METRICS FOR SOFTWARE CONCEPTUAL MODELS

Editors

Marcela Genero
Mario Piattini
Coral Calero

University of Castilla–La Mancha, Spain

ICP

Imperial College Press

Published by

Imperial College Press
57 Shelton Street
Covent Garden
London WC2H 9HE

Distributed by

World Scientific Publishing Co. Pte. Ltd.
5 Toh Tuck Link, Singapore 596224
USA office: 27 Warren Street, Suite 401-402, Hackensack, NJ 07601
UK office: 57 Shelton Street, Covent Garden, London WC2H 9HE

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library.

METRICS FOR SOFTWARE CONCEPTUAL MODELS

ISBN 1-86094-497-3

Typeset by Stallion Press
Email: sales@stallionpress.com

Printed in Singapore.

Dedicated with admiration and warmth to Isidro Ramos,
with whom the authors have had the pleasure to work.

CONTENTS

Computer Science from California Polytechnic State University, San Luis Obispo, and his MS and PhD in Information Systems from the University of Colorado, Boulder. His research interests include developing theoretically grounded models and metrics for evaluating business processes, investigating the problems people have shifting to emerging technologies, and determining the business value of information technology. Jim generally teaches the more technical courses in information systems including object oriented technology, systems analysis and design, database theory and practice, and business data communications.

GEERT POELS

He is a lecturer in the Department of Management Information, Operations Management, and Technology Policy of the Faculty of Economics and Business Administration at Ghent University and guest professor in the Center for Industrial Management of the Catholic University of Leuven. He holds a Master Degree in business engineering (1991) from the Limburg Business School and a Master Degree in computer science (1993) and PhD in applied economics (1999) from the Catholic University of Leuven. His research interests include theoretical foundations of software measurement, metrics for event-based models, OO software metrics, conceptual model quality, accounting system data models and ontologies, and functional size measurement. From 1999 till 2001, he was the Program Chair of the FESMA series of conferences on software measurement. Since 1996, he has co-organized workshops on OO software metrics, business software component identification, quantitative approaches in OO software engineering, and conceptual modelling quality at the OT, ECOOP and ER conferences. He has published papers in *IEEE Transactions on Software Engineering and Information and Software Technology*, and presented at conferences such as CAiSE, ER, and OOIS.

LUIS REYNOSO

Luis is an assistant professor at the National University of Comahue, Neuquen, Argentina. He received his MSc degree in computer science from the National University of South Argentine in 1993. He also obtained a Magister in computer science at the same university in 2003. He was a fellow in the International Institute of Software Technology, one of the

Research and Training Centres of the United Nations University, researching on a project about object-oriented design patterns. He has published papers in several international conferences. His research interests are focused on object-oriented metrics and the combination of formal and informal methods applied to software engineering.

FRANCISCO RUIZ

He obtained his PhD in computer science from the University of Castilla-La Mancha (UCLM) and MSc in Chemistry-Physics from the University Complutense de Madrid. He is a full time associate professor in the Department of Computer Science at UCLM in Ciudad Real (Spain). He was Dean of the Faculty of Computer Science between 1993 and 2000. Previously, he was a Computer Services Director in the aforementioned university (1985–1989) and he has also worked in private companies as an analyst programmer and project manager. His current research interests include software process technology and modelling, software maintenance, methodologies for software project planning and managing, and advance database modelling. In the past, other work topics have included GIS (geographical information systems), educational software systems, and deductive databases. He has written eight books and fourteen chapters on the aforementioned topics and has published ninety papers in magazines, congresses and conferences. He belongs to several scientific and professional associations — ACM, IEEE-CS, ISO JTC1/SC7, EASST.

MANUEL SERRANO

Manuel Serrano received his MSc and Technical degree in computer science from the University of Castilla-La Mancha. Nowadays, he is developing his PhD at UCLM. He is an assistant professor in the Department of Computer Science at the University of Castilla-La Mancha University in Ciudad Real. He is a member of the Alarcos Research Group, in the same university, specialising in information systems, databases and software engineering. He is the secretary of the ATI (Computer Technicians Asociation) group in Castilla — La Mancha. His research interests are datawarehouse quality and metrics, and software quality.

Chapter 5

MEASURING OCL EXPRESSIONS: AN APPROACH BASED ON COGNITIVE TECHNIQUES

LUIS REYNOSO*, MARCELA GENERO[†,a] and MARIO PIATTINI[†,b]

*National University of Comahue, Department of Computer Science
Neuquén – Argentina
lreynoso@uncoma.edu.ar

[†]Alarcos Research Group, Department of Computer Science
University of Castilla La Mancha, Ciudad Real – Spain
[a]Marcela.Genero@uclm.es
[b]Mario.Piattini@uclm.es

1. Introduction

Class diagram quality is clearly a crucial issue that must be evaluated (and improved if necessary) in order to get quality OO software. This fact is corroborated by the huge amount of metrics that can be applied to UML (OMG, 2003c) class diagrams at a high level design stage, that have been recorded in relevant literature. Most of these studies are focused on the measurement of internal quality attributes such as structural complexity, coupling, size, etc. However, none of the proposed metrics take into account the added complexity involved when class diagrams are complemented by expressions written in Object Constraint Language (OCL).

OCL, defined by OMG (2003b), has became a fundamental language in developing OO software using UML, as it allows complete and consistent

UML modelling. A model specified in a combination of the UML and OCL languages is mentioned in Warmer and Kleppe (2003), as a UML/OCL combined model, or just a UML/OCL model. OCL enriches, for example, UML class diagrams with expressions that specify semantic properties of a model (Gogolla and Richters, 2001). The OCL expressions are unambiguous and make the model more precise and more detailed (Warmer and Kleppe, 2003) improving its understandability at early stages of OO software development. Moreover OCL is essential in building consistent and coherent platform-independent models (PIM) and helping to raise the level of maturity of the software process (Warmer and Kleppe, 2003).

Even though applying OCL to software specification has great potential for improving software quality and software correctness (Hennicker et al., 2001), there are no metrics for OCL expressions.

The theoretical basis for developing quantitative models relating to structural properties and external quality attributes has been provided by Briand and Wüst (2001). In this work, we assume that a similar representation holds for OCL expressions. We implement the relationship between the structural properties on one hand, and external quality attributes on the other hand (see Fig. 1).

We hypothesise that the structural properties (such as coupling, size, length, etc.) of an OCL expression have an impact on its cognitive complexity. By cognitive complexity we mean the mental burden of the persons who have to deal with the artifact (e.g. modellers, designers, maintainers). High cognitive complexity leads to a reduction in the understandability of an artifact, and this leads to undesirable external qualities, such as decreased maintainability.

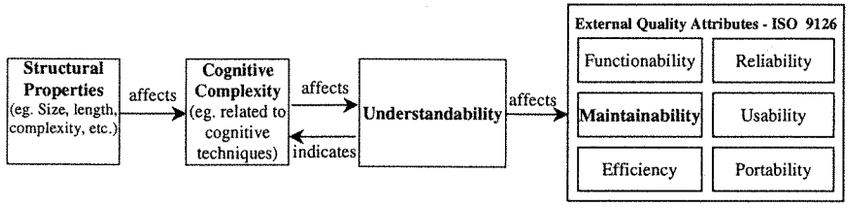

Fig. 1. Relationship between structural properties, cognitive complexity, understandability and external quality attributes (based on Briand and Wüst (2001) and ISO/IEC 9126, (2001)).

We suppose that OCL expression structural properties have an impact on the cognitive complexity of modellers. From a cognitive point of view, Cant et al. (1994) argue that measuring complexity should affect attributes of human comprehension since complexity is relative to human cognitive characteristics. Therefore, we have considered the cognitive techniques applied by modellers when they try to understand an OCL expression. These techniques are: "chunking" and "tracing" (Cant et al., 1994; Cant et al., 1992; El-Eman, 2001).

Therefore, this chapter pursues two main goals:

(1) To propose, in a methodological way, a set of metrics for measuring structural properties of OCL expressions, considering those OCL concepts specified in its metamodel (OMG, 2003b) which involve the use of two cognitive techniques — "chunking" and "tracing".
(2) To assure that the proposed metrics measure what they purport to measure through their theoretical validation, following a property-based framework proposed by Briand et al. (1996), (1997) and (1999).

In relation to our first aim we start in the following section by describing some concepts of the cognitive model of Cant et al. (1992), which we have used as a basis to define the metrics. Section 3 presents OCL and its concepts related to "tracing" and "chunking". The proper definition of the metrics is presented in Sec. 4, whilst their theoretical validation is presented in Sec. 5. Finally, in the last section some conclusions are drawn and future work is described.

2. Cognitive Techniques for Software Comprehension

The Cognitive Complexity Model (CCM) defined by Cant et al. (1992) gives a general cognitive theory of software complexity that elaborates on the impact of structure on understandability (El-Eman, 2001). Although the study of Cant et al. (1992) has been considered a reasonable point of departure for understanding the impact of structural properties "on understandability of code and the coding process", we believe that this model can also be applied to UML developers when they try to understand OCL expressions. The underlying rationale for the CCM argues that

comprehension consists of two techniques or processes — "chunking" and "tracing", that are concurrently and synergistically applied in problem solving. Cant el al. (1992) argue that both techniques have implication for software complexity:

- "Chunking" technique — a capacity of short term memory, involves recognising groups of statements (not necessarily sequential) and extracting information from them which is remembered as a single mental abstraction — a "chunk" (Cant et al., 1992).
- "Tracing" technique — involves scanning, either forward or backward, in order to identify relevant "chunks" (El-Eman, 2001), resolving some dependencies.

Cant et al. (1992) argue that it is difficult to determine what constitutes a "chunk" since it is a product of semantic knowledge. For our purposes, we will consider an OCL expression as a "chunk" unit, whilst the comprehension of an operation, an attribute or a relationship with their associated OCL expressions are also considered "chunks". Henderson-Sellers (1996) notes that "tracing" disrupts the process of "chunking".

The comprehension of a particular "chunk" is the sum of three components: (1) the difficulty of understanding the "chunk" itself; (2) the difficulty of understanding all the dependencies on the "chunks" upon which a particular "chunk" depends, and (3) the difficulty of "tracing" these dependencies to those "chunks" (El-Eman, 2001). "Tracing" is applied when a method calls for another method to be used in a different class, or when an inherited property needs to be understood (El-Eman, 2001). UML modellers or developers also commonly perform these cognitive techniques during the understandability of OCL specifications.

3. OCL

As our intention is to define metrics for OCL expressions which are specified attached to a UML class diagrams, it is important to introduce OCL and its concepts as used in the definition of our metrics. Firstly, we will briefly describe OCL and its main elements in Sec. 3.1. In Sec. 3.2., we will structure the presentation of the OCL concepts in terms of their relation to the cognitive techniques mentioned in the previous section.

3.1. *OCL characteristics*

OCL is a textual specification language defined to solve different problems:

- UML is limited in its expressiveness, and many constraints cannot be defined using only UML graphical features (Cook et al., 2001), (Warmer and Kleppe, 2003).
- Frequently the system properties and constraints that cannot be defined using UML diagrams are defined using natural languages and this leads to misinterpretations, misunderstanding (Warmer and Kleppe, 1999), and ambiguities (OMG, 2003b).
- The use of formal methods can help to alleviate this problem, in order to specify correctly the system behaviour, but the use of formal methods by the object technology community' members requires a strong mathematical background, and formal methods are not a subject with which the average business or system modeller are familiar (OMG, 2003b).
- To provide precise information in the definition of standards, like the UML standard itself, use of a precise language is required.

OCL was defined as a textual add-on to the UML diagrams (Cook et al., 2001). Its main elements are OCL expressions that represent declarative and side-effect-free textual descriptions that are associated to different features of UML diagrams. OCL expressions add precision to UML models beyond the capabilities of the graphical diagrams of UML. Although OCL is considered in OMG (2003b) to be a formal language easy to read and write, the misuse of the language can lead to complicated written OCL expressions. Warmer and Kleppe (1999) give some tips and hints in writing OCL expressions (these recommendations are still valid although OCL has been modified through different versions); Furthermore, they recognise that the way OCL expressions are defined has a large impact on readability, maintainability and the complexity of the associated diagrams.

As previously mentioned, we will consider an OCL expression as a "chunk". An OCL expression is a suitable chunk unit, which modellers should understand as a whole declaration constraining an aspect of the system being modelled. It is therefore important to describe several concepts related to an OCL expression:

- Each OCL expression is written in the context of an instance of a specific type. This instance, *self*, provides a point of reference for interpretation of the expression, and is commonly referred to as the contextual instance.

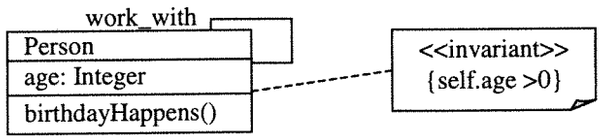

Fig. 2. Example of an invariant.

- The context in which an expression is written is introduced through the keyword *context*. Any OCL expression starts with the definition of the context, that involves the keyword *context* followed by the name of a type, then any specification of *self* within the OCL expression will be associated with the type declared in the context declaration.
- OCL expressions in UML class diagrams are used most importantly: to specify invariants[1] on classes and types in the class diagram, to specify constraints on operations and methods, to describe pre- and post conditions on operations, to specify initial values and derivation rules for attributes, to specify query operations, and to introduce new attributes and operations (OMG, 2003b).
- Invariants, preconditions and postconditions are constraints stereotyped respectively by ≪invariant≫, ≪precondition≫ and ≪postcondition≫. Although it is common for stereotypes to be attached to a UML feature using a graphical notation such as a note box, as shown in Fig. 2, the quantity of OCL expressions defined for a class diagram is significantly higher and this clutters the understandability of the UML diagram. For that reason all the OCL expressions shown in this chapter will be written in textual form and will not be shown using a note box.

Example 1:

An invariant definition for the Person class is:

context Person inv:
self.age > 0

The keyword *inv* means that the OCL expression, which comes after the colon, is an invariant expression. In fact, the *inv* keyword denotes

[1]An invariant is a constraint that must hold true anytime in the system, whereas pre- or post-conditions represent respectively a true condition that must be true just before/after the execution of an operation.

the stereotype ≪invariant≫. This expression means that all the values of any instance of the class Person must not be zero or lower. *self.age* uses a dot notation to refer to a property *age* of the object represented by *self*, however it is possible that *self* will be implicit in the expression (in the example is possible to simply write *age* instead of *self.age*). In general, whenever a property called *property* is specified without the object to which it applies — in the form of "*property*" instead of "*object.property*" —, the object is left out, or it is implicit in the specification of the *property* property.

Example 2 introduces a postcondition restriction for the *BirthdayHappens* operation of Person class. It is a postcondition as the post keyword is used, which in turn refers to a ≪postcondition≫ stereotype. This postcondition means that the *age* of a person is incremented by one when the birthday of a person had happened (age@pre is used to refer to the previous value of *age* just before the execution of BirthdayHappens).

Example 2:

context Person::BirthdayHappens()
post: $age = age@pre + 1$

Before giving more examples, we will introduce two important concepts used in the following sections:

- Properties: an attribute, an association-end, and side-effect-free operation or method are considered properties on an object (OMG, 2003b). The way an object property is specified in an OCL expression is by using a dot notation. *Object.property 1* refers to a *property 1* property of *object* wherever *object* is a valid reference to an object. To illustrate this concept, see Example 3.

Example 3:

In the following expression the *work_with* property is used in an expression, meaning that a person cannot work with himself or herself. In this case work_with represents an association-end property.

context Person inv:
not self.work_with.exists(self)

- Classifier: a classifier is a UML metaclass which represents a type, a class, an interface, an association (acting as types) and datatypes (OMG, 2003b). Each classifier defined within a UML model represents a distinct OCL type (OMG, 2003b).

3.2. *OCL concepts related to cognitive techniques*

The understanding of an OCL expression as a "chunk" involves a strong intertwining of "tracing" and "chunking" techniques. We need to understand which OCL concepts, specified in its metamodel (OMG, 2003b), are relevant to these techniques. Analysis of each of these techniques in turn leads to the identification of structural properties, which can be measured.

In order to describe the OCL concepts which involve "chunking" we have basically considered those concepts which belong to one expression (the chunk) and which do not require solving dependencies to other chunks. In order to analyse "tracing", however, we have considered those OCL concepts that imply solving dependencies to other chunks. *Self* is used as the main concept related to the OCL expression itself, i.e. to the "chunking" technique. Other instances (object or object collections), whose types are different to the type represented by the contextual instance, are commonly accessed by "tracing" techniques.

Table 1 shows the main OCL concepts involved in these cognitive techniques.

3.2.1. *OCL concepts related to the "Chunking" technique (group 1)*

In this section, we will describe the OCL concepts related to the "chunking" cognitive technique of group 1 (mentioned in Table 1).

In this group, we have included those OCL concepts that are intrinsic to the language itself, allowing the modellers:

- To specify a variable definition.
- To use conditional expressions.
- To use predefined iterator expressions.
- To use literals, boolean operations, etc.

Table 1. OCL concepts which involve "tracing" or "chunking".

Cognitive technique	**OCL concepts related to the cognitive technique**		**Common characteristics of the group of OCL concepts**
Chunking	Group 1	Variable definitions through *let* expression, *if* expression condition, predefined iterator variables, literals, etc.	OCL facilities related to the language itself.
	Group 2	Reference to attributes or operations of the contextual instance, values postfixed by @pre, variables defined through ≪definition≫ constraints.	OCL concepts related to the contextual instance and some of its properties, values before the execution of an operation (that is, properties postfixed by @pre) of the contextual instance, variables defined through ≪definition≫ constraints in the type represented by the contextual instance, etc.
Tracing	Navigation and collection operation, parameter whose type are classifiers defined in the class diagram, Messaging, etc.		OCL concepts which allow an expression to use properties belonging to other classes or interfaces, different to the type of the contextual instance

The concepts are:

- **Let expressions**
  The *let* expression allows the definition of a variable to stand for a subexpression, and wherever the subexpression needs to be specified it is possible to reuse the variable instead, as a shorthand. The variable declared through a *let* definition is allowed to be used only inside the OCL expression which defines it (see Example 4).

  Example 4:

  In the following expression taken from OMG (2003b) a variable called *income* is defined to represent the function self.job.salary−>sum().

context Person inv:
let income : Integer = self.job.salary–> sum() in
if isUnemployed then income < 100
else income ≥ 100 endif

- **If-expressions**
  The *if, then, else* and *endif* keywords allow one to write a conditional expression, this kind of expression should always have a value. The *if* expression takes the form of "*if b then e1 else e2 endif*". Both OCL expressions, *e1* and *e2*, should be of compatible types and neither *e1* nor *e2* can be omitted from the expression (OMG, 2003b) (see Example 5).

  Example 5:

  The following example is taken from Warmer and Kleppe (1999):

  context customer inv:
  title = (if isMale = true
  then 'Mr.'
  else 'Ms.'
  endif)

- **Collection literals**
  OCL provides four specific collection types: set, bag, orderedset and sequence[2]. The simplest way to define a collection is through collection literals. The syntax used is to define the collection elements inside curly brackets and separated by commas. Also collection literals can be specified by means of interval declaration. Another kind of literal that can be specified is a tuple[3] (see Example 6).

  Example 6:

  Set {'apple', 'strawberry', 'orange'},
  Sequence {'apple, 'orange'},
  Bag {1,3,4,3,2},
  Tuple {name: String = 'John ', age: Integer = 10}

[2]A set is a collection of different elements whereas a bag can contain duplicated elements. An OrderedSet is a set in which the element are ordered. A sequence is a bag but their elements are ordered.
[3]A tuple consists of named parts, each of which can have a distinct type.

Table 2. Predefined iterator expressions.

Collection	Set	Bag	Sequence
exists any forAll one isUnique collect	select reject collectNested sortedBy	select reject collectNested sortedBy	select reject collectNested sortedBy

- **Logical operators**
  The Boolean type is a predefined type composed of two values: true and false. OCL defines the following logical operators for Boolean: *or*, *xor*, *and*, *not* and *implies*. It is common to use logical operators in OCL expression because they represent general connectors of subexpressions.
- **Predefined iterator expressions**
  The semantic of predefined iterator expressions is defined in terms of an *iterate* expression. The set of standard iterator expressions defined in OCL (OMG, 2003b) is included in Table 2. The *reject* operation shown in Example 7, allows us to obtain a subset from a collection, its syntax is specified using the arrow-syntax: *collection–>reject(Boolean-expression)*. The subset obtained from the collection using *reject*, is composed of all the elements of the collection from which the expression evaluates to *false*. However, the operation can adopt three different forms (sse Example 7). The last two include an iterator variable, being the iterator, in the last form, specified by its type. The iterator variable is used to refer explicitly to the collection elements. The use of these predefined expressions involves dealing with collections and iterators.

  Example 7:

  collection–>reject(Boolean-expression)
  collection–>reject(v | Boolean-expression-with-v)
  collection–>reject(v: Type | Boolean-expression-with-v)

3.2.2. *OCL concepts related to the "Chunking" technique (group 2)*

In this section, we will describe the OCL concepts related to the "chunking" cognitive technique of group 2 (mentioned in Table 1).

Due to the fact that OCL expressions are textual add-on to UML class diagrams there should be OCL mechanisms for refering, for example, to class diagram elements. Implicitly, an OCL expression is associated to a specific element (of a class diagram) through the contextual instance *self*, because *self* is the main point of reference for the comprehension of the OCL expression. In this group, we have included those OCL concepts that allow one to refer to some properties of the Classifier:

- Attributes and operations of the Classifier (which is represented by *self*).
- Variable definitions known in the context of the Classifier (which is represented by *self*).
- Some OCL predefined operations which relate the Classifier (which is represented by *self*) with other supertypes of this Classifier.

The concepts are:

- **Accessing attributes and operations belonging to the Classifier represented by *self***
  As was previously mentioned, an OCL expression can refer to a Classifier and its properties. Using the contextual instance and the dot notation it is possible to refer to attributes and operations defined in the Classifier represented by *self*. Considering the OCL expression shown in the Example 8, two properties are referred:
  - The *level* attribute belonging to Person, and
  - The *age()* operation belonging to the same type.

  Example 8:

  context Person inv:
  self.level = "Senior" implies self.age() = 21

- **≪definition≫ constraints**
  To allow the reuse of a variable and/or operation over multiple OCL expressions is possible to define a ≪definition≫ constraint, using the keyword *def*. In fact, this OCL expression means a stereotype ≪definition≫, and the constraint is attached to a Classifier. The keyword *def*, can be used after the attribute or operation definition. The

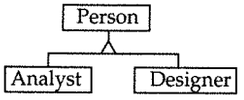

Fig. 3. An example of a class diagram.

constraint is exemplified as follows:

Example 9:

In the next OCL expression a variable called *income is defined.*

context Person
def: income: Integer = self.job.salary−> sum()

The *income* variable is known in the same context as any property of Person. For example in:

context Person inv:
if self.isUnemployee then
sueldo < 100 else
sueldo ≥ 100 endif

- **Predefined properties that can be applied to any object**
  The following predefined operations can be applied to all objects[4]:
  - **oclIsTypeOf (t :OclType): Boolean:** The operation returns true if its argument (*t*) is equal to the type of *self*.
  - **oclIsKindOf (t :OclType): Boolean:** The operation determines whether *t* is either the direct type or one of the supertypes of an object.

  Example 10:

  According to the class diagram shown in Fig. 3, the following examples are defined in the context of the Designer class:

  self.oclIsTypeOf(Person) = false
  self.oclIsKindOf(Person) = true
  self.oclIsTypeOf(Designer) = true
  self.oclIsKindOf(Designer) = true

[4]Not all the predefined properties on all objects are included in this section, only those related to "chunking" — group 2.

– **oclAsType (t :OclType): instance of OclType:** Property of supertypes when they are overridden within a type can be accessed through oclAsType().

Example 11:

If B is supertype of A then it is possible to write:

context B inv:
self.oclAsType(A).p1

in order to refer to the *p1* property of A.
The three properties are included in this group as they are commonly used with inheritance concepts of the Classifier represented by *self*.

- **Accessing previous values in postconditions**
Whenever a property is postfixed with the keyword "@pre" in a postcondition, the value accessed is the property value before the execution of the operation (where the postcondition is defined).

Example 12:

In the following example taken from Cook et al. (2001) the *usage* property refers to the property of Bathroom whereas *usage@pre* refers to the value of *usage* of Bathroom before the execution of the *uses* operation.

context Bathroom::uses (g: Guest)
pre: ....
post: usage = usage@pre + 1

3.2.3. *OCL concepts related to the "Tracing" technique*

In this section, we will describe the OCL concepts related to the "tracing" cognitive technique. This technique was defined in Sec. 2. The OCL concepts related to "tracing" techniques allow the modeller to write an expression using properties belonging to other classes or interfaces, different to the Classifier that self represents, such as:

- **Navigations**
Starting from a specific object, it is possible to navigate an association in the class diagram, to refer to other objects and their properties (OMG, 2003b). A relation is navigated when we use the rolename of the opposite association-end of a relation, that links the class where the expression is

defined with another class in the diagram class (when the association-end is missing we can use the name of the type at the association-end as the rolename). The result of a navigation is a single object or a collection of objects depending on the multiplicity of the association-end (Richters, 2002). The syntax uses the dot notation followed by an association-end property. It is possible to navigate many relationships in order to access as many properties as needed in an expression.

Example 13:

The following expression is specified for the class diagram of Fig. 4.

context LoyaltyProgram inv:
membership.card –> forAll (goodThru = Date::fromYMD (2007,1, 1))
and self.customer–>forAll (age()>30)

membership.card used in the expression represents a navigation from LoyaltyProgram to CustomerCard. It navigates two relationships: one from LoyaltyProgram to Membership (an association class), and another from Membership to CustomerClass. In the former relationship there is no rolename attached to the association-end where Membership is the sink class, and for that reason the name of the class is used. Meanwhile, in the latter relationship, the navigation is represented by its rolename "card". The expression also contains another navigation self.customer.

- **In, Out and In/Out Parameters, and Return Values**
Operations may have *in*, *out*, *in/out* parameters. If the operation has *out* or *in/out* parameters, the result of this operation is a tuple containing all *out*, *in/out* parameters and the return value (OMG, 2003b).
- **Collection Operations**
OCL defines many operations for handling the elements in a collection. The operations allow the modeler to project new collections from the existing one. Operations like *select*, *reject*, *iterate*, *forAll* and *exists*, take each element in a collection and evaluate an expression for them. The expression evaluated for each collection can be defined in terms of new navigations. We will take into account those expressions of collection operations which are defined in terms of other navigations.

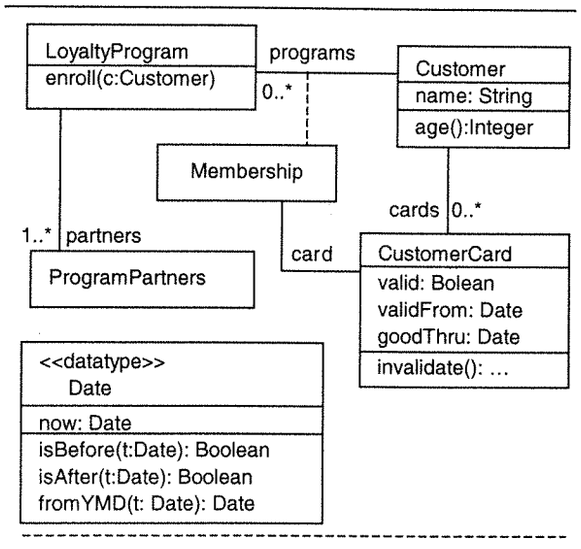

Fig. 4. Part of the Loyal and Royal class diagram of Warmer and Kleppe (2003).

Example 14:

self.customer of Example 13 specifies a *forAll* collection operation to express that all customer' ages of a LoyaltyProgram should be more than 30 years old.

- **Messages**
  OCL message expressions are used to specify the fact that an object has, or will send some message to another object at some moment in time (Kleppe and Warmer, 2000), (OMG, 2003b).

Example 15:

The following expression broadcasts a message called update to the observers of a subject.

context Subject:: haschanged()
post: observer ^ update(?:Integer, ?:Integer)

- **User-Defined DataType**
  "A data type is a special kind of classifier, similar to a class, whose instances are pure values (not objects). Usually, a data type is used for

specification of the type of an attribute. A data type is denotated using the rectangle symbol with keyword ≪dataType≫ or, when it is referenced by e.g. an attribute, denoted by a string containing the name of the data type." (OMG, 2003a)

Example 16:

A user-defined data type called Date is included in the class diagram of Fig. 4. The expression of the Example 13 uses it and access to its property (the *fromYMD* property).

4. Metrics Definition for OCL Expressions

Using the GQM (Basili and Rombach, 1998; Van Solingen and Berghout, 1999) template for goal definition, the goal pursued for the definition of the metrics for OCL expression is:

Analyze	*OCL expression structure related to cognitive techniques*
for the purpose of	*Evaluating*
with respect to	*Their Understandability*
from the point of view of	*the OO Software modellers*
in the context of	*OO software organisations*

Although our objective is to evaluate OCL expression understandability we are conscious that understandability is an external quality attribute and therefore it is influenced by structural properties of the OCL expressions. Therefore, we will focus the metrics on the structural properties of OCL expressions, and afterwards will ascertain through experimentation if these metrics could be used as early understandability indicators.

As Fenton and Pfleeger (1997) suggest that it is not advisable to define a single measure for capturing different structural properties, we will define a set of metrics, each of which captures different structural properties of an OCL expression related to a specific cognitive technique, considering those groups of elements presented in the previous section.

In order to define valid and reliable metrics we have applied a method based on (Calero et al., 2001; Cantone and Donzelli, 2000), which is composed of many steps beginning with the definition of the metrics goals

and finishing with the acceptance of these metrics in real projects. Even though all the steps are equally important, in this chapter we only address the definition of the metrics goals, the definition of the metrics and their theoretical validation. It is advisable to perform the theoretical validation of the metrics before the empirical validation. In the context of an empirical study, the theoretical validation of metrics demonstrates their construct validity, i.e. it "proves" that they are valid measures to be used as variables in the empirical study. The rest of the steps will be tackled in future studies.

4.1. *Metrics for "chunking" (group 1)*

In this section, we present the first set of metrics for OCL expressions considering those elements which involve "chunking" grouped as "group 1" in Table 1. For each metric, we provide its proper definition, its goals and an example to illustrate its calculus.

- **Number of OCL KeyWords (NKW)**
  DEFINITION: This metric counts the total number of OCL keywords used in an expression. The OCL keywords are: *and*, *attr*, *context*, *def*, *else*, *endif*, *endpackage*, *if*, *implies*, *in*, *inv*, *let*, *not*, *oper*, *or*, *package*, *post*, *pre*, *then*, and *xor*. Each occurrence of these keywords will increment by one the value of NKW, with the exception of the groups:

  – *if*, *then*, *else* and *endif* keywords,
  – *package* and *endpackage* keywords,
  – *let* and *in* keywords

  that will be counted as only one keyword, because they are used together to represent a single subexpression.
  GOAL: The number of keywords is an indicator of the complexity of an OCL expression, in terms of its size. A higher number of keywords used in an OCL expression the greater its complexity.

  Example 17:

  In the Expression of Example 5 the value of NKW is 3, the keywords used are: *context*, *inv*, *if*, *then*, *else*, and *endif*.

- **Number of Explicit *Self* (NES)**
  DEFINITION: This metric counts the number of times *self* is used in an explicit form in an OCL expression.
  GOAL: *Self*, as explained in Sec. 3.1, provides a point of reference for the interpretation of an OCL expression. By using it in an explicit or implicit form it is possible to access different properties (attributes, operations, and associations-end). The greater the number of times *self* is used may indicate the greater the difficulty of the context to be understood.

  Example 18:

  The expression of Example 3 contains a navigation of a reflexive association written as *self.work_with.exists(self);* in this example the first *self* could be written in implicit form but the last *self* must be explicitly declared because *self* is sent as a parameter. The value of NSE is 2, as *self* was used in explicit form twice.

- **Number of Implicit *Self* (NIS)**
  DEFINITION: This metric counts the number of times *Self* is used in an implicit form in an expression.
  GOAL: The goal of NES is also valid for the NIS metric; however, as *self* (and iterator variables in operation collections) can be left implicit, the number of times *self* is left implicit introduces a difficulty for modellers, because they have to evaluate to which object a property is applied. This evaluation will interrupt the process of chunking an expression.

  Example 19:

  In the following expression, the value of NIS is 1, *self* is implicit when the property *work_with* is referred.

  context Person inv:
  not work_with.exists(self)

- **Number of Variables defined by *Let* expressions (NVL)**
  DEFINITION: This metric counts the total number of variables defined by *Let* expressions in an expression. This metric does not take into account the number of times a variable is reused, but the quantity of different defined variables.

GOAL: NVL metric is related to the degree of reuse of the variables within an OCL expression. Although a low use of *let* expressions can improve the readability of an OCL expression, we believe that a higher number of variables defined through *let* can indicate that the OCL expression is reasonably complex.

Example 20:

In Example 4, a variable called *income* is defined to represent the expression: *self.job.salary−>sum()*. The value of NVL is 1, as there is only one variable defined through a *let*-expression.

- **Number of *If* Expressions (NIE)**
  DEFINITION: This metric counts the total number of *if* expressions used in an expression.
  GOAL: A high number of *if*-expressions can increase the complexity of an OCL expression, in terms of conditional situations to be understood.

Example 21:

The value of NIE in the expression of Example 4 is 1, there is only one if-expression specified.

- **Number of *Set*, *OrderedSet*, *Bags*, *Sequence* or *Tuple* literals**
  DEFINITION: The total number of *set*, *orderedset*, *bags*, *sequence* or *tuple* literals used in an OCL expression are considered respectively by NSL, NOSL, NBL, NSQL, NTL metrics.
  GOAL: A high number of collection literals used in an expression could reduce its simplicity.

Example 22:

The value of NSQL is 1 in the following expression, as a sequence literal is specified:

= (s :Sequence(T)):Boolean
post: result = Sequence{1 ...self−>size()}−>
forAll(index :Integer |self−>at(index) = s−>at(index)) and
self−>size() = s−>size()

- **Number of Boolean Operators (NBO)**
  DEFINITION: This metric counts the total number of boolean operators used in an expression.
  GOAL: We believe that the number of boolean operators is an indicator of the complexity of an OCL expression, in the same way as the number of keywords used in it. In Warmer and Kleppe (1999, 2003) is also recommended to split a constraint with many boolean *and* operator, as a correct style for writing less complex (and easier to read and write) expressions. Those expressions with a high number of boolean operators can be candidates to be evaluated in order to be rewritten.

Example 23:

NBO = 2 in the following expression:

context ProgramPartner inv:
partners.deliveredServices−>forAll(pointsEarned = 0) and
membership.card−>forAll(goodThru = Date::fromYMD(2000,1,1))
and customer−>forAll(age() >55)

NBO = 2 because two *and* operators are used in the expression. This expression taken from Warmer and Kleppe (2003), is an example of an invariant that can be rewritten splitting it into three different invariants. Each of the new invariants will be composed of an operand of the *and* logical operator.

- **Number of Comparison Operators (NCO)**
  DEFINITION: This metric counts the number of times an operator like: $<, \leq, >, \geq, =$ y $<>$[5] is used in an expression. If an operator is used many times the metrics take into account each occurrence of it.
  GOAL: It is common to use a comparison operator as a way of expressing a constraint. The goal is similar to that of the previous metric.

[5] Some operators are overloaded = is defined for OclAny, OclModelElement, OclType and OclState.

Example 24:

The value of NCO metric is 4 in the following expression:

context Person inv:
age() > 30 and
Person.allinstances()−>forAll(p1, p2 | p1 <>p2
implies p1.dni <> p2.dni) and
work_with−>size() ≤ 5

- **Number of Explicit/Implicit Iterator variables (NEI, NII)**
DEFINITION: These metrics count the total number of iterator variables specified in explicit or implicit form respectively. The way to determine the values of the NEI and NII metrics is similar to NSE and NSI metrics. NEI is the number of times an iterator variable appears in an expression (except in the way it is declared), whilst the way to compute the value of the NII metric involves the evaluation of each property with an implicit object in order to determine the object to which the property applies. If the property belongs to an object represented by an iterator variable, the metric is incremented by one. In OMG (2003b) there is a clear example of the resolution of ambiguities for an implicit object.
GOAL: As already mentioned, the use of implicit objects and the determination of which object a property is applied to, leads to the interruption of the understandability of the expression as a chunk and could also reduce its understandability, as it is not always easy to know immediately which is the target object.

Example 25:

Given the class diagram of Fig. 5 and the following expression:

context Person inv:
self.employer−>forAll (iter1 | iter1.employee−>
exists (lastname = name))

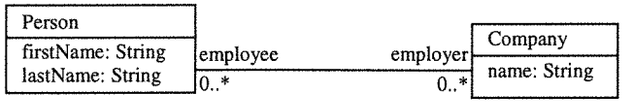

Fig. 5. An example of class diagram.

The value of NEI is 1 due to *iter1* is an explicit iterator variable for the *forAll* operation (*iter1* is an explicit variable whose type is Company), and it is used in an explicit form when iter1.employee is used.
The value of NII is 2 due to:

- *exists* operation does not have an explicit iterator variable (an iterator variable whose type is Person), and *lastname* refers to this implicit variable.
- The *name* attribute is a property of *self* and *iter1*, and this constitutes an ambiguity. To determine to which object *name* is applied, the most inner scope is used. The result is: *name* attribute refers to *iter1*.

4.2. *Metrics for "chunking" (group 2)*

In this section, we define a set of metrics for OCL expressions considering those elements which involve "chunking" grouped as "group 2" in Table 1. For each metric, we provide its proper definition, its goals and an example to illustrate its calculus.

- **Number of Attributes belonging to the classifier that *Self* represents (NAS)**
DEFINITION: This metric counts the total number of attributes belonging to the classifier that *Self* represents. The attributes are directly referred using the notation *self.attributename.*
GOAL: A higher number of this kind of attributes will increase the complexity of the expression. The comprehension of attributes used in an expression not only involves the meaning of them as a constituent of a class diagram but also the different OCL expressions that declare restriction on them.

Example 26:

In Example 5, two attributes of Person are used, *title* and *isMale*, the former has an implicit *self* instance, while in the latter it is explicit, thus the value of NAS is 2.

- **Number of Operations belonging to the classifier that *Self* represents (NOS)**
  DEFINITION: This metric counts the total number of Operations belonging to the classifier that *self* represents. These operations are directly referred using the notation *self.operationname.*
  GOAL: The same goal as NAS but considering operations instead of attributes. The comprehension of an operation involves the comprehension of its meaning as a class diagram constituent and also the pre and postconditions associated to it.

  Example 27:

  The value of NOS in the OCL expression of Example 8 is 1, only one operation is used: age().

- **Number of Variables defined through ≪Definition≫ constraints**
  DEFINITION: This metric counts the total number of variables used in an expression which are defined through ≪definition≫ constraints. If a variable is reused many times in an expression the variable is counted only once.
  GOAL: A high number of variables reused in an expression can interrupt the meaning of the expression itself.

  Example 28:

  The value of NVD in Example 9 is 1 as the *income* variable is used in the expression and this variable has been defined through a ≪definition≫ constraint.

- **Number of oclIsTypeOf, oclIsKindOf or oclAsType Operations (NIO)**
  DEFINITION: This metric counts the number of times an oclIsTypeOf, oclIsKindOf or oclAsType operation is used in an expression (NIO). These are some predefined properties. We are conscious that NIO does not only use the type represented by *self*, which identifies the metrics of group 2, but also uses other types connected by *self* through inheritance (a characteristic of group 1). Despite this, we prefer to leave this metrics in this group. This is an example where "chunking" and "tracing" are performed sinergically.

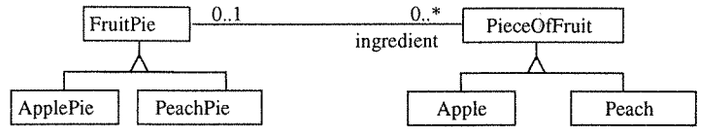

Fig. 6. Example for illustrating NIO.

GOAL: A high number of this kind of predefined operations can increase the complexity of the expression, as the modelers have to deal with inheritance concepts. The complexity will also depend on the complexity of the inheritance tree in which the classifier represented by *self* is included.

Example 29:

Given the class diagram of Fig. 6 and the following expression of the *ApplePie* class:

context ApplePie inv:
self.ingredient−>forAll(oclIsKindOf(Apple))

The value of NIO is 1, because the expression uses oclIsKindOf().

- **Number of properties postfixed by @Pre (N@P)**
  DEFINITION: This metric represents the number of different properties postfixed by @pre. This metric can be used exclusively for postconditions.
  GOAL: A high number of variables postfixed by @pre could increase the complexity of an OCL expression.

  Example 30:

  In the expression of Example 12 the value of N@P is 1, as the postfix @pre is used with the *usage* property.

4.3. *Metrics for "tracing"*

In this section, we propose a set of metrics for OCL expressions considering those elements which involve "tracing" (see Table 1). The definition of each metric is accompanied by the goals and an example to illustrate its calculus.

- **Number of Navigated Relationships (NNR)**
  DEFINITION: This metric counts the total number of relationships that are navigated in an expression. If a relationship is navigated twice, for example using different properties of a class or interface, this relationship is counted only once. Whenever an *association class* is navigated we will consider the association to which the association class is attached.
  GOAL: As Warmer and Kleppe (2003) remark: An "argument against complex navigation expressions is that writing, reading and understanding invariants becomes very difficult". The meaning of each relationship involves the understanding of how the objects are coupled to each other. The larger the set of relationships to be navigated, the greater is the context to be understood.

Example 31:

In the following expression (valid for Fig. 4 and defined for the LoyaltyProgram, class) two different relationships are navigated: (1) the relationship between LoyaltyProgram and Customer (it is navigated from LoyaltyProgram to Membership, and from LoyaltyProgram to Customer), (2) the relationship between Membership and CustomerCard.

context LoyaltyProgram inv:
membership.card −> forAll (
goodThru = Date::fromYMD (2007,1, 1)
and self.customer−>forAll (age()>30)

thus NNR = 2 because two relationships were navigated.

- **Number of Attributes referred through Navigations (NAN)**
  DEFINITION: This metric counts the total number of attributes referred through navigations in an expression.
  GOAL: NAN measures the extent of usage of attributes of other classes by the class where the expression is defined. The larger the set of attributes referred through navigations, the greater is the context to be understood. The understanding of attributes belonging to other classes involves the comprehension of them as a chunk, i.e. their meaning and the OCL specification associated to them.

Table 3. WNON metric definition.

$\sum(1 + \lvert Par(m)\rvert)(1 + R + \lvert P_{out,\ in/out}(m)\rvert)\ m \in N(expression)$
N(expression): Set of different[6] operations referred through navigations.
$\lvert Par(m)\rvert$: quantity of actual parameters of m
R stands for the m operation result and represents the value of 1.
$\lvert P_{out,\ in/out}(m)\rvert$: quantity of *out* and *in/out* parameters of the m operation. Through a navigation it is possible to exclusively access one result as a time (OMG, 2003b), this weighted formula giving a higher weight to those operations used many times to access different results.

Example 32:

In the following expression, valid for Fig. 4 and defined for the LoyaltyProgram class, only the *goodThru* attribute is used, thus NAN = 1:

context LoyaltyProgram inv:
membership.card −> forAll (goodThru = Date::fromYMD (2007,1, 1)) and self.customer−>forAll (age()>30)

- **Weighted Number of referred Operations through Navigations (WNON)**
  DEFINITION: The metric is defined as the sum of weighted operations (operations which are referred through navigations). For that reason we must consider the operation calls. The operations are weighted by the number of actual parameters (only the values of all *in* or *in/out* parameters are necessary to specify in the operation call (OMG, 2003b)) and the number of *out* parameters used (also considering the return type of the operation). The definition of the WNON metric is defined as it is shown in Table 3.
  GOAL: WNON measures the extent of usage of operations of other classes by the class where the expression is defined. The larger the set of operations (referred through navigations) and its parameters, the greater is the context to be understood.

[6]We will consider the actual parameter of the operation call in order to analyse if two operations (referred through navigations) are different elements in the N(expression)-set. Having two operation calls they will be different if they have different operation names or different actual parameters (their parameter names or/and the quantity of actual parameter are not equal).

Example 33:

The following operation "income" has a result of type Integer, an *in* parameter (d) and an *out* parameter (bonus):

context Person::income(d: Date, bonus: Integer): Integer
post: result = type { bonus = ...,
result = ... }

Now, consider an expression in which we navigate to the Person class, and we operate with the two returned values of income:

context Salary::calculate()
post: person.income(aDate).bonus + person.income(aDate).result

Applying the metric WNON to the postcondition expression of the *calculate* operations we obtain: WNON = $(1 + 1)(1 + 1 + 1) = 6$.

- **Number of Navigated Classes (NNC)**
DEFINITION: This metric counts the total number of classes, association classes or interfaces to which an expression navigates. If a class contains a reflexive relation and an expression navigates it, the class will be considered only once in the metric. Also, as a class might be reachable from a starting class/interface from different forms of navigations (i.e. following different relationships) we must consider this situation as a special case: If a class is used in two (or more) different navigations the class is counted only once .
GOAL: Warmer and Kleppe (1999) argue that "any navigation that traverses the whole class model creates a coupling between the object involved". A high number of navigated classes will increase the coupling between the objects.

Example 34:

In the expression of Example 32, the value of NNC = 3, because the classes Membership, Customer and CustomerCard are used.

- **Weighted Number of Messages (WNM)**
DEFINITION: This metric counts the total number of messages defined in an expression weighted by its actual parameters. The weighted operation is carried out according to Table 4.

Table 4. WNM metric definition.

$\sum(1 + \lvert Par(m)\rvert)$ $m \in$ M(expression)
M(expression): Set of different operations[7] used through messaging in an expression.
$\lvert Par(m)\rvert$: quantity of actual parameter of the m operation.

GOAL: The modeller should consider that a communication has taken place (OMG, 2003b) whenever a message is specified in an OCL expression. A high number of operations called — through messaging — could reduce the understandability of the expression. The understanding of each message involves the understanding of its parameters and its semantics (OCL expressions associated with them).

Example 35:

If we apply WNM metric to the following expression (an expression valid for the class diagram of Fig. 4) the value of WNM is 1, because there is only one message, without parameters, in the expression.

context CustomerCard inv:
validFrom.isBefore(goodThru) or
goodThru.isAfter(Date::now) implies self ^ invalidate()

- **Number of Parameters whose Types are classes defined in a class diagram (NPT)**
DEFINITION: This metric is specially used in pre and postcondition expressions and it counts the method parameters, and the return type (also called result) used in an expression, each parameter/result having a type representing a class or interface defined in the class diagram.
GOAL: In an object oriented system a typical method of communication is by using an object as a parameter (Gamma et al., 1995). Parameters can be used in the specification of an OCL constraint. However if the quantity of parameters whose types are classes in the class diagram is high, the context of the object involved will affect the understanding of the OCL expression.

[7]In order to analyse if two messages are different we will consider the object to which the message is sent and its operation call as we describe in WNO metric.

Example 36:

In the following expressions (both, pre- and post-conditions, are valid expressions for the LoyaltyProgram class of Fig. 4), the value of NPT = 1 because only one parameter (c), whose type is a class in the class diagram (Customer), is used in the expression.

LoyaltyProgram::enroll(c: Customer)
pre: not customer−>includes(c)
post: customer = customer @ pre−> including (c)

- **Number of User-Defined Data Type Attributes (NUDTA)**
  DEFINITION: This metric counts the total NUMBER of attributes belonging to a user-defined data type used in an expression. Attributes are counted once if they belong to the data type class, even if they are used more than once.
  GOAL: NUDTA is a measure of the potential reuse of user-defined data type attributes.

Example 37:

In the expression of Example 35, the value of NUDTA = 1 because only one class attribute (*now*) of a data type (*Date*) is used.

- **Number of User-Defined Data Type Operations (NUDTO)**
  DEFINITION: The definition of this metric is analogous to the NUDTA metric, but considering operations instead of attributes.
  GOAL: NUDTO is a measure of the potential reuse of user-defined data type operations.

Example 38:

In the expression of Example 35, NUDTO = 2 because the *isBefore* and *isAfter* operations (belonging to the data type Date) are used.

- **Weighted Number of Navigations (WNN)**
  DEFINITION: As we explain in the Sec. 3.2.3 an operation collection is composed of an expression which is evaluated for each collection element, and if the evaluated expression involves a new navigation (or many) we will give a higher weight to the new navigation used inside the

definition of the outermost expression. As the collection operation can be defined in terms of a new navigation and its collection operations, i.e. in a recursive way, we will refer to the different compositions of navigation as "level". In the case that navigation B is used in the immediate definition of an operation collection for navigation A, we would say that B is in level 2 and A in level 1.

The weight associated with each level is equal to the level number. Therefore the definition of the WNN metric is:

$$\text{WNN} = \sum \text{weight of the level} * \text{number of navigations of the level.}$$

GOAL: This metric is an estimate of overall coupling among objects (those involved through relationships) in the specification of an OCL expression. The value of WNN will provide an indicator of how the relationships are used together for specifying semantics of an expression in terms of a coupling set of objects. A high number of WNN will indicate an intertwining specification of relationships and this could reduce the understandability of an OCL expression.

Example 39:

In the precondition expression of Example 36, the value of WNN is 1, and there is only one navigation (self.customer). Now, we will show how the WNN is obtained in the following expression:

context LoyaltyProgram inv:
self.customer −>forAll(age() <= 30) and
self.customer −>forAll (c1 | self.customer −>
forAll (c2| c1 <> c2 implies c1.name <> c2.name)

Two subexpressions are connected by an *and* operator. Each subexpression involves navigations. Whilst the navigation of the first subexpression does not include a new navigation in its evaluation, the second one uses a collection operation defined in terms of another, and the value of WNN is obtained in the following way: $\mathbf{1} * 2 + \mathbf{2} * 1 = 4$.

The number shown in bold print font represents the applied weight, and the number shown in normal font indicates the number of navigations.

- **Depth of Navigations (DN)**
  DEFINITION: Given that in an OCL expression there can be many navigations regarding its definition, we build a tree of navigation using the class name to which we navigates. We will only consider navigations starting from the contextual instance (from *self*). The root of the tree is the class name which *self* represents. Then we build a branch for each navigation, where each class we navigate to is a node in the branch. Nodes are connected by "navigation relations". DN is defined as the maximum depth of the tree.

  When a navigation includes a collection operation expression defined in terms of a new navigation(s), we will build a new tree for the navigation used in the collection operation expression, using the same method, then we will connect both trees using a "definition connection". A dashed line will represents a definition connection. When we obtain the depth of the tree, we will apply the following rule: "Navigation connection is counted once, and definition connection twice".
  GOAL: A high depth of navigations may involve a complicated navigation. Warmer et al. (2003) suggest avoiding complex navigation expressions, they also argue that: "using long navigation makes details of distant objects known to the object where we started the navigation".

  This metric was proposed as a measure of class complexity and design complexity. It is based on the idea that a high value of the metrics will be an indicator of how distant are the objects known by the Classifier (where the expression is defined).

Example 40:

A tree built for the expression of Example 32, using the method described above, is shown in Fig. 7(a). In this example the value of DN is 2.

Example 41:

According to the expression of Example 39, the tree built is shown in Fig. 7(b), where a dashed line represents a definition connection. The DN value for the expression of Fig. 7(b) is equal to 4.

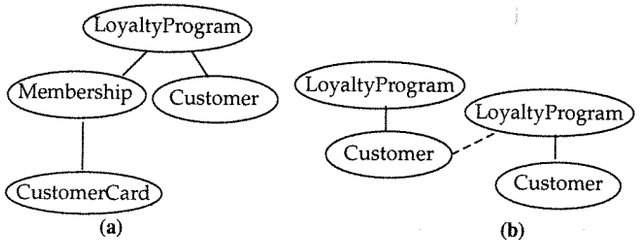

Fig. 7. Examples of navigation.

- **Weighted Number of Collection Operations (WNCO)**
  DEFINITION: The collection operations used in the expression definition are weighted according to the level in which they are defined, so the metric is defined thus:

$$\text{WNCO} = \sum \text{weight of the level} * \text{number of collection operations of the level.}$$

  GOAL: The value of WCO will provide an indicator of how the operation collections are specified using a sort of composition. A high number of WNCO will indicate an intertwining specification of operation collections and this could reduce the understandability of an OCL expression.

Example 42:

In the expression of Example 39 WNCO $= 4$, and the value is obtained in the following way: $1 * 2 + 2 * 1 = 4$. The number shown in bold font represents the weight, and the number shown in normal font indicates the number of operation collections. Three operation collections are used in the specification of the expression, two of them are used in the same subexpression at different levels.

5. Theoretical Validation of the Proposed Metrics

To develop the theoretical validation of metrics we have used the property-based framework of Briand et al. (1996) and (1997) and its adaptation for interaction-based metrics for coupling and cohesion (Briand et al., 1999).

The framework and its adaptation, before being used, are explained in Sec. 5.1. Then they are applied for the validation of the metrics defined in Sec. 4. We will only show one example of the theoretical validation for each kind of measure proposed.

5.1. *Briand et al.'s frameworks*

Property-based approaches (also called axiomatic approaches) such as the Briand et al.'s frameworks, formally define desirable properties of the measures for a given software attribute. These properties are properties of the numerical relation system of measures. They aim to formalise the empirical properties that a generic attribute of software or a system (e.g. the length or size) must satisfy in order for it to be used in the analysis of any measurement proposed for that attribute. Property-based approaches propose a measure property set that is necessary but not sufficient (Briand et al., 1996; Poels and Dedene, 2000). They can be used as a filter to reject proposed measures (Kitchenham and Stell, 1997), but they are not sufficient to prove the validity of the measure. The two best known approaches were proposed by Weyuker (1988) and Briand et al. (1996).

5.1.1. *The original framework of Briand et al. (1996)*

Briand et al. (1996) have provided a set of mathematical properties that characterise and formalise several important measurement concepts such as size, length, complexity, cohesion and coupling, related to internal software attributes. After some criticisms made by Poels and Dedene (1997), Briand et al. (1997) made some modifications to the definition of the properties they had initially proposed. Hereafter, we refer to the final framework, i.e. the modified version. This framework is based on a graph-theoretic model of a software artifact, which is seen as a set of elements linked by relationships. The idea is to characterise the properties for measurement of a given software attribute via a set of mathematical properties, based on this graph-theoretic model.

The properties they provide are generally enough to be applied not only to code, but also to other artifacts produced during the software process, for example for OCL expression metrics.

- **Systems, Modules and Modular Systems.** In this framework, a system is characterised by its elements and the relationships between them. The authors want the properties they define to be as independent as possible of any product abstraction. Thus, the framework does not reduce the number of possible system representations, as elements and relationships can be defined according to the needs.

  A software artifact is modelled by a graph $S = \langle E, R \rangle$, called system, where E is the set of elements of S, and R is a binary relation among the elements of $E (R \subseteq E \times E)$. From this point, we say that m is a module of S if and only if $E_m \subseteq E$, $R_m \subseteq E_m \times E_m$ and $R_m \subseteq R$. A module is connected to the rest of the system by external relationships, whose set is defined as $\text{Outer}R(m) = \{\langle e_1, e_2 \rangle | (e_1 \in E_m \wedge e_2 \notin E_m) \vee (e_1 \notin E_m \wedge e_2 \in E_m)\}$. A modular system is one where all the elements of the system have been partitioned into different modules. Therefore the modules of a modular system do not share elements, but there may be relationships across modules. Figure 8 shows a modular system with three modules m_1, m_2 and m_3.

  We will now introduce inclusion, union, intersection operations for modules and the definitions of empty and disjoint modules.
- **Inclusion.** Module $m_i = \langle E_{mi}, R_{mi} \rangle$ is said to be included in module $m_j = \langle E_{mj}, R_{mj} \rangle$ (notation: $m_i \subseteq m_j$) if $E_{mi} \subseteq E_{mj}$ and $R_{mi} \subseteq R_{mj}$.

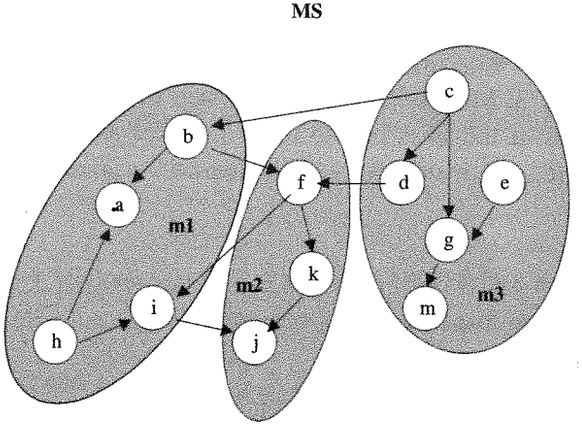

Fig. 8. A modular system.

- **Union.** The union of modules $m_i = \langle E_{mi}, R_{mi}\rangle$ and $m_j = \langle E_{mj}, R_{mj}\rangle$ (notation: $m_i \cup m_j$) is the module $\langle E_{mi} \cup E_{mj}, R_{mi} \cup R_{mj}\rangle$.
- **Intersection.** The intersection of modules $m_i = \langle E_{mi}, R_{mi}\rangle$ and $m_j = \langle E_{mj}, R_{mj}\rangle$ (notation: $m_i \cap m_j$) is the module $\langle E_{mi} \cap E_{mj}, R_{mi} \cap R_{mj}\rangle$.
- **Empty module.** Module $\langle \emptyset, \emptyset\rangle$ (denoted by $\emptyset$) is the empty module.
- **Disjoint modules.** Modules m_i and m_j are said to be disjoint if $m_i \cap m_j = \emptyset$.
- Since in this framework modules are just subsystems, all systems can theoretically be decomposed into modules. The definition of a module for a particular measure in a specific context is just a matter of convenience.

5.1.1.1. Properties for size and length

We will describe in this section only the properties for the internal attributes size and length, as only these two properties were applied in the theoretical validation. Both properties may be defined for entire systems or modules of entire systems.

- **Size.** The basic idea is that size depends on the elements of the system. The size of a system $S = \langle E, R\rangle$ is a function Size(S) that is characterised by the following properties:
  - Property 1. Nonnegativity. Size(S) ≥ 0
  - Property 2. Null value. The size of a system S is null if E is empty:

$$E = \emptyset \Rightarrow \text{Size(S)} = 0$$

  - Property 3. Module additivity. The size of a system S is equal to the sum of the sizes of two of its modules $m_1 = \langle E_m1, R_m1\rangle$ and $m_2 = \langle E_{m2}, R_{m2}\rangle$ such that any element of S is an element of either m_1 or m_2:

$$(m_1 \subseteq S \text{ and } m_2 \subseteq S \text{ and } E = E_{m1} \cup E_{m2} \text{ and } E_{m1} \cap E_{m2} = \emptyset) \Rightarrow \text{Size}(S) = \text{Size}(m_1) + \text{Size}(m_2)$$

- **Length.** The length of a system $S = \langle E, R\rangle$ is a function Length (S) characterised by the following properties:
  - Property 1. Nonnegativity. Length (S) ≥ 0

  - Property 2. Null value. The length of a system S is null if E is empty:

$$E = \emptyset \Rightarrow \text{Length}(S) = 0$$

  - Property 3. Nonincreasing monotonicity for connected components. Let S be a system and m be a module of S such that m is represented by a connected component of the graph representing S. Adding relationships between elements of m does not increase the length of S:

$$(S = \langle E, R\rangle \text{ and } m = \langle E_m, R_m\rangle ym \subseteq Sym \text{ "is a connected component of } S\text{" and } S' = \langle E, R'\rangle \text{ and } R' = R \cup \{\langle e_1, e_2\rangle\} \text{ and } \langle e_1, e_2\rangle \notin R \text{ and } e_1 \in E_{m1}, \text{ and } e_2 \in E_{m1}) \quad \text{Length}(S) \geq \text{Length}(S')$$

  - Property 4. Nondecreasing monotonicity for non-connected components. Let S be a system and m_1 and m_2 be two modules of S such that m_1 and m_2 are represented by two separate connected components of the graph representing S. Adding relationships from elements of m_1 to elements of m_2 does not decrease the length of S.

$$(S = \langle E, R\rangle \text{ and } m_1 = \langle E_{m1}, R_{m1}\rangle ym_2 = \langle E_{m2}, R_{m2}\rangle \text{ and } m_1 \subseteq S \text{ and } m_2 \subseteq S \text{ "are separate connected components of } S\text{" and } S' = \langle E, R'\rangle \text{ and } R' = R \cup \{\langle e_1, e_2\rangle\} \text{ and } \langle e_1, e_2\rangle \notin R \text{ and } e_1 \in E_{m1}, \text{ and } e_2 \in E_{m2}) \quad \text{Length}(S') \geq \text{Length}(S)$$

  - Property 5. Disjoint modules. The length of a system S made of two disjoint modules m_1, m_2 is equal to the maximum of the lengths of m_1 and m_2.

$$(S = m_1 \cup m_2 \text{ and } m_1 \cap m_2 = \emptyset \text{ and } E = E_{m1} \cup E_{m2}) \quad \text{Length}(S) = \max\{\text{Length}(m_1), \text{Length}(m_2)\}$$

5.1.2. *An adaptation of Briand et al.'s framework for interaction-based metrics for coupling*

In the Briand et al.'s framework adaptation for interaction-based metrics (Briand et al., 1999) coupling is defined as a relation between an individual software part and its associated software system, rather than as a relation between two software parts. The interactions described in Briand et al. (1999) are directly related with the definition of a *high level design*. They use two kinds of interactions: *Interaction between data* (if a piece of data appears in the definition of other) and *interaction between data and function* (if a piece of data appears in the definition of a function). Most of the metrics defined for OCL for tracing techniques are interaction-based metrics, and for that reason in order to use a similar approach to Briand et al. (1999) we need to define what we consider a high level design, which kind of interaction we will use and how coupling is defined in terms of the interaction:

- *High level design*: A UML class diagram with the specification of OCL expression defining invariants, pre- and post-condition of operations (which only declares the effect of the operation but not how the operation is performed (Warmer and Kleppe, 1999)) will be considered a high level design.
- *Relation*: The relations are defined between a software individual part — in our context, an OCL expression — and its associated software system — mainly, attributes and operations which it is possible to access through messaging, navigations, etc.
- *DU-interaction/OU-interaction*: The interaction from Data declaration (or Operation declaration) to data Used (or operation used) in an OCL expression.
- *Import coupling*: Given a software part *sp*, import coupling of *sp* is the number of DU- or OU-interaction between data declaration (or operation declaration) external to *sp* and data used (or operation used) within *sp*.

We have only defined metrics related to import coupling.[8] Our hypothesis is similar to the ISP-hypothesis of Briand et al. (1999): The larger the

[8]The extent to which a software part depends on the rest of the software system.

number of imported software parts, the larger the context to be understood, the more likely the occurrence of a fault.

5.2. *NAN properties as coupling (interaction-based) metric*

We will make some definitions prior to the application of properties of interaction-based metrics for coupling to the NAN (the number of attributes referred through navigations in an expression) metric:

- Relation: The relations are defined between a software individual part- in our context, an OCL expression) and its associated software system (attributes which it is possible to access through navigations in the NAN metric).
- DU-interaction: The interaction from Data declaration to Data Used (attributes used through navigations) in an OCL expression.
- Import coupling: Given a software part *sp* (an OCL expression), the import coupling of *sp* is the number of DU-interactions between data declaration external to *sp* and data used within *sp*.

Our hypothesis is similar to the ISP-hypothesis of Briand et al. (1999): The larger the number of "used" software parts, the larger the context to be understood, the more likely the occurrence of a fault.

Following a similar approach applied in Briand et al. (1999) the properties for interaction-based measures for coupling are instantiations, for our specific OCL context, of the properties defined in Briand et al. (1996) and (1997) for coupling.

- Nonnegativity: Is directly proven, and it is impossible to obtain a negative value. An expression *sp* without navigation (referring to attributes) in its definition has $\mathrm{NAN}(sp) = 0$.
- Monotonicity: Is directly verified, adding import interactions — in this case, DU-interactions of navigations referring to attributes — to an OCL expression cannot decrease its import coupling. If we add a new navigation referring to an attribute in an expression *sp*, two possible situations can happen: (1) the attribute referred to in the added navigation is an attribute already used by a DU-interaction. Thus the metric NAN applied to the new expression obtained, is equal to $\mathrm{NAN}(sp)$. (2) If the

added navigation refers to a new attribute, then NAN applied to the new expression is greater than NAN(*sp*).
- Merging of Modules: This property can be expressed for our context in the following way: "the sum of the import coupling of two modules is no less than the coupling of the module which is composed of the data used of the two modules". The value of NAN for an expression which consists of the union of two original expressions, is equal to the NAN of each expression merged when the sets of attributes referred to in each original expression are disjointed, otherwise it is less than NAN of each expression merged.

In a similar way, it is possible to show that NNR, WNON, NNC, WNM, NPT, NUDTA, NUDTO, NAS, NOS, N@P and NIO are interaction-based measures for coupling.

5.3. *WNM as a size metric*

For our purpose and in accordance with the framework of Briand et al. (1996) and (1997), we consider that an OCL expression is a system composed of OCL messages (elements) and relationships are represented by the relation "belong to", which reflects that a message belongs to an OCL expression. A sub-expression will be considered a module. We will demonstrate that WNM fulfils all of the axioms that characterize size metrics, as follows:

- Nonnegativity: Is directly proven and it is impossible to obtain a negative value.
- Null value: An expression e without a message, has a $\text{WNM}(e) = 0$.
- Module Additivity: If we consider that an OCL expression is composed of modules with no message in common, the number of messages of an OCL expression will always be the sum of the number of messages of its modules. In other words, when two modules (sub-expression) without messages in common are merged then the new expression has as many messages as each of the merged expressions. But if the original merged modules (sub-expressions) have some message in common, then the WNM of the resulting expression should be less than adding the WNM of the original expressions. In a similar way, it is possible to show that WNN, WCO are size metrics.

In a similar way, it is possible to show that WNN, WNCO, NKW, NES, NIS, NIE, NSL, NOSL, NBL, NSQL, NTL, NBO, NCO, NEI, NII, NVL and NVD are size metrics.

5.4. *DN as a length metric*

For our purpose and in accordance with the framework of Briand et al. (1996) and (1997), we consider that an OCL expression is a system. The elements are the classes (to which the expression navigates) and the relationships are the navigations of a UML relationship. We will demonstrate that DN fulfils all of the axioms that characterise length metrics, as follows:

- Nonnegativity and Null Value are straightforwardly satisfied, the depth of a tree can never be negative, and an expression without navigation has an empty tree, and DN is 0.
- Nonincreasing monotonocity for connected components: If we add relationships between elements of a tree (classes or interfaces) the depth does not vary.
- Nondecreasing monotonocity for non-connected components: Adding a relationship to two unconnected components (two trees) makes them connected, and its length is not less than the length of the two unconnected components.
- Disjoint modules: The depth of a tree is given by the component that has more levels from the root to the leaves.

6. Conclusions and Future Work

This chapter gives the definition of a set of metrics for OCL expressions to measure structural properties of OCL expressions, considering OCL concepts related to the cognitive techniques of "tracing" and "chunking".

After performing the theoretical validation of the proposed metrics using the original framework of Briand et al. (1996; 1997) and its adaptation

Table 5. Theoretical validation of metrics according to Briand et al. (1996), (1997) and (1999).

Metric classification	OCL expression metrics defined in terms of cognitive techniques					
	Chunking (group 1)	Chunking (group 2)		Tracing		
	NKW, NES, NIS, NVL, NIE, NSL, NOSL, NBL, NSQL, NTL, NBO, NCO, NEI, NII	NAS, NOS, NIO, N@P	NVD	NNR, NAN, WNON, NNC, WNM, NPT, NUDTA, NUDTO	WNM WNN WNCO	DN
Interaction-based metrics for coupling		X		X		
Length						X
Size	X		X		X	

to interaction-based metrics (Briand et al., 1999), the results are summarised in Table 5.

As Table 5 shows, most of the measures obtained for chunking techniques are size metrics, and most of the metrics defined for tracing techniques are interaction-based metrics for coupling. This correlation between a cognitive technique and some kinds of measures for internal attributes is significant, and the reason is because of the proper definition of the cognitive technique. In fact, Klemola (2000) argues that "some traditional complexity metrics can be supported by the fact they are clearly related to cognitive limitations".

We believe that measures obtained for tracing technique will be more important than measures obtained for chunking technique, as the former cognitive technique has been observed as a fundamental activity in program comprehension (Boehm-Davis, 1996; Klemola, 2000).

We are aware that it is necessary to provide comprehensive information about this set of metrics in order to perform the empirical validation of them. As many authors mentioned (Basili et al., 1998; Fenton and

Pfleeger, 1997; Kitchenham et al., 1995; Schneidewind, 1992) empirical validation employing experiments or case studies is fundamental to assure that the metrics are really significant and useful in practice. We are currently planning a controlled experiment for corroborating if the proposed metrics could be useful as early indicators of the OCL expression understandability. Moreover, once we have empirically validated these metrics at an expression level, we will be able to extend them at a class level.

Acknowledgements

This work has been partially funded by the MUML project financed by "University of Castilla-La Mancha" (011.100623), the network VII-J-RITOS2 financed by CYTED, and the UNComa 04/E048 project financed by "Subsecretaría de Investigación de la Universidad Nacional del Comahue". Luis Reynoso enjoys a postgraduate grant from the agreement between the Government of Neuquen Province (Argentina) and YPF-Repsol.

References

Basili, V. R. and Rombach, H. (1998). The TAME Project: Towards Improvement-Oriented Software Environments. *IEEE Transactions on Software Engineering*, Vol. 14, No. 6, pp. 758–773.

Basili, V., Shull, F. and Lanubile, F. (1999). Building Knowledge Through Families of Experiments. *IEEE Transactions on Software Engineering*, Vol. 25, No. 4, pp. 435–437.

Boehm-Davis, D. A., Fox, J. E. and Philips, B. (1996). Techniques for Exploring Program Comprehension. *Empirical Studies of Programmers, Sixth Workshop*. Eds. Gray W. and Boehm-Davis D. Norwood, NJ. Ablex, pp. 3–21.

Briand, L. C., Morasca, S. and Basili, V. (1999). Defining and Validating Measures for Object-Based High Level Design. *IEEE Transactions on Software Engineering*, Vol. 25, No. 5, pp. 722–743.

Briand, L., Morasca, S. and Basili, V. (1996). Property-Based Software Engineering Measurement. *IEEE Transactions on Software Engineering*, Vol. 22, No. 1, pp. 68–86.

Briand, L. C., Morasca S. and Basili, V. (1997). Response to: Comments 'Property-Based Software Engineering Measurement': Refining the Additivity Properties. *IEEE Transactions on Software Engineering*, Vol. 22, No. 3, pp. 196–197.

Briand, L. and Wüst, J. (2001). Modeling Development Effort in Object-Oriented Systems Using Design Properties. *IEEE Transactions on Software Engineering*, Vol. 27, No. 11, pp. 963–986.

Brito e Abreu, F. and Melo, W. (1996). Evaluating the Impact of Object-Oriented Design on Software Quality. *Proceedings of the 3rd International Metric Symposium*, pp. 90–99.

Calero, C., Piattini, M. and Genero, M. (2001). Method for Obtaining Correct Metrics. *Proceedings of the 3rd International Conference on Enterprise and Information Systems (ICEIS'2001)*, pp. 779–784.

Cant, S. N., Henderson-Sellers, B. and Jeffery, D. R. (1994). Application of Cognitive Complexity Metrics to Object-Oriented Programs. *Journal of Object-Oriented Programming*, Vol. 7, No. 4, pp. 52–63.

Cant, S. N., Jeffery, D. R. and Henderson-Seller, B. (1992). A Conceptual Model of Cognitive Complexity of Elements of the Programming Process. *Information and Software Technology*, Vol. 7, pp. 351–362.

Cantone, G. and Donzelli, P. (2000). Production and Maintenance of Software Measurement Models. *Journal of Software Engineering and Knowledge Engineering*, Vol. 5, pp. 605–626.

Cook, S., Kleepe, A., Mitchell, R., Rumpe, B., Warmer, J. and Wills, A. (2001). *The Amsterdam Manifiesto on OCL*. Eds. Clark T. and Warmer J. Advances in Object Modelling with the OCL. *Lecture Notes in Computer Science 2263*, Springer, Berlin, pp. 115–149.

El-Eman, K. (2001). *Object-Oriented Metrics: A Review of Theory and Practice.* National Research Council Canada. Institute for Information Technology.

Fenton, N. and Pfleeger, S. (1997). *Software Metrics: A Rigorous and Practical Approach*. Chapman & Hall, London, 2nd Edition. International Thomson Publishing Inc.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995). *Design Patterns.* Addison Wesley. Massachusetts.

Gogolla, M. and Richters, M. (2001). *Expressing UML Class Diagrams properties with OCL*. Eds. Clark T. and Warmer J. Advances in Object Modelling with the OCL. *Lecture Notes in Computer Science 2263*, Springer, Berlin, pp. 85–114

Henderson-Sellers, B. (1996). *Object-Oriented Metrics: Measures of Complexity.* Prentice-Hall.

Hennicker, R., Hussmann, H. and Bidoit, M. (2001). *On the Precise Meaning of OCL Constraints*. Eds. Clark T. and Warmer J. Advances in Object Modelling with the OCL. *Lecture Notes in Computer Science 2263*, Springer, Berlin, pp. 69–84.

ISO/IEC 9126 (2001). *Software Product Evaluation-Quality Characteristics and Guidelines for their Use.* Geneva.

Kitchenham, B., Pflegger, S. and Fenton, N. (1995). Towards a Framework for Software Measurement Validation. *IEEE Transactions of Software Engineering*, Vol. 21, No. 12, pp. 929–943.

Kitchenham, B. and Stell, J. (1997). The Danger of Using Axioms in Software Metrics. *IEEE Proc.-Soft. Eng.*, Vol. 144, No. 5–6, pp. 279–285.

Klemola, T. (2000). A Cognitive Model for Complexity Metrics. *Proceedings of the 4th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*. Sophia Antipolis and Cannes, France.

Kleppe, A. and Warmer, J. (2000). *Extending OCL to Include Actions*. UML 2000, York, UK, October 2000, Proceedings, *Lecture Notes in Computer Science*, Springer, pp. 440–450.

Object Management Group. (2003a). *UML 2.0 Infrastructure Final Adopted Specification. OMG Document* ptc/03-09-15. [On-line] Available http://www.omg.org/
cgi-bin/doc?ptc/2003-09-15.

Object Management Group. (2003b). *UML 2.0 OCL 2nd revised submission. OMG Document ad/2003-01-07*. [On-line] Available: http://www.omg.org/ cgi-bin/doc?ad/2003-01-07.

Object Management Group. (2003c). *UML Specification Version 1.5, OMG Document formal/03-03-01*. [On-line] Available: http://www.omg.org/cgi-bin/doc?formal/03-03-01.

Poels, G. and Dedene, G. (1997). Comments on "Property-Based Software Engineering Measurement": Refining the Additivity Properties. *IEEE Transactions on Software Engineering*, Vol. 23, No. 3, pp. 190–195.

Poels, G. and Dedene, G. (2000). Distance-Based Software Measurement: Necessary and Sufficient Properties for Software Measures. *Information and Software Technology*, Vol. 42, No. 1, pp. 35–46.

Richters, M. (2002). *A Precise Approach to Validating UML Models and OCL Constraints*. Biss Monographs Vol. 14. (Series Editors) Gogolla, M., Kreowski, H. J., Krieg-Brückner, B., Peleska, J., Schlingloff, B.H. Logos Verlag. Berlin.

Schneidewind, N. (1992). Methodology For Validating Software Metrics. *IEEE Transactions on Software Engineering*, Vol. 18, No. 5, pp. 410–422.

Van Solingen, R. and Berghout, E. (1999). *The Goal/Question/Metric Method: A Practical Guide for Quality Improvement of Software Development*. McGraw-Hill.

Warmer, J. and Kleppe, A. (1999). *The Object Constraint Language. Precise Modeling with UML*. Object Technology Series. Addison-Wesley. Massachusetts.

Warmer, J. and Kleppe, A. (2003). *The Object Constraint Language. Second Edition. Getting Your Models Ready for MDA*. Object Technology Series. Addison-Wesley. Massachusetts.

Weyuker, E. (1988). Evaluating Software Complexity Measures. *IEEE Transactions Software Engineering*, Vol. 14, No. 9, pp.1357–1365.