Cross-Validation of a Component Metrics Suite

Miguel Goulão, Fernando Brito e Abreu

Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, Portugal
{miguel.goulao, fba}@di.fct.unl.pt

Abstract. This paper describes an independent validation study for a suite of reusability metrics for component based design published in the literature. The metrics under validation were originally proposed using a semi-formal notation, namely a combination of mathematical formulae with natural language descriptions for elementary parts of those formulae. They were then computed using proprietary tools. By contrast, we present a formalization for the metrics suite that combines the version 2.0 of the UML metamodel with OCL. This technique provides a formal, portable and executable definition of the metrics set that can be used to perform other cross-validations of the metrics suite. A prototype working environment to perform such cross-validation experiments, both with this and other metrics sets was developed and a metrics collection experiment using that environment is discussed.

1. Motivation

Software quality modeling involves not only the definition of adequate taxonomies of quality attributes, but also the establishment of methods to assess those attributes. This assessment may be performed both in a qualitative and in a quantitative way. In this paper, we are concerned with the latter. The quantitative assessment of quality attributes requires the usage of software metrics. Quality models can then be built based on a combination of such metrics, using general purpose statistical techniques, such as multivariate regression models.

Statistical models require validation before they can be adopted by a large community. This validation should cover:

- **Internal consistency** – The model is specified through a set of mathematical statements, whose validity should be checked for mathematical correctness. A set of inputs are collected from the system represented by the model, along with relevant information on the assumptions made about the system elements. The model allows computing a set of outputs representing the predicted behavior of the system being modeled. In an internally consistent model, the outputs are valid if the inputs are valid.
- **External consistency** – A model has external consistency if information collected from it is not contradicted by other information observed in practice. This relates to the applicability of the model, as it focuses on the extent to which the assumptions made in the model apply beyond the sample from which the model was delivered.

The credibility of component-based development (CBD) quality models and related metrics suites depends not only on their soundness, but also on the extent to which they are validated, both by their own proponents and by independent parties. The current state of practice of experimental software engineering in what concerns CBD quality models and metrics is still far from reaching this level of maturity. Some quality models and metrics have been proposed [1, 2], but they lack proper validation in order to be widely accepted by the CBD community.

Although more validation studies are required, several difficulties hamper this task, such as (i) the lack of available data for conducting case studies, (ii) problems with the interpretation of the models and metrics specifications and (iii) lack of supporting tools for data collection.

In this paper, we present a CBD metrics suite cross-validation effort. On the original metrics set proposal, its authors used semi-formally defined metrics and proprietary tools to collect them and support their validation effort. Here, we formalize the metrics' definition and present an experimental environment that combines publicly available tools with standard languages for representing the metrics specification, heuristics based on those metrics, and the analyzed software components specification.

Our formalization technique, developed within the QUASAR research group (http://ctp.di.fct.unl.pt/QUASAR/) uses a metamodel of the target domain – CBD modeling with the Unified Modeling Language (UML), in the present case – upon which metrics are defined using the Object Constraint Language (OCL). The advantages of this formalization technique are manifold:

i. it is not affected by subjectivity, since the underlying metamodel removes any concept imprecision, or fuzzy inter-relationship interpretation;
ii. it is formal, because OCL [3] offers the required mechanisms for granting precision in metrics definition;
iii. it is standard, since both the metamodel and the metrics specification are expressed using the UML standard, of which OCL is a part;
iv. it is understandable to software practitioners, because both class diagrams (used for representing the meta-model) and the OCL itself were conceived for simplicity, without sacrificing expressiveness;
v. it is generic, therefore creating the conditions either for replication of validation experiments with this metrics suite, or with others;
vi. it is easily automated, since with the upcoming adoption of the new UML 2.0 standard [4, 5] it is likely that many tool producers will provide direct support to the new UML metamodel, as well as to OCL, which is also part of the UML standard.

This paper is organized as follows. In section 2, we present some related work concerning independent validation of software metrics. In section 3, we present our technique for formally defining metrics upon the UML 2.0 metamodel, using OCL. The formal definition of Washizaki et al.'s metrics set is presented in section 4. Section 5 contains the formal definitions of a set of heuristics that helps interpreting the metrics. In section 6, a metrics collection experiment is described and discussed. The used metrics set is then analyzed for its weaknesses. Conclusions are presented in section 7, while further work is outlined in section 8.

2. Related Work

The idea of independent validation of quality models and metrics is not new. It was borrowed from other sciences, where it is common to cross-check model validity by conducting independent experiments. For instance, in the pharmaceutical industry, new drugs have to undergo independent testing before being approved for public use (see, for instance, the regulations concerning this process in the United States of America – http://www.fda.gov/).

In the realm of experimental software engineering, there are several examples of this sort of independent scrutiny, concerning software metrics suites. In [6], Harrison et al. validated the MOOD (Metrics for Object-Oriented Design) metrics set [7]. In [8], Basili et al. checked the adequacy of Chidamber and Kemerer's metrics suite [9] to predict class fault-proneness. The same metrics suite was assessed as a maintainability predictor in [10]. Metrics such as McCabe's cyclomatic complexity [11] have been extensively used both in academic and industrial settings and integrated in several commercial development tools.

However, the previous examples are the exception and not the rule. Most of the metrics proposed in the literature have gone through scarce validation, if any at all. This phenomenon is particularly true within the more recent quest for CBD metrics (e.g. [12-14]). One noteworthy exception is the proposal of Washizaki et al. [15], which has gone further along the way of validating the proposed models. Their validation relies on a sounder statistical analysis, rather than on anecdotal examples or "gut-feeling" hints on descriptive statistics collected from small samples (see [16] for an overview on current CBD metrics proposals).

No known cross-validation has been performed on any of the CBD metrics sets. Washizaki et al.'s metrics set is no exception, regarding this. This paper is expected to mitigate this problem and to open the road for other cross-validation studies (of this and other metrics sets) by sharing our methodological approach.

The formalization approach presented in this paper is an evolution of our previous work concerning the formalization of metrics for object-oriented (OO) design. The approach of using OCL to perform such formalization was introduced in [17], using the metamodel of the GOODLY (a Generic Object Oriented Design Language? Yes!) OO design language [18]. With the growing adoption of UML by the software industry and academia, the need to make metrics available to the common practitioner has led us to develop the FLAME (Formal Library for Aiding in Metrics Extraction) metrics extraction library in OCL, on top of the UML 1.x metamodel, upon which several metrics suites were formalized [19-22]. The work presented in this paper differs from our previous work in metrics formalization in the following ways:

- instead of version 1.x, we are now using the UML 2.0 metamodel as a basis for our experiments; this is a sensible choice, due to its extended expressiveness for specifying component based architectures, as shown in [23].
- the focus of the metrics being analyzed has shifted from OO design to CBD;
- the tool support became more independent from proprietary formats, with the adoption of XMI as the input format for our metrics collection tools.

3. Metrics formalization and collection technique

Our metrics formalization and collection technique relies on the specification of OCL clauses that allow us to navigate through a metamodel and collect the required information for metrics computation. Consider a very small subset of the UML 2.0 metamodel, depicted in Figure 1. From this metamodel extract, we can see, for instance, how components are linked to their operations (`ownedOperation`) and properties (`ownedAttribute`), as well as how operations may contain parameters (`ownedParameter`). Due to space constraints, and for the sake of readability, we omit a few model elements and attributes used in the formalization described in this paper, as they would imply a much denser class diagram.

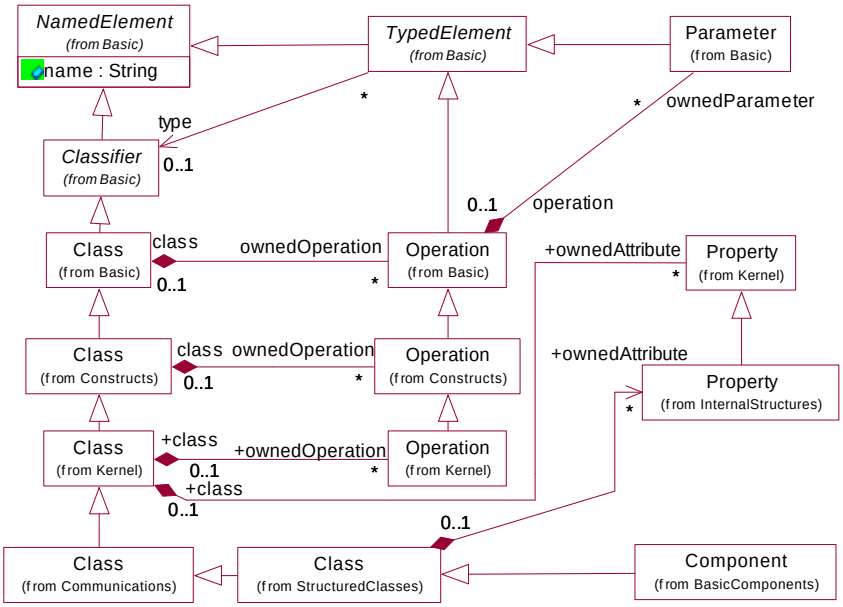

Figure 1 – UML 2.0 metamodel extract

In order to illustrate the metrics collection process, we now provide a small example. Consider the `SQL_Select` component, in Figure 2. This fine grained component allows us to show how metrics can be computed. It contains 9 attributes and 21 operations. Out of those 21 operations, 1 is a constructor, 5 are getters, 5 are setters, and the remaining 10 are business operations (all the operations with no stereotype, in the example). We use the term “business operations” to conform to Washizaki et al.’s classification for all the available operations that implement the functionality of components other than constructors, getters and setters.

The following OCL expressions compute the set of owned properties, its size, the set of owned operations, and its size, respectively. The two size expressions, in

particular, show how simple metrics can be computed for a given component. The results of the evaluation of each of the expressions are presented in ***italic***. For the sake of simplicity, we assume the objects that populate the metamodel to have the same name as the model elements they represent, in the following OCL clauses.

```
SQL_Select.ownedAttribute          = {NO_WORK, …, maxRows}
SQL_Select.ownedAttribute->size() = 9
SQL_Select.ownedOperation          = {SQL_Select, …, writeObject}
SQL_Select.ownedOperation->size() = 21
```

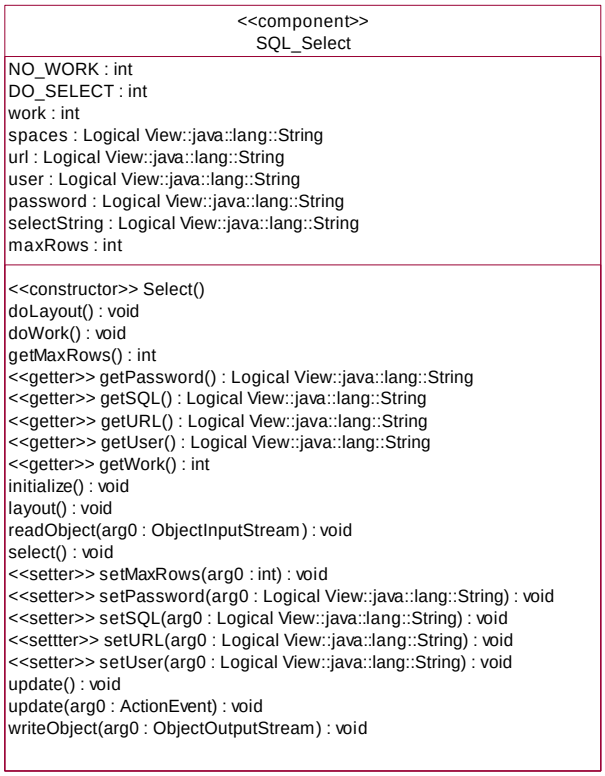

Figure 2 – The select component

Furthermore, we can define clauses within a given context. The following OCL expressions define a set of clauses that we will use later in the paper as auxiliary clauses to specify a component metrics set. The first set of clauses is defined for the model element **Component**, while the second one is defined for **Operation**.

```
Component
-- Readable Properties
Pr(): Integer = self.ownedOperation->select(o: Operation|
                o.stereotype = 'getter')->size()
```

```
-- Writable Properties
Pw(): Integer = self.ownedOperation->select(o: Operation|
                o.stereotype = 'setter')->size()
-- Properties in the component
A(): Integer  = self.ownedAttribute->size()
-- Total number of constructors in the component
Co(): Integer = self.ownedOperation->select(o: Operation|
                o.stereotype = 'constructor')->size()
-- Business methods with no return value
Bv(): Integer = self.ownedOperation->select(o: Operation|
                (not (o.stereotype = 'constructor'))and
                (not (o.stereotype = 'getter')) and
                (not (o.stereotype = 'setter')))->
                     select(b: Operation|
                             b.ReturnType()= 'void')->size()
-- Business methods with no parameters
Bp(): Integer = … -- Similar to Bv(), but counting operations
                  -- with an empty parameter list.
-- Business methods (all bur getters, setters and constructors)
B() : Integer = self.ownedOperation->size()
              + self.Pr() + self.Pw() + self.Co()
```

Operation

```
-- Set of formal parameters (except return parameter)
Params(): Set(Parameter) = self.formalParameter->
                    select(fp: Parameter |
                            fp.direction <> #return)
-- Set of return parameters of an Operation
ReturnParams(): Set(Parameter) =
                 self.formalParameter->
                    select(fp: Parameter |
                            fp.direction = #return)
-- Return type of an Operation
ReturnType (): String =
                if (self.formalParameter ->
                        exists(direction = #return))
                then
                   if (self.ReturnParams()->
                           asSequence() ->first.type.isDefined)
                   then
                      self.ReturnParams()->asSequence()->
                          first.type.name
                else
                   'void'
                endif
           else
                'void'
       endif
```

We can then use these clauses to extract information about a given component. For instance, the number of constructors provided by c1 is given by:

```
c1.Co() = 1
```

4. Washizaki et al.'s metrics set formalization

4.1. Metrics set description

Washizaki et al. proposed a metrics set of 5 metrics for *JavaBeans* component reusability assessment in [15]: Existence of Meta-Information (EMI), Rate of Component Observability (RCO), Rate of Component Customizability (RCC), Self-Completeness of Components parameters (SCCp) and Self-Completeness of Components' return values (SCCr). For each of the metrics, the authors presented:

- their intent;
- their definitions, combining a mathematical and an informal formulation;
- a confidence interval [Lower Limit; Upper Limit] for each metric; if its value is outside this interval, the component is regarded as a potential source of future problems, badly designed, or prone to exhibit bad behavior;
- interpretation heuristics based on such confidence intervals.

The quality characteristics, factors and criteria that lead to the development of each of the metrics are presented in Figure 3.

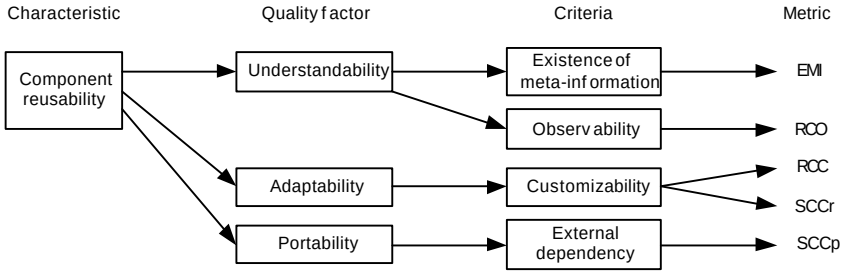

Figure 3 – Washizaki et al.'s quality model for reusability (adapted from [15])

In order to validate their approach, Washizaki et al. performed a case study where a sample of 125 JavaBeans from *jars.com* [24] was used, along with a prototype tool to extract the metrics from jar files. At *jars.com*, components are rated for quality with an 8 levels scoring system that employs expert opinions on the components. The criteria publicized by *jars.com* are presentation, functionality and originality. The rationale is that this rating can be used for component reusability assessment. This rating was used as a dependent variable in their validation experiment. They computed 95% confidence intervals for high quality components, based upon the average value of each metric, for components on the two top quality levels.

4.2. Metrics formalization

The first step for applying our technique is the formalization of the original metrics definitions with OCL, using the subset of the UML 2.0 metamodel that corresponds to component diagrams. A discussion on the formalization was presented in [25]. Only 4 out of the 5 metrics in Washizaki et al.'s set were formalized. The remaining metric

(EMI, valued 1 when a BeanInfo class exists for the component, or 0 otherwise) cannot be conveniently expressed upon the UML 2.0 metamodel, as we will discuss in section 4.3. Here, we present the metrics formalization for the purpose of self-containment of the paper. All the elementary OCL clauses were defined in section 3.

```
Component
-- Rate of Component Observability
RCO(): Real = if self.A() = 0 then
                 0.0
              else
                 self.Pr()/self.A()
              endif

-- Rate of Component Customizability
RCC(): Real = if self.A() = 0 then
                 0.0
              else
                 self.Pw()/self.A()
              endif

-- Self-Completeness of Component's return value
SCCr(): Real = if self.B() = 0 then
                  1.0
               else
                  self.Bv()/self.B()
               endif

-- Self-Completeness of Components Parameter
SCCp(): Real = if self.B() = 0 then
                  1.0
               else
                  self.Bp()/self.B()
               endif
```

4.3. On the formalization expressiveness

As seen on the previous section, we did not formalize the EMI metric. Our formalization technique relies on the expressiveness of the underlying metamodel, as well as on our ability to populate it from an existing specification (in this case, a *jar* archive). As long as a concept can be consistently and unambiguously represented by the meta-objects of the metamodel, it is possible to explore those meta-objects with appropriate OCL expressions and extract relevant information from them. In this particular case, detecting the `BeanInfo` class would require that we would parse the name of classes associated with our *JavaBean*, as our reverse engineering tools do not make any special annotations to express this. On the one hand, the UML metamodel itself does not include any special abstraction for this *JavaBean's* specific mechanism; on the other, OCL is not a convenient language to implement string parsing.

This is not an intrinsic limitation of the formalization technique. We could extend the metamodel to address this issue and improve the parsing mechanism to populate the extended metamodel with appropriate meta-objects. The trade-off would be to sacrifice the conformance to the standard meta-model, with its obvious negative implications to the portability of our formalization.

The formalization of metrics concerning extra-functional properties is also a challenge for the future. The UML 2.0 metamodel would have to be extended with convenient profiles in order for us to use our formalization approach.

5. Formalizing a suite of reusability heuristics

Along with their metrics set, Washizaki et al. also proposed a set of heuristics, to aid in the metrics interpretation. Three of the heuristics based on these metrics, referred to as `WarningRCO`, `WarningRCC` and `WarningSCCp`, behave as a band-pass filter, in the sense a potential problem warning should be issued if the metrics value is either lower than the lower threshold, or higher than the upper one (for `RCO`, `RCC` and `SCCp`, respectively). The `WarningSCCr` heuristic only establishes a minimum threshold for the value of the `SCCr` metric. If the metric value is below the threshold, this should be interpreted as an indication of a potential problem regarding the quality characteristic being assessed by that particular metric (portability). These heuristics can be formalized with OCL.

```
Class
AboveRange (limit: Real, value: Real): Boolean = value > limit
BelowRange (limit: Real, value: Real): Boolean = value < limit
OutOfRange (lowerLimit: Real, upperLimit: Real, value: Real):
Boolean =   (self.BelowRange (lowerLimit, value))
         or (self.AboveRange (upperLimit, value))
pre: lowerLimit < upperLimit

Component
WarningRCO(lowerThreshold: Real, upperThreshold: Real):
Boolean = self.OutOfRange (lowerThreshold, upperThreshold,
                           self.RCO())
WarningRCC(lowerThreshold: Real, upperThreshold: Real):
Boolean = self.OutOfRange (lowerThreshold, upperThreshold,
                           self.RCC())
WarningSCCr(lowerThreshold: Real): Boolean =
          self.BelowRange(threshold, self.SCCr())
WarningSCCp(lowerThreshold: Real, upperThreshold: Real):
Boolean =
   self.OutOfRange (lowerThreshold, upperThreshold, self.SCCp())
```

We defined the first three clauses in the **`Class`** context, rather than at the **`Component`** one. In the UML 2.0 metamodel, **`Component`** is a subclass of **`Class`**, so the clauses defined for **`Class`** can be used with **`Component`**. This allows us to reuse those clauses in the definition of heuristics based on other kinds of metrics based upon the UML metamodel (e.g. OO design metrics upon class diagrams).

Metric	Average	LowerLimit	Upper Limit	# components
RCO	0,40	0,17	0,42	36
RCC	0,35	0,17	0,34	35
SCCr	0,85	0,61	1,00	108
SCCp	0,74	0,42	0,77	28
EMI	0,84	0,50	1,00	105

Table 1 – Washizaki et al. heuristics thresholds

Table 1 summarizes the heuristics thresholds information provided by Washizaki et al.. For each metric, we present its acronym, the average value found in the metrics sample, described in 4.1, the lower and upper thresholds for the quality filters and the number of components in the sample which fulfill that criterion. For the sake of completeness, we include also the thresholds for `EMI`. All the metrics in this suite are defined as ratios and their maximum possible value is 1,00. For this reason, the predicate of `WarningSCCr` does not require an `upperThreshold`.

Finally the `DesignWarning` heuristic is defined as a simple combination of the previous ones. The arguments of the `DesignWarning` predicate allow calibrating each heuristic, as more data gets collected.

```
Component
DesignWarning(RCO_LL:  Real, RCO_UL:  Real,
              RCC_LL:  Real, RCC_UL:  Real,
              SCCp_LL: Real,
              SCCr_LL: Real, SCCp_UL: Real): Boolean =
                (self.WarningRCO(RCO_LL, RCO_UL))
              or (self.WarningRCC(RCC_LL, RCC_UL))
              or (self.WarningSCCr(SCCr_LL))
              or (self.WarningSCCp(SCCp_LL, SCCp_UL))
```

Checking if a given component (e.g. `c1`) violates any of these heuristics can then be achieved by evaluating the following OCL expression:

```
c1.DesignWarning (0.17, 0.42, 0.17, 0.34, 0.61, 0.42, 0.77)
```

6. Cross-validation experiment

6.1. Metrics collection

In order to test our formalization technique we conducted the following experiment: we collected Washizaki et al.'s metrics upon a public domain component library, the *FukaBeans* component library [26]. This library was developed according to the

JavaBeans component model [27] by Washizaki's research team. Each component is distributed as a separate *jar* file.

Table 2 contains the metrics values for each of the components in the library. In our analysis, we used reverse engineered models of those components, obtained from their *jar* files. The bold values represent data points where the heuristics proposed by Washizaki et al. trigger design warnings.

Although all the average values for the metrics are well inside the quality intervals suggested by Washizaki et al.'s experiments, only two of the components (GameBean and GraphBean) comply with all the quality heuristics. Seven out of twelve components fail three out of four heuristics. It may seem surprising that the components developed by the metrics set proponents fail to meet the structural quality standards proposed by themselves. It is worth mentioning that while Washizaki et al.'s model was calibrated with commercial *JavaBeans* components, this sample contains *JavaBeans* developed with academic purposes, with usually less than 10 methods. A possible interpretation for the apparent lack of reusability, with respect to Washizaki et al.'s quality model, is that, for such relatively small interfaces, the model is vulnerable: since all the metrics are defined as ratios, the small number of elements used in their computation leads to a high standard deviation of metrics values.

JavaBean	**RCO**	**RCC**	**SCCr**	**SCCp**
CellBean	**0,037**	**0,111**	0,909	**0,818**
FileUtil	**1,000**	**0,667**	1,000	**1,000**
FilterBean	0,267	**0,133**	0,933	**0,200**
FukaCalendarBean	**0,444**	**0,444**	0,857	0,571
FukaGraphBean	**1,000**	**1,000**	1,000	0,733
FukaStopWatchBean	**0,667**	**0,667**	1,000	**0,200**
FukaTextBean	**0,000**	**0,000**	1,000	**1,000**
GameBean	0,250	0,250	1,000	0,556
GraphBean	0,182	0,273	1,000	0,714
StatementBean	**0,667**	**0,667**	**0,500**	0,500
DocumentBean2	**0,000**	**0,000**	1,000	**1,000**
WordBean2	**0,000**	**0,000**	1,000	**1,000**
Mean	***0,376***	***0,351***	***0,933***	***0,691***
Standard deviation	***0,376***	***0,333***	***0,145***	***0,294***

Table 2 - Washizaki et al.'s metrics for the *FukaBeans* Library

6.2. Definition of the metrics set

Several issues can be raised regarding the definition of this metrics set. The original metrics definition is ambiguous in what concerns inheritance. It is unclear how inherited features (methods and attributes) should be accounted for. Our formalization only uses the directly defined features. While for this particular sample of components this is not a problematic issue, it is possible to define hierarchies of object-oriented components where this option would have an influence on the metrics values.

Another possible concern relates to the complexity associated with parameter types in the evaluation of the complexity of method interfaces. The metrics just count the number of parameters, thus being blind to parameter type repetition and parameter type complexity. For instance, a method with N parameters of distinct types is intuitively more complex than another method with N parameters of the same type. Also, arguments of atomic types (e.g. Integer, Real or Boolean) are intuitively less complex than ones of a composed type.

6.3. Applicability of the metrics set

Washizaki et al.'s metrics set was designed to assess reusability of fine grained components (JavaBeans) through the analysis of their interface complexity. This limits somewhat the scope of model elements being analyzed. UML architectural components have a much richer expressiveness than the one used in these metrics, which leaves out important model elements such as the provided and required interfaces, as well as the events the component may produce or consume. Furthermore, it does not address non-functional aspects of the components.

7. Conclusions

Independent validation of metrics is an essential step to foster adoption by a broader audience. The current CBD metrics state of the art is dominated by proposals that are insufficiently validated not only by their own authors, but mainly by third parties. Although we could argue that most CBD metrics proposals are recent, from our experience in the field for the past ten years, we believe that the most important factors hampering the replication of experiments are the ambiguity in metrics definition and the usage of inadequate specifying formalisms.

The proposal presented in this paper solves these two problems. It provides a formal, portable and executable way of specifying metrics for CBD, using standard notations such as UML 2.0 class diagrams and OCL. The emphasis on standard technologies aims to bring together the academic and practitioners communities, by providing a simple, but powerful solution to integrate metrics collection and heuristics-based help for CBD with the practitioners' normal development environments. This level of integration is a key factor to foster cross-validation of metrics proposals. Furthermore, using precise mechanisms based on standard notations to support the validation (and subsequently, the usage) of metrics makes

validation activities become testable and reproducible, two essential characteristics to promote them from a “craft” to a “science”.

8. Future work

Although the metrics formalized in this paper are centered in fine-grained components, the approach is flexible enough to be applied at different levels of granularity and with different concerns. In particular, we are interested in exploring metrics for component assemblies, rather than evaluating isolated components, with the objective of capturing the effect of the different components interaction in the overall quality of the component assembly.

The proposed approach is generic in the sense that, from a conceptual point of view, it is independent from the underlying metamodel. In particular, we expect to explore this technique for the evaluation of software product lines, using metrics sets such as the one proposed by [13]. In a parallel effort, we are exploring the definition of metrics for object-relational database schemas [28].

References

[1] M. Bertoa and A. Vallecillo, "Quality Attributes for COTS Components", 6th International Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE'2002), Málaga, Spain, 2002.

[2] N. S. Gill and P. S. Grover, "Component-Based Measurement: Few Useful Guidelines", *ACM SIGSOFT Software Engineering Notes*, vol. 28, 2003.

[3] OMG, "Unified Modeling Language: OCL (version 2.0)", Object Management Group Inc. ptc/03-08-08, August 2003.

[4] U2-Partners, "3rd revised submission to OMG RFP ad/00-09-01: Unified Modeling Language: Infrastructure - version 2.0", U2-Partners January 2003.

[5] OMG, "Unified Modeling Language: Superstructure - Version 2.0 - Final Adopted Specification", Object Management Group Inc. ptc/03-08-02, 2003.

[6] R. Harrison, S. J. Counsell, and R. V. Nithi, "An Evaluation of the MOOD Set of Object-Oriented Software Metrics", *IEEE Transactions on Software Engineering*, vol. 24, pp. 491-496, 1998.

[7] F. B. Abreu, "Candidate Metrics for Object-Oriented Software within a Taxonomy Framework", *Journal of Systems and Software*, vol. 26, pp. 87-96, 1994.

[8] V. Basili, L. Briand, and W. L. Melo, "A Validation of Object-Oriented Design Metrics as Quality Indicators", *IEEE Transactions on Software Engineering*, vol. 22, pp. 751-760, 1996.

[9] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object Oriented Design", *IEEE Transactions on Software Engineering*, vol. 20, pp. 476-493, 1994.

[10] W. Li and S. Henry, "Object-Oriented Metrics that Predict Maintainability", *Journal of Systems and Software*, vol. 23, pp. 111-122, 1993.
[11] T. McCabe, "A Complexity Measure", *IEEE Transactions on Software Engineering*, vol. 2, pp. 308-320, 1976.
[12] R. Dumke and A. Schmietendorf, "Possibilities of the Description and Evaluation of Software Components", *Metrics News*, vol. 5, 2000.
[13] A. v. d. Hoek, E. Dincel, and N. Medvidovic, "Using Service Utilization Metrics to Assess and Improve Product Line Architectures", Ninth International Software Metrics Symposium (Metrics'03), Sydney, Australia, 2003.
[14] M. A. S. Boxall and S. Araban, "Interface Metrics for Reusability Analysis of Components", Australian Software Engineering Conference (ASWEC'2004), Melbourne, Australia, 2004.
[15] H. Washizaki, H. Yamamoto, and Y. Fukazawa, "A Metrics Suite for Measuring Reusability of Software Components", Metrics'2003, 2003.
[16] M. Goulão and F. B. Abreu, "Software Components Evaluation: an Overview", 5ª Conferência da APSI, Lisbon, 2004.
[17] F. B. Abreu, "Using OCL to formalize object oriented metrics definitions", INESC Technical Report, Software Engineering Group ES007/2001, May 2001.
[18] F. B. Abreu, L. M. Ochoa, and M. A. Goulão, "The GOODLY Design Language for MOOD2 Metrics Collection", ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering, Lisboa, Portugal, 1999.
[19] A. L. Baroni, S. Braz, and F. B. Abreu, "Using OCL to Formalize Object-Oriented Design Metrics Definitions", QUAOOSE'2002, Malaga, Spain, 2002.
[20] A. L. Baroni, "Formal Definition of Object-Oriented Design Metrics": Vrije Universiteit Brussel - Belgium, in collaboration with Ecole des Mines de Nantes - France and Universidade Nova de Lisboa - Portugal, 2002.
[21] A. L. Baroni and F. B. Abreu, "Formalizing Object-Oriented Design Metrics upon the UML Meta-Model", Brazilian Symposium on Software Engineering, Gramado - RS, Brazil, 2002.
[22] A. L. Baroni and F. B. Abreu, "A Formal Library for Aiding Metrics Extraction", International Workshop on Object-Oriented Re-Engineering at ECOOP'2003, Darmstadt, Germany, 2003.
[23] M. Goulão and F. B. Abreu, "Bridging the gap between Acme and UML for CBD", Specification and Verification of Component-Based Systems (SAVCBS'2003), at the ESEC/FSE'2003, Helsinki, Finland, 2003.
[24] JARS, "http://www.jars.com/".
[25] M. Goulão and F. B. Abreu, "Formalizing Metrics for COTS", MPEC'2004, Edimburgh, 2004.
[26] Y. Fukazawa, H. Washizaki, H. Yamamoto, T. Adachi, Y. Sakai, K. Satoh, and D. Hoshi, "FukaBeans: JavaBeans Components Library, http://www.fuka.info.waseda.ac.jp/Project/CBSE/fukabeans/".
[27] G. Hamilton, "JavaBeans (version 1.01-A)", Sun Microsystems, API Specification August 1997.
[28] A. L. Baroni, C. Calero, F. Ruiz, and F. B. Abreu, "Formalizing Object-Relational Structural Metrics", 5ª Conferência da APSI, Lisbon, 2004.