Formal Definition of Metrics upon the CORBA Component Model

Miguel Goulão, Fernando Brito e Abreu
QUASAR Research Group, Departamento de Informática, FCT/UNL, Portugal
{miguel.goulao, fba}@di.fct.unl.pt

Abstract

Objective: *Formalization of metrics definitions to assess CORBA component assemblies' quality attributes.*

Method: *Representation of a component assembly as an instantiation of the CORBA Component Model metamodel. The resulting meta-object diagram can then be traversed using Object Constraint Language clauses. These clauses are a formal and executable definition of the metrics that can be used to assess quality attributes from the assembly and its constituent components.*

Results: *Demonstration of the expressiveness of our technique by formally defining metrics proposed informally by several authors on different aspects of components' and assemblies' quality attributes.*

Conclusion: *Providing a formal and executable definition of metrics for CORBA component assemblies is an enabling precondition to allow for independent scrutiny of such metrics which is, in turn, essential to increase practitioners confidence on predictable quality attributes.*

Keywords: *Software Metrics, CORBA Component Model, OCL, Component-Based Software Engineering*

1. Introduction

One of the goals of Component-Based Software Engineering (CBSE) is to achieve predictability of system quality based on the quality attributes of the constituent components [1]. Currently, developers are unable to make such predictions. Difficulties hampering this task include determining which properties would be useful to component developers and users, how the properties of individual components should be combined to predict the properties of assemblies, how they should be measured and how this information should be presented to component users. Current component models used in industry are not prediction-enabled, although this is an active topic of research [2, 3].

In current component models, the functional aspects of component wiring are supported, at least to a certain extent. Simple component models, such as JavaBeans [4] offer the ability to access components through their provided interfaces. Other, more sophisticated component models, such as the CORBA Component Model (CCM) [5], allow several input and output interfaces, support synchronous and asynchronous operations, as well as publishing and subscribing events.

The non-functional aspects of components are less supported than the functional ones by the aforementioned component models. For instance, the new UML 2.0 standard [6, 7] includes constructs for representing several of the above mentioned component wiring mechanisms, but not for the representation of non-functional properties. These are defined as a UML profile [8], an extension mechanism, but are not part of the core language. This may be a limitation in practice, since modeling tools are less likely to support UML extensions as part of their standard distribution.

Quality attributes can be classified according to a quality model. As noted by some authors [9, 10] specific quality models must be developed for CBSE. From a component user perspective, components are black-boxes whose evolution the user does not control. A component user is more concerned with the complexity involved in selecting and composing components.

Likewise, existing metrics for structured or object oriented development are not well suited for CBSE, since those were mainly concerned with internal complexity. Several traditional complexity metrics are useless to a component user, as their computation depends on having access to implementation details (e.g. McCabe's cyclomatic complexity [11]).

Due to the black-box nature of components and their specification state of practice, we are restricted to consider functional aspects when assessing their

quality attributes, either in isolation or in assembly. In a recent survey, we have identified several proposals for the quantitative assessment of components and assemblies, based upon their functional properties [12]. We have observed several recurrent problems in those metrics proposals, which are also common in metrics proposals for other purposes, such as OO design evaluation:

(1) **Lack of a quality framework** – occurs when metrics definition is not framed by a particular quality model.
(2) **Lack of an ontology** – occurs when the architectural concepts, either of functional or non-functional nature, to be quantified, are not clearly defined, namely in what regards their interrelationship. An ontology for modeling architectural concepts is called a metamodel.
(3) **Lack of an adequate formalism** – occurs when metrics are defined either with a formalism that requires a strong mathematical background, often not held by practitioners, or using natural language, which normally leads to subjective definitions that will jeopardize the correctness of metrics collection.
(4) **Lack of validation** – occurs when independent cross validation is not performed, mainly due to difficulties in experiment replication. Such validation is required before widespread acceptance is sought.

A solution to address (1) is to use the Goal-Question-Metric (GQM) approach [13], along with an appropriate quality model. In this paper, we present an approach to mitigate the remaining three problems. We use the CCM as a representation for components and component assemblies, due to its wide coverage of features provided by current component models used in industry, such as Enterprise Java Beans, COM or .Net. The CCM has a metamodel, upon which we define Object Constraint Language (OCL) expressions to specify and collect the metrics. Having a standard metamodel clearly defines the basic concepts being measured, thus solving problem (2). OCL expressions formally define how we measure such concepts. OCL combines formality with a syntax easily understood by practitioners familiar with OO development and is therefore an adequate formalism to help solve problem (3). Moreover, OCL expressions can be automatically evaluated. Since both OCL and the metamodel being used are Object Management Group (OMG) standards, the approach is inherently portable, thus making it suitable for independent replication in different settings. The combination of formality, with understandability and replicability is a facilitator to the independent scrutiny of metrics-based approaches to CBSE, therefore creating conditions to mitigate problem (4).

This paper is organized as follows: In section 2 we discuss some related work. In section 3, we briefly present the CCM, as well as its underlying metamodel. In section 4 we formalize metrics for CBSE upon the CCM metamodel. In section 5, we present a component assembly example and the metrics collected upon it. In section 6 we discuss our formalization technique within the framework of the problems identified in the introduction. Conclusions and further work are presented in section 7.

2. Related Work

Some proposals aim at establishing requisites and guidelines for CBD metrics, both concerning individual components [14] and component assemblies [15]. Although these proposals do not contribute with concrete metrics, they provide useful insight on the specificities to consider when developing metrics for CBD, mainly in what concerns the focus of such metrics. Several authors have contributed with proposals for the evaluation of component interfaces and dependencies [16-18]. These proposals focus on different aspects of the interfaces and dependencies of components and are mostly concerned with the complexity involved in understanding those interfaces, and reusing the components. Narasimhan and Hendradjaya proposed metrics to assess component integration density (a measure of its internal complexity) and interaction density (a measure of the complexity of relationships with other components) [19]. Hoek *et al.* proposed metrics to assess service utilization in component assemblies [20].

All of these proposals include, to some extent, informal specifications. While reading them, one has to make educated guesses to fill in the details that are left ambiguous. In this paper, we present a collection of metrics taken from some of these proposals, and formalized with OCL upon the CCM metamodel. Our contribution makes explicit our interpretation of the metrics definition and provides an executable specification for them. This facilitates independent validation efforts for these metrics.

In [21] we formalized a metrics set for component reusability assessment [18]. We used that formalization to conduct an independent validation experiment on the same metrics set in [22], using the UML 2.0 metamodel. Here, we will use the CCM metamodel, because the latter has more expressive power than UML 2.0 for representing components.

Our formalization has also been used in other contexts, therefore with different metamodels. In [23], we formalized well-known OO design metrics. This formalization was based on an OCL expressions library, named FLAME, aimed at helping metrics extraction in UML 1.x models [24]. More recently, we have successfully applied the same technique with object-relational database schema metrics [25].

3. CORBA components

3.1. The CORBA Component Model

The CCM [5] is the Object Management Group (OMG) standard for the specification of software components. As such, it is independent from a specific vendor, both in what concerns the component's programming languages and platforms. CORBA components are created and managed by homes (a home is a meta-type which offers standard factory and finder operations and is used to manage a component instance), run in containers that handle system services transparently and are hosted by generic application component servers. Each component may have several provided (facets) and required (receptacles) interfaces as well as the ability to publish (event sources) and subscribe (event sinks) events. Components also offer navigation and introspection capabilities. The CCM also has support for distribution and Quality of Service (QoS) properties. Overviews on the CCM can be found at [26, 27].

3.2. The CCM metamodel

The CCM specification includes a Meta Object Facility–compliant metamodel [28], where the CCM modeling elements are precisely defined. The metamodel includes three packages (Figure 1).

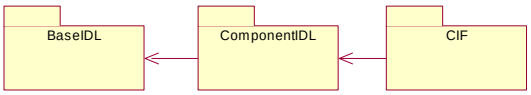

Figure 1 - The CCM metamodel packages

The `BaseIDL` package contains the modeling elements concerning the CORBA Interface Description Language (IDL) An excerpt of it is represented in Figure 3 (provided in Appendix B, for increased readability). `BaseIDL` is extended by `ComponentIDL`, to add the component specific constructs. Finally, `ComponentIDL` is extended by the `CIF` package, which includes the definitions relating to the component life cycle.

Figure 4 (also in Appendix B) represents an excerpt of the `ComponentIDL` package, where we can observe how the metamodel relates components with their required and provided interfaces, as well as with the events they publish or subscribe.

4. Metrics formalization with OCL

4.1. Formalization technique

A CCM assembly can be represented as an instance of the CCM metamodel. This instance can be seen as a directed graph of meta-objects (nodes) representing the modeling elements used in the assembly, and the appropriate meta-links (edges) among them. By traversing this graph, we can collect information on the assembly we want to analyze.

OCL expressions can be used to collect the relevant information from the meta-data (meta-objects and meta-links). Those expressions provide us the distilled information required for our metrics computation. Although expressions on specific meta-data could be written, our option is to define OCL functions at the meta-class level, thus making the expressions (and, therefore, the metrics definitions) reusable for all instances of the meta-class where they were defined.

OCL functions are defined in a given context. In our approach we use them in the context of a meta-class to facilitate information retrieval from instances of that meta-class. Consider the following example, where we define the functions `Operations` and `OperationsCount` in the context of the **`InterfaceDef`** meta-class, to represent the set of operations available in that interface and the number of elements in that set, respectively:

`InterfaceDef`

```
Operations(): Set(OperationDef) =
   self.contents->select(o |
     o.oclIsKindOf(OperationDef))->
     collect(oclAsType(OperationDef))->asSet()

OperationsCount(): Integer =
  self.Operations()->size()
```

To support our metrics definitions, we built a library of reusable functions, which includes these ones, described in Appendix A.

4.2. Formalizing metrics for CBD

In this section, we formalize several metrics for CBD proposed in the literature, using the library introduced in the previous section. For the sake of uniformity, we follow a similar pattern for each metric, or group of related metrics. We start by presenting their ***(i)*** *name and original specification*, keeping the notation used by their proponents (thus illustrating the variability of notations commonly used in metrics definitions), and the ***(ii)*** *metric's rationale*, in their proponents' view. These are followed by ***(iii)***

considerations and assumptions made during the formalization process and the *formalization of the metric in OCL*. This may include auxiliary functions. The `AuxilaryFuncion` typeface is used to identify these functions. The final expression of the metric is expressed within an outline bar.

4.2.1. Component interface complexity assessment

In this section we present the formalization of a selection of metrics concerning the complexity of component interfaces proposed by Boxall and Araban [16]. These metrics aim at assessing the understandability of a component interface.

Arguments per Procedure (APP)[16]

(i) The average number of arguments in publicly declared procedures (within the interface) was defined as in (Eq. 1),

$$APP = \frac{n_a}{n_p} \quad \text{(Eq. 1)}$$

where:

n_a = total count of arguments of the publicly declared procedures

n_p = total count of publicly declared procedures

(ii) The rationale for this metric is that humans have a limited capacity of receiving, processing and remembering information [29], so the number of chunks of information in the procedure definition (in this case, its arguments) should be limited. It is suggested that an increased number of arguments damages the interface's understandability.

(iii) Overloaded and overridden procedures (operations, in the CCM) are considered, but not inherited ones. The original metric specification makes no reference to the latter, so we assume them to be outside the scope of this metric. If the component is implemented in an OO language, all public and protected OO methods should be counted, but not the private ones.

The metric's definition assumes a single, or at least unified, interface for the component. There is no directly equivalent modelling element in the CCM metamodel. The component equivalent interface is broader, as it includes all implicit operations (a set of operatrions defined by component homes), operations and attributes which are inherited by the component (also through supported interfaces) and attributes defined inside the component. On the other hand, considering just a single interface as the context would lead to a different metric than the one proposed by Boxall and Araban. To be precise, we use the union of procedures in the provided interfaces as the set of procedures to be analyzed.

The context for the metric definition is **`ComponentDef`**. We start by defining `ProvidedOperations`, the set of operations used in the metrics definition, and `ProvidedOperationsCount`, the size of this set. The formalization of the **`APP`** metric becomes straightforward, with these auxiliary functions.

`ComponentDef`

```
ProvidedOperations(): Set(OperationDef) =
   self.ProvidesNoDups()->
   collect(Operations())->flatten()->asSet()

ProvidedOperationsCount(): Integer =
   self.ProvidedOperations()->size()

NA(): Integer =
   self.ProvidedOperations()->
     collect(ParametersCount())->sum()

NP(): Integer =
   self.ProvidedOperationsCount()
```

```
APP(): Real =
   self.NA()/self.NP()
```

Distinct Argument Count (DAC), and Distinct Arguments Ratio (DAR) [16]

(i) The number of distinct arguments in publicly declared procedures was defined as in (Eq. 2). Its percentage on the component interface was defined as in (Eq. 3),

$$DAC = |A| \quad \text{(Eq. 2)}$$

where:

A = set of the <name,type> pairs representing arguments in the publicly declared procedures

$|A|$ = number of elements in the set A.

$$DAR = \frac{DAC}{n_a} \quad \text{(Eq. 3)}$$

where:

n_a = total count of arguments of the publicly declared procedures

(ii) *DAC* is influenced by the adoption of a consistent naming convention for arguments in the operations provided by a component. If the same argument is passed over and over to the component's operations, the effort required for understanding it for the first time is saved in that argument's repetitions

throughout the interface. The smaller the number of distinct arguments a component user has to understand, the better. Likewise, a lower *DAR* leads to a higher understandability. Unlike *DAC*, *DAR* is immune to the size of the interface.

(iii) Boxall and Araban consider a parameter as a duplicate of another one if the pair `<name, type>` is equal in both arguments. `ExistsNameType` returns true if a duplicate of the parameter is found in a set of parameters. `DistinctArguments` returns the list of arguments used in the provided interfaces operation signatures, without duplicates. Finally, **`DAC`** computes the distinct arguments count and **`DAR`** their percentage in the component interface.

`ParameterDef`

```
ExistsNameType(s:Set(ParameterDef)): Boolean =
   s->exists((self.identifier = identifier)
         and (self.idlType = idlType))
```

`ComponentDef`

```
DistinctArguments(): Set(ParameterDef) =
   self.ProvidedOperations().Parameters()->
     iterate(p: ParameterDef;
             noDups: Set(ParameterDef) =
               oclEmpty(Set(ParameterDef)) |
       if (not (p.ExistsNameType(noDups)))
       then noDups->including(p)
       else noDups
       endif)
```

```
DAC(): Integer =
   self.DistinctArguments()->size()

DAR(): Real =
   self.DAC()/self.NA()
```

Argument Repetition Scale (ARS) [16]

(i) The *ARS* aims to account for the repetitiveness of arguments in a component's interface (Eq. 4).

$$ARS = \frac{\sum_{a \in A} |a|^2}{n_a} \quad \text{(Eq. 4)}$$

where:

A = set of name-type pairs in the interface

$|a|$ = count of procedures in which argument name-type a is used in the interface

n_a = argument count in the interface

(ii) Repetitiveness of arguments increases an interface's understandability.

(iii) We define two additional auxiliary functions `aCount` and `Sum_A`, which compute the count of procedures in which argument is used, and the sum of the squares of `aCount`.

`ComponentDef`

```
aCount(a: ParameterDef): Integer =
   self.ProvidedOperations()->
      select(o: OperationDef |
      a.ExistsNameType(o.Parameters()))->
         size()

Sum_A (): Integer =
   self.DistinctArguments()->collect(p|
      aCount(p)*aCount(p))->sum()
```

```
ARS(): Real = self.Sum_A()/self.NA()
```

4.2.2. Component packing density

The metric presented in this section was proposed by Narasimhan and Hendradjaya and aims at assessing the complexity of a component, with respect to the usage of a given mechanism [19].

Component Packing Density (CPD) [19]

(i) The *CPD* represents the average number of constituents of a given type in a component (Eq. 5),

$$CPD_{constituent\_type} = \frac{\#<constituent>}{\#components} \quad \text{(Eq. 5)}$$

where:

constituent_type = can be one of lines of code, operations, classes, modules and so on

#<constituent> = number of elements of *constituent_type* in the assembly

#components = number of components in the assembly

(ii) A higher density indicates a higher complexity of the component, thus requiring, as Narasimhan and Hendradjaya suggest, a more thorough impact analysis and risk assessment. CPD can be defined for a multitude of different constituents, but most of those suggested by Narasimhan and Hendradjaya are not available for users of black-box components. We will exemplify a possible formalization of this metric considering the number of operations in the provided interfaces as constituent, but other formalizations could be proposed similarly, for other constituents (e.g. interfaces). This metric can be formalized in OCL with the **`CPD`** function. To compute the number of operations made available by the component, we reuse the auxiliary function `ProvidedOperationsCount`, which we have formalized earlier.

ModuleDef

```
Components(): Set (ComponentDef) =
   self.contents->
   select(oclIsKindOf(ComponentDef))->
   collect(oclAsType(ComponentDef))->asSet()

ComponentsCount(): Integer =
   self.Components()->size()

ConstituentsCount(): Integer =
   self.Components()->
     collect(ProvidedOperationsCount())->
        sum()
```

```
CPD(): Real = self.ConstituentsCount()/
              self.ComponentsCount()
```

4.3. Increasing the coverage of the metrics set

The metrics formalized in this paper focus mainly on the provided interfaces of components. To increase the coverage of this metrics set, we include 3 extra metrics, based on simple counts provided by our metrics collection library, so that we can also asses the complexity of understanding the events emitted and consumed, as well as the one resulting from the configurability of each component. Since we are proposing these metrics ourselves, we provide the definition directly in OCL. Therefore, we only present their definition and Rationale.

Event Fan-In (EFI) and Event Fan-Out (EFO) and Configurable Properties Count (CPC)

(i) The EFI represents the number of Events emitted or published by a component. Conversely, the EFO represents the number of Events consumed by the component. CPC counts the number of configuration properties in each component. Their formal definition in OCL is as follows:

ComponentDef

```
EFI(): Integer =
   self.PublishesCount() + self.EmitsCount()

EFO(): Integer = self.ConsumesCount()

CPC(): Integer = self.PropertiesCount()
```

(ii) For **EFI** and **EFO**, the understandability of the component interaction with other components gets lower as the number of events gets higher. The same applies to **CPC**. More configurable properties imply a higher complexity in configuring a component, but they also increase the flexibility of its configuration.

5. Metrics collection example

5.1. The elevator control system example

Consider an elevator control system. Figure 2 depicts a component assembly with 4 components, `MotorsController`, `Alarm`, `ElevatorsController` and `RequestManager` that interact to implement it.

The `RequestManager` is responsible for handling the requests of the elevator users and sending adequate instructions to the `ElevatorsController` component, as well as handling any interactions with the `Alarm`. The `ElevatorsController` send orders to the `MotorsController`, and notifies the `RequestManager` of any change in the elevators' status. `MotorsController` controls the elevator's motion, ordering it to move up or down, at different speeds, and stop.

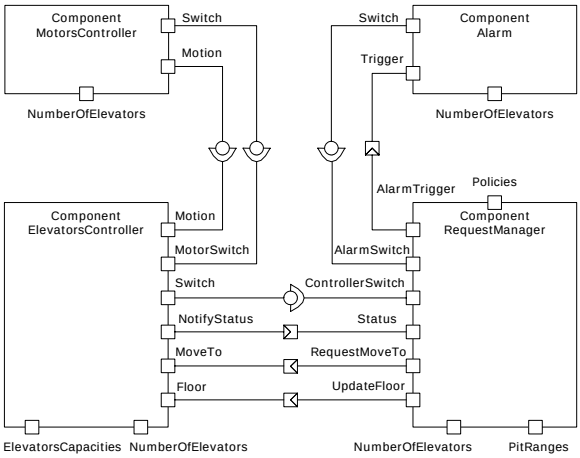

Figure 2 – The Elevator CCM assembly

The following IDL definitions complement the information on the assembly. For each interface we identify the facet(s) that provide it. For each event type we identify the event source that emits it. The components' configurable properties have the following types: `Policies` is of type `PolicyType`; `PitRanges` is of type `PitRangeSeq`; `Capacities` is of type `ShortSeq` and the several `NumberOfElevators` properties are of type `Short`.

```
enum StatusType {Stopped, Moving, Overload};

enum PolicyType {Closer, Direction};

struct PitRangeType {
   short lower;
   short upper;
}

typedef PitRangeSeq sequence <PitRangeType>;

typedef ShortSeq sequence <short>;
```

```
interface ISwitch {
   void On(in short motor);
   void Off(in short motor);
} // Used in the several Switch facets

interface IMotion {
   void Up(in short motor, in double speed);
   void Down(in short motor, in double speed);
   void NewSpeed(in short motor,
                 in double speed);
   void Stop(in short motor);
} // Used in the Motion facet

eventtype UpdateStatusEvent {
   public short elevator;
   public StatusType theStatus;
} // Used in the NotifyStatus event source

eventtype MoveRequestEvent {
   public short elevator;
   public short theFloor;
} // Used in the RequestMoveTo event source

eventtype UpdateFloorEvent {
   public short elevator;
   public short theFloor;
} // Used in the UpdateFloor event source

eventtype AlarmTriggerEvent {
   public short elevator;
} // Used in the AlarmTrigger event source
```

5.2. Metrics results

The formalized metrics were computed for the elevator example. Table I summarizes the metrics values for each of the components. Please note that as some of the metrics are computed as ratios, it is not possible to compute them when the denominator is 0. Those cases are written as N/A.

Table I - Metrics for the Elevator assembly

Context	APP	DAC	DAR	ARS	EFI	EFO	ATC
MotorsController	1,50	2	0,22	5,00	0	0	1
ElevatorsController	1,00	1	0,50	2,00	2	1	2
Alarm	1,00	1	0,50	2,00	1	0	1
RequestManager	N/A	0	N/A	N/A	1	3	3

The remaining metric, CPD, is computed for the whole component assembly. Its value is 2,50.

6. Discussion

6.1. Quality framework

Without a clear notion of the quality attribute we wish to assess and the criteria we will use to interpret the metrics values, it is not possible to interpret the metrics values. Although the authors of the proposed metrics provide a rationale for them, the lack of a well defined quality framework is noticeable.

When analyzing the values presented in Table I, based on the rationale presented during their formalization, one can only make relative judgments on their values. For instance, from the point of view of these metrics, the understandability of components `ElevatorsController` and `Alarm` is similar in what concerns their provided interfaces, but `ElevatorsController` emits and consumes more events, and has more configuration parameters. So, the overall interaction with this component is expected to be more complex than with the `Alarm` component. Further research is required before we can establish thresholds for any of the presented metrics.

6.2. Ontology

The lack of an adequate ontology in the original metrics definitions justifies our need to include several comments on the assumptions made before formalizing each metric (see section ***(ii)*** of all metrics formalizations). An ontology clarifies the used concepts and their interrelationships, providing a backbone upon which we can formalize the metrics definitions with OCL. The combination of the ontology with the OCL expressions removes the subjectivity from the metrics definitions. The ontology is also useful for the automation of metrics collection.

6.3. Specification formalism

We deliberately used the original formalisms in metrics definitions (see section *(i)* of all metrics formalization) to illustrate their diversity. For instance, the concept of collection size is conveyed with three different notations in (Eq. 1-5): a plain identifier (e.g. n_a), an identifier between a pair of '|' characters (e.g. |A|), and the # notation (e.g. #<constituents>). (Eq. 4) uses simultaneously two of the notations. This may lead to misinterpretations of the formulae.

Ambiguity resulting from the usage of natural language is also a problem. Suppose that rather than counting provided operations as constituents for the *CPD* metric, we would like to count provided interfaces. It is possible for different components to provide the same interface. In that case, should we count it once, or several times? If we use the informal version of the definition, we might just write *"constituent_type = provided interface"* and be left with an ambiguous definition. Now, consider the two following alternative `ConstituentsCount` function definitions:

ModuleDef

```
-- Constituents as Interfaces with duplicates
ConstituentsCount(): Integer =
   self.Components()->
     collect(ProvidesCount())->sum()

-- Constituents as interfaces w/out duplicates
ConstituentsCount(): Integer =
   self.Components()->
      collect(ProvidesNoDupsCount())->sum()
```

From the formal definition, it is clear that what we mean is “several times” in the first version and “once” on the second one, thus removing the ambiguity. A similar argument can be made for several of the metrics presented in this paper.

6.4. Validation

To the best of our knowledge, none of the metrics presented in this paper has undergone a thorough validation, so far. Due to the problems presented from sections 6.1. through 6.3., it should become clear that the ideal conditions for independent scrutiny of these metrics were not present in their original definitions. Several plausible interpretations can be provided to the definitions and this hampers experimental replicability.

6. Conclusions and further work

In this paper we explored the expressiveness of the CCM metamodel as a valuable ontology upon which we can formally define metrics for CBSE, using OCL expressions. We formally defined 5 metrics found in the literature, along with 3 new metrics, so that the resulting set covers most composition mechanisms used in the CCM.

We discussed our technique with respect to recurrent problems with metrics definitions (lack of a quality framework, lack of an ontology, inadequate specification formalism and insufficient validation) and how to mitigate them. Having a formal and executable definition of metrics for CORBA component assemblies is an enabling precondition to allow for independent scrutiny of such metrics, when combined with an adequate quality framework. While the provided metrics formalization is in itself a contribution to such an independent scrutiny, the formalization technique is amenable to the definition of new metrics, not only for CCM assemblies, but also for other component models and even other domains.

This paper is our second essay on the formalization of metrics sets for CBSE, proposed by other authors. Both the current and the previous [22] focused on metrics applicable to components in isolation. In a following paper we will present a similar formalization essay focused on component assemblies. While metrics of the first kind may somehow help component integrators in their selection process, the current components marketplace has not yet achieved the point where quasi-equivalent parts are available from multi-vendor parties as it is common in other engineering fields. Therefore, we believe that metrics for component assemblies, by allowing evaluating the resulting software architectures, will be much more useful in the short term. Among other things they will help in the evaluation and comparison of alternative design approaches, on the identification of cost effective improvements and on long term financial planning (total cost of ownership) by allowing to produce estimates on deployment and evolution costs.

References

[1] I. Crnkovic, H. Schmidt, J. A. Stafford, and K. Wallnau, "6th ICSE Workshop on Component-Based Software Engineering: Automated Reasoning and Prediction", *ACM SIGSOFT Software Engineering Notes*, vol. 29, 2004.

[2] K. Wallnau, "Volume III: A Technology for Predictable Assembly from Certifiable Components", Carnegie Mellon, Software Engineering Institute, Technical Report, CMU/SEI-2003-TR-009, April 2003.

[3] M. Larsson, "Predicting Quality Attributes in Component-based Software Systems", PhD Thesis, Department of Computer Science and Engineering, Mälardalen University, Västeras, Sweden, 2004.

[4] V. Matena and M. Hapner, "Enterprise JavaBeans Specification 1.1", Sun MicroSystems, Inc. 1999.

[5] OMG, "CORBA Components - Version 3.0", Object Management Group Inc., formal/02-06-65, June 2002.

[6] OMG, "UML 2.0 Infrastructure Final Adopted Specification", Object Management Group, Inc., ptc/03-09-15, September 2003.

[7] OMG, "UML 2.0 Superstructure Final Adopted Specification", Object Management Group Inc., ptc/03-08-02, August 2003.

[8] OMG, "UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanics", Object Management Group Inc., OMG Adopted Specification, ptc/04-09-01, September 2004.

[9] 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.

[10] R. P. S. Simão and A. D. Belchior, "Quality Characteristics for Software Components: Hierarchy and Quality Guides", in *Component-Based Software Quality: Methods and Techniques*, *LNCS 2693*, A. Cechich, M. Piattini, and A. Vallecillo, Eds.: Springer, 2003, pp. 184-206.

[11] T. McCabe, "A Complexity Measure", *IEEE TSE*, vol.2, pp.308-320, 1976.

[12] M. Goulão and F. B. Abreu, "Software Components Evaluation: an Overview", 5th APSI Conference, Lisbon, Portugal 2004.

[13] V. R. Basili, G. Caldiera, and D. H. Rombach, "Goal Question Metric Paradigm", in *Encyclopedia of Software*

Engineering, vol. 1, J. J. Marciniak, (Ed.) John Wiley & Sons, 1994, pp. 469-476.
[14] N. S. Gill and P. S. Grover, "Component-Based Measurement: Few Useful Guidelines", *ACM SIGSOFT Software Engineering Notes*, vol. 28, 2003.
[15] S. Sedigh-Ali, A. Ghafoor, and R. A. Paul, "Software Engineering Metrics for COTS-Based Systems", *IEEE Computer*, 2001.
[16] M. Boxall and S. Araban, "Interface Metrics for Reusability Analysis of Components", Australian Software Engineering Conference (ASWEC'2004), Australia, 2004.
[17] N. Gill and P. Grover, "Few Important Considerations for Deriving Interface Complexity Metric for Component-Based Software", *Software Engineering Notes*, vol. 29, 2004.
[18] H. Washizaki, H. Yamamoto, and Y. Fukazawa, "A Metrics Suite for Measuring Reusability of Software Components", 9th IEEE International Software Metrics Symposium (METRICS 2003), Sydney, Australia, 2003.
[19] V. L. Narasimhan and B. Hendradjaya, "A New Suite of Metrics for the Integration of Software Components", The First International Workshop on Object Systems and Software Architectures (WOSSA'2004), Australia, 2004.
[20] 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.
[21] M. Goulão and F. B. Abreu, "Formalizing Metrics for COTS", International Workshop on Models and Processess for the Evaluation of COTS Components (MPEC 2004) at ICSE 2004, Edimburgh, Scotland, 2004.
[22] M. Goulão and F. B. Abreu, "Cross-Validation of a Component Metrics Suite", IX Jornadas de Ingeniería del Software y Bases de Datos, Málaga, Spain, 2004.
[23] A. Baroni and F.B. Abreu, "Formalizing Object-Oriented Design Metrics upon the UML Meta-Model", Brazilian Symposium on Software Engineering, Brazil, 2002.
[24] A. Baroni and F. B. Abreu, "A Formal Library for Aiding Metrics Extraction", Int. Workshop on Object-Oriented Reengineering at ECOOP'2003, Germany, 2003.
[25] A. Baroni, C. Calero, F. Ruiz, and F. B. Abreu, "Formalizing Object-Relational Structural Metrics", 5th APSI Conference, Lisbon, Portugal, 2004.
[26] N. Wang, D. C. Schmidt, and C. O'Ryan, "Overview of the CORBA Component Model", in *Component-Based Software Engineering: Putting the Pieces Together*, G. T. Heineman and W. T. Councill, (Eds.), Addison-Wesley Publishing Company, 2001, pp. 557-571.
[27] J. Estublier and J.-M. Favre, "Component Models and Technology", in *Building Reliable Component-Based Software Systems*, I. Crnkovic and M. Larsson, (Eds.), Artech House, 2002, pp. 57-86.
[28] OMG, "Meta Object Facility (MOF) Specification (Version 1.4)", Object Management Group Inc., April, 2002.
[29] G. A. Miller, "The Magical Number Seven, Plus or Minus Two : Some limits in our Capacity for Processing Information", *Psychological Review*, vol.63, pp.81-97, 1956.

Appendix A – Auxiliary functions library

We only present the signatures of the functions of our library for metrics specification (the full version available at http://ctp.di.fct.unl.pt/QUASAR/).

Collection functions are useful when building up a more complex metric; counting functions increase the readability of other functions using them.

OperationDef

```
-- Operation's parameters
Parameters(): Set(ParameterDef)
ParametersCount(): Integer

-- Input parameters
InParameters (): Set(ParameterDef)
InParametersCount(): Integer

-- Output parameters
OutParameters(): Set(ParameterDef)
OutParametersCount(): Integer

-- Input/Output parameters
InOutParameters(): Set(ParameterDef)
InOutParametersCount(): Integer

-- Number of parameters with a different
-- <name,type> pair.
DistinctParametersTypeCount(): Integer
```

InterfaceDef

```
-- Operations available in the interface
Operations(): Set(OperationDef)
OperationsCount(): Integer
```

ComponentDef

```
-- Events emitted by the component
Emits(): Set (EmitsDef)
EmitsCount(): Integer

-- Events published by the component
Publishes(): Set(PublishesDef)
PublishesCount(): Integer

-- Events consumed by the component
Consumes(): Set(ConsumesDef)
ConsumesCount(): Integer

-- Facets of the component
Facets(): Set(ProvidesDef)
FacetsCount(): Integer

-- Configuration properties of the component
Properties(): Set(AttributeDef)
PropertiesCount(): Integer

-- Interfaces provided by the component
ProvidesNoDups(): Set(InterfaceDef)
ProvidesNoDupsCount(): Integer
Provides(): Bag(InterfaceDef)
ProvidesCount(): Integer

--Interfaces Required by the component
Receptacles(): Set(UsesDef)
ReceptaclesCount(): Integer
ReceptaclesNoDups(): Bag(InterfaceDef)
ReceptaclesNoDupsCount(): Integer
ReceptaclesInterfacesNoDups():
   Set(InterfaceDef)
ReceptaclesInterfacesNoDupsCount(): Integer

--Interfaces supported by the component
SupportsInterfacesNoDups(): Bag(InterfaceDef)
SupportsInterfacesNoDupsCount(): Integer
SupportsInterfaces(): Set(InterfaceDef)
SupportsInterfacesCount(): Integer
```

Appendix B – The CCM metamodel (excerpt)

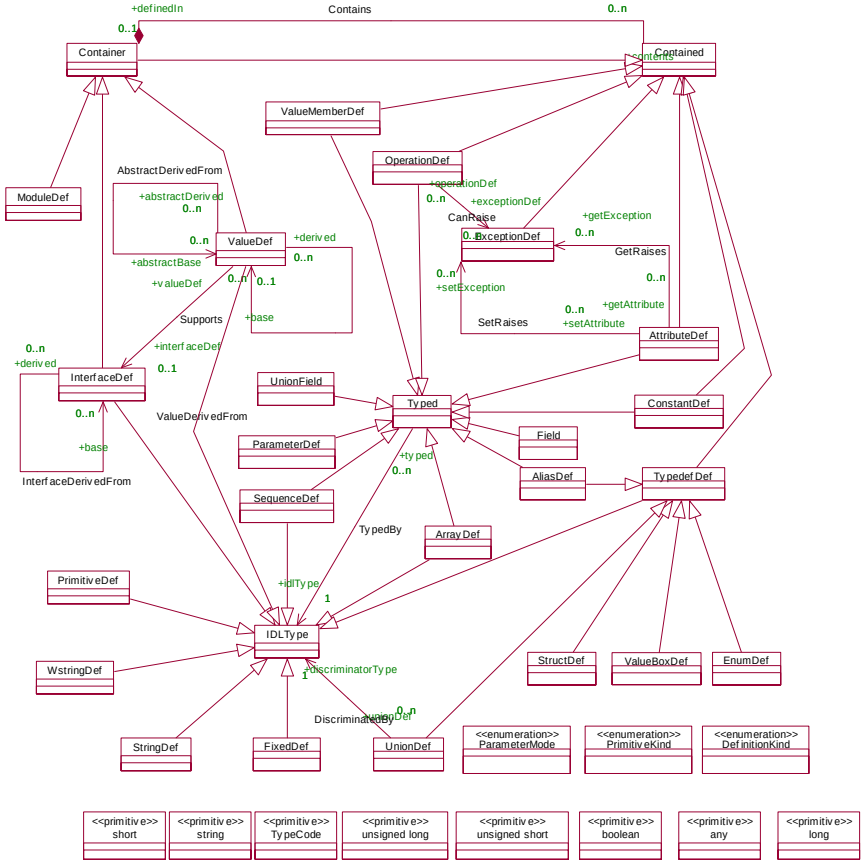

Figure 3 - Excerpt of the BaseIDL package

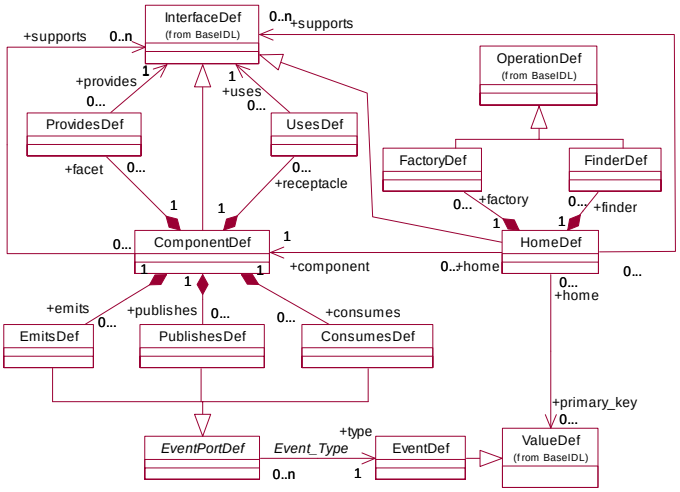

Figure 4 - Excerpt of the ComponentIDL package