shown below, the rest of the proof is omitted, but it is similar to this.

[m1]	$\forall$s:Set.instances [s.migrates(Bag)] s.classifier=Bag	(from def. 1.6)
[m2]	$\forall s \in$ Set.instances $\bullet\forall$e:Object $\bullet$ occurrences(s,e) $\leq$1	(because sets have no repetitions)
[m3]	$\forall s \in$ Set.instances $\bullet\forall$e:Object $\bullet$ [s.migrates(Bag)] [s.*remove*(e)] occurrences(s,e)$\leq$0	(applying modus ponens to [h12], [m1] and [m2]
[m4]	$\forall s \in$ Set.instances $\bullet\forall$e:Object $\bullet$ [s.*migrates*(Bag)] [s.*remove*(e)] [s.*migrates*(Set)] occurrences(s,e)$\leq$0	(from [m3] because action *migrates* has no effect on the number of elements in the bag)
[m5]	$\forall s \in$ Set.instances $\bullet\forall$e:Object $\bullet$ [s.*migrates*(Bag)] [s.*remove*(e)] [s.*migrates*(Set)] e$\notin$s	(from [m4] because the expression occurrences(s,e)$\leq$0 is equivalent to the expression e$\notin$s)

<u>Proof of the theorem</u>:

Now we can prove the theorem:

[t1]	$\exists m_1,m_2 \bullet$ ($m_1 \in$ Set.operations $\wedge$ $m_2 \in$ Bag.operations $\wedge$ m_1.name= *remove* $\wedge$ m_2.name= *remove* $\wedge$ m_1.visibility=m_2. visibility $\wedge$ hasSameSignature(m_1,m_2) $\wedge$ hasSameBehavior(*remove*,Set,Bag))	(from lemma 1, lemma2, [h0], [h1] and [h7])
[t2]	Polymorphic(*remove*,Set,Bag)	(applying modus ponens to def. 1.1 and [t1])

6.2 Applying the metric

The polymorphism metric is applied to the Smalltalk Collection hierarchy in figure 4. In that figure polymorphic methods have been previously detected (using the definitions in section 4) and moved up in the hierarchy (i.e. the root class of the hierarchy is also the largest core class in the hierarchy). Methods appearing twice (or more times) in the hierarchy actually are non-polymorphic methods, for example, the method add: is non-polymorphic for Set and Bag.

<u>*Measuring Polymorhism to the Collection hierarchy:*</u>

polymorphism_measure(h_C) = polymorphic_methods(h_C) / #methods(h_C)

= polymorphic_methods(h_C) / 106 (because #methods(h_C) =106)

= (width(h_C) * #classes(h_C)

+ $\sum_{hi\in children(h_C)}$ polymorphic-methods(h_i – core(h_C))) / 106 (from definition of polymorphic_methods(h_C))

= (6 * 11 + $\sum_{hi\in children(h_C)}$ polymorphic-methods(h_i – core(h_C))) /106 (because width(h_C)=6, #(classes(h_C)=11)

= (66 + polymorphic-methods(sub-hierarchy-NotIndexedCollection) + polymorphic-methods(sub-hierarchy-IndexedCollection)) / 106

= (66 + 3 + 31) / 106 = 0.94 (or 94 %)

<u>*Value to the NotIndexedCollection sub-hierarchy (h_N):*</u>

polymorphic_methods(h_N) = (width(h_N) * #classes(h_N) + $\sum_{hi\in children(h_N)}$ polymorphic-methods(h_i – core(h_N)))

$= (1 * 3 + \sum_{hi \in children(} h_{N)}$ polymorphic-methods$(h_i$ – core$(h_N))$) = (1 * 3 + 0) = 3 (because children of h_N are trivial hierarchies)

Value to the IndexedCollection sub-hierarchy (hX):

polymorphic_methods(h_X) = (width(h_X) * #classes(h_X) + $\sum_{hi \in children(} h_{X)}$ polymorphic-methods$(h_i$ – core$(h_X))$) $= (4 * 7 + \sum_{hi \in children(} h_{X)}$ polymorphic-methods$(h_i$ – core$(h_X))$) = (28 + polymorphic-methods(sub-hierarchy-FixedSizeCollection) + polymorphic-methods(sub-hierarchy-VariableSizeCollection)) = (28 + 0 + 3) = 31

Value to the VariableSizeCollection sub-hierarchy (hV):

polymorphic_methods(h_V) = (width(h_V) * #classes(h_V) + $\sum_{hi \in children(} h_{V)}$ polymorphic-methods$(h_i$ – core$(h_V))$)

$= (1 * 3 + \sum_{hi \in children(} h_{V)}$ polymorphic-methods$(h_i$ – core$(h_V))$) = (3 + 0) = 3 (because children of h_V are trivial hierarchies)

It can be observed in the outcome, the high degree of polymorphism of the Collection hierarchy (reaching the value of 94 %), which contributes potentially to the readability, extensibility, and ultimately to their maintainability. However, some studies, that should further be confirmed, indicate that polymorphism may increase the probability of faults in O-O software –see for example, Benlarbi and Melo (1999).

7. Towards a Validation of the Polymorphism Metric

There are two strategies to corroborate or falsify the validity of metrics: the *theoretical* and the *empirical* validation. The theoretical validation is mainly based on mathematical proofs that allows us to formally confirm that the measure does not violate the properties of the empirical systems, the definition models and criteria. On the other hand, the empirical validation consists on the realization of experiments and observations on the real world in order to corroborate or falsify the metric.

In addition, validation approaches can be classified according to the class of attribute that is taken into consideration. From this point of view a metric is valid *internally* or “valid in the narrow sense” (Fenton and Pfleeger, 1997), if it analyses properties that are inherent of the system, while a metric is valid *externally* or “valid in the wide sense” if it considers higher level characteristics (e.g., cost, quality, maintainability, etc.) mainly for prediction purposes. Finally, some metrics can be measured directly (such as the number of classes or methods of a class hierarchy), while others can only be measured indirectly by means of an equation or model.

In this section, we will analyze aspects of the *theoretical* validation for the polymorphism metrics discussed and exemplified in sections 5 and 6. These metrics consider *internal* attributes of a product entity, e.g., an O-O design specification of a software system. In a general sense, the Kitchenham et at. (1996) assumption is that in order for a measure to be valid these two conditions must be held: 1) the measure must not violate any necessary property of its elements; 2) each model used in the process must be valid. The structural framework of Kitchenham et at. (1996) can be combined with the axiomatic framework of Zuse (1998) to yield a wider conceptual framework (Olsina et al., 2000). Regarding the proposed conceptual framework in order to decide whether a metric is valid, it is necessary at least to check:

- ✓ *Attribute validity*, i.e., whether the attribute is actually exhibited by the entity being measured. For a given attribute, there is always at least an empirical relationship of interest that can be captured and represented in the numerical domain, enabling us to explore the relationship analytically. This can imply a theoretical and/or empirical validation.

- ✓ *Unit and Scale Type validity*, i.e., whether the measurement unit and scale type being used are an appropriate means of quantifying the internal or external attribute. When we measure a specific attribute of a particular entity, we consider a scale type and unit in order to obtain magnitudes of type value. Thus, the measured value can not be interpreted unless we know to what entity is applied, to what attribute is measured and in what unit is expressed (i.e., the empirical and numeric relational systems should be clearly specified). On the other hand, a scale type is defined by admissible transformations of measures.

- ✓ *Instrument validity*, i.e., whether any model underlying a measuring instrument is valid and the same one is properly calibrated. In order to obtain the measured value we can do it either manually or automatically by using partial or totally a measurement instrument (a software tool).

- ✓ *Protocol validity*, i.e., whether an acceptable measurement protocol has been used in order to guarantee repeatability and reproducibility in the measurement process.

Regarding the polymorphism metric, some empirical considerations should be made. As aforementioned, the hierarchies(S) function returns the collection containing all the non-trivial disjoint hierarchies defined in S. This guarantees, for example, that the intersection between two hierarchies gives the empty set. In addition, we are only considering tree hierarchies which allow us to model single inheritance (Java and Smalltalk languages, among others, only support single inheritance). In order to try guarantee the ratio scale for the *polymorphism_measure* metric, we started to investigate the modified extensive structure and the additive properties discussed in Zuse (1998). However, the initial results draw that the metric does not accomplish the independence condition C1, and the axiom of weak monotonicity. So, for that metric the absolute scale has in principle been validated as follows:

Attribute	Scale Type	Unit	Criteria and Properties that Apply
#classes(h)	Absolute	Number of classes in h	✓ These internal attributes are exhibited in O-O design and implementation specifications. They are simply direct metrics. ✓ Different hierarchy specifications may have different number of classes, methods, etc. for the respective attribute. Conversely, different hierarchy specifications may have the same number of classes, methods, etc. ✓ They fulfill the representation condition ✓ The unit and scale type are defined and confirmed. Accordingly, they are obtained by counting elements where an absolute scale is generally implied (but not always). The only possible transformation is the identity.
#methods(h)		Total number of methods in h (regarding bags)	
width(h)		Number of polymorphic methods (to h)	
#hierarchies(S)		Number of hierarchies in the S specification	
Polymorphic_methods.	Absolute	(Number of polymorphic methods * Number of classes)	✓ It is an indirect metric. The equation is shown in Section 5. ✓ The unit and scale type are defined. It yields an absolute scale.
Polymorphism_measure.	Absolute	[(Number of polymorphic methods * Number of classes) / Total number of methods to h], It represents the percentage of polymorphic methods of a hierarchy	✓ It is an indirect metric. The equation is shown in Section 5. ✓ It fulfills the representation condition (That is, greater number of polymorphic methods with regard to the total amount of methods of a hierarchy leads to a higher degree of polymorphism –hence, the specification can be more understandable, reusable and extensible). The absence of polymorphic methods in a hierarchy yields a zero value. Conversely, the 1 value (or 100%) means that all methods are polymorphic. ✓ The unit and scale type are defined. It yields an absolute scale as demonstrated by the theorem 7.1.

Figure 5: Descriptions of theoretical validity for the polymorphism metric and its elements

<u>Theorem 7.1</u>: The scale type of the metric is absolute.

Proof: Let m = A/B be the metric, and let A, B be absolute values (1)

Where A represents the *polymorphic_measure* attribute; and B represents the total number of methods of a hierarchy (*#methods(h)*). It is always satisfied that A <= B, and therefore it holds that $A \subseteq B$. The relationship between A and B can be described by:

A = c B; with c>0.   (2)

Replacing (1) in (2), the following equation is obtained: m = c B/B = c.

The resulting m is an absolute scale, as considered by Zuse. Percentage measures can be used as an absolute scale, but they do not assume an extensive structure (Zuse, 1998), pp. 237-238.

Figure 5, shows descriptions of the theoretical validity for a set of used functions for the metric. The target entity is an O-O design specification or a source code of an O-O program. The instrument validity is applicable because data collection and calculations can be carried out automatically. The main algorithm is supported by the recursive model.

Ultimately, the measure of polymorphism of a set of disjoint hierarchies defined in the S specification is computed by making an average as shown in Section 5. This statistical analysis is allowed to magnitudes of an absolute scale type.

Concluding Remarks

Although quality is not easy to evaluate since it is a complex concept compound by different aspects, several properties that make a good O-O design have been recognized and widely accepted by the software engineering community.

We agree that both the traditional and the new O-O properties or attributes should be analyzed in assessing the quality of O-O design. But we believe that it is necessary to pay special attention to the concepts and metrics for *polymorphism*, since it should be considered one of the key concerns in determining the quality of an O-O software system.

In this paper, we have given a rigorous definition of polymorphism in the framework of the M&D-theory (Pons et al. 99). Besides, on top of this formalization we propose a metric for measuring polymorphism, that provides an objective and precise mechanism to detect and quantify dynamic polymorphism. It is proven that the metric is valid regarding a theoretical validation framework. Furthermore, it is important to remark that the metric takes information coming from the first stages of the development lifecycle giving developers the opportunity to early evaluate and improve the quality of the software product.

References

Abreu, F.B. and Carapuça,R. Object Oriented Software Engineering: Measuring and controlling the development process, 4th International Conference on Software Quality, Virginia, USA, 1994.

Bansiya, J. , Assessing quality of object-oriented designs using a hierarchical approach, OOPSLA'97 Workshop#12 on Object-oriented design quality. Atlanta, USA, October 1997.

Bansiya, J. Davis, C. Etzkorn L. and Li, W. (a) An entropy-based complexity measure for object oriented designs, Theory and Practice of Object Oriented Systems, 5(2), 1999.

Bansiya, J. Davis, C. Etzkorn L. and Li, W. (b) A class cohesion metric for object oriented design, Journal of Object Oriented Programming, January 1999.

Benlarbi, S., Object-oriented design metrics for early quality prediction, OOPSLA'97 Workshop#12 on Object-oriented design quality. Atlanta, USA, October 1997.

Benlarbi, S. and Melo, W. Polymorphism measures for early risk prediction, In International Conference of Software Engineering (ICSE'99), Los Angeles, CA, USA, 1999.

Briand, L. Devanbu P.and Melo, W., An investigation into coupling measures for C++. In International Conference of Software Engineering (ICSE'97), Boston, USA, May 1997.

Chidamber S.and Kemerer, C., A metric suite for object oriented design, IEEE Transaction on Software Engineering, 20. 1994.

Chen J.and Lu, J., A new metric for object oriented design, Information and Software Technology, 35. 1993.

Fenton, N.E.; Pfleeger, S.L., Software Metrics: a Rigorous and Practical Approach, 2nd Ed., PWS Publishing Company. 1997.

Harel, D., Kozen, D. and Tiuryn, J. Book on Dynamic Logic. to appear.

Kim, E. Chang, O. Kusumoto, S. Kikuno, T., Analysis of metrics for object oriented program complexity, Procs. 18th Annual International Computer Software and applications Conference, COMPSAC´94.

Kitchenham, B., Pfleeger, S. L., Fenton, N., 1996, Towards a Framework for Software Measurement Validation. *IEEE Transactions on Software Engineering*, 21(12), pp. 929-944

Lalonde, Wilf, Discovering Smalltalk. Addison Wesley. 1994.

Li W.and Henry, S., Object oriented metrics that predict maintainability, The Journal of Systems and Software, #23. 1993.

Moore, Ivan, Automatic inheritance hierarchy restructuring and method refactoring, in proceedings of OOPSLA'96, ACM Sigplan, vol.31,no.10, October 1996.

Olsina, L.; Pons, C; Rossi, G.; 2000, Toward Metric and Model Validation in Web-site QEM. Submitted paper.

Poulin, J., Measuring Software Reuse- Principles and Practices and Economical Models, Addison Wesley, 1997.

Pons, C. Baum G.and Felder, M., Foundations of Object-oriented modeling notations in a dynamic logic framework, In Fundamentals of Information Systems, Chapter 1, T.Polle,T.Ripke,K.Schewe Editors, Kluwer Academic Publisher, 1999.

Pons,C., Ph.D Thesis, Faculty of Science, University of La Plata, Buenos Aires, Argentina, http://www-lifia.info.unlp.edu.ar/~cpons/ (1999)

Price M.and Demurjian, S., Analysing and measuring reusability in object-oriented design, in proceedings OOPSLA'97, Atlanta, USA, 1997.

Tegarden, D. Sheetz S.and Monarchi, D., Effectiveness of traditional software metrics for object-oriented systems. 25th Annual Conference of System Science, Maui,HI. 1992.

UML 1.3, Object Management Group, The Unified Modeling Language (UML) Specification – Version 1.3, in http://www.omg.or (1999).

Wieringa R.and Broersen, J., Minimal Transition System Semantics for Lightweight Class and Behavior Diagrams, In PSMT Workshop on Precise Semantics for Software Modeling Techniques, Ed: M.Broy, D.Coleman, T.Maibaum, B.Rumpe, Technische Universitat Munchen, Report TUM-I9803, April 1998.

Wilde N.and Huitt, R., Maintenance support of object-oriented programs, IEEE Transactions on Software Engineering, 18. 1992.

Woolf, B., Polymorphic hierarchy. The Smalltalk Report. January 1997.

Zuse, H., A Framework of Software Measurement, Walter de Gruyter, Berlín-NY. 1998.