Formal Definition of Metrics upon the CORBA Component Model

Miguel Goulão[1], Fernando Brito e Abreu[1]

[1] QUASAR Research Group, Departamento de Informática, FCT/UNL, Portugal
{miguel.goulao|fba}@di.fct.unl.pt
http://ctp.di.fct.unl.pt/QUASAR/

Abstract. *Objective:* In this paper we present a formalization of the definition of metrics to assess quality attributes of CORBA components and assemblies. The focus is on the formalization technique, rather than on the evaluation of the metrics themselves.
Method: We represent 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. With these clauses we construct a formal and executable definition of the metrics.
Results: We demonstrate 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 components and 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.

1 Introduction

1.1 Aspects of component modeling

Current component models support the functional aspects of component wiring, at least to a certain extent. In simple component models, such as JavaBeans [1], components are accessed through their provided interfaces. In more sophisticated component models, such as the CORBA Component Model (CCM) [2], components may have multiple provided and required interfaces, synchronous and asynchronous operations, as well as the ability to publish and subscribe events.

There is usually less support for the specification of non-functional properties of components than for functional ones. For instance, the new UML 2.0 standard [3, 4] includes constructs for representing several of the above-mentioned component wiring mechanisms, but none for the representation of non-functional properties. The latter are defined as a UML profile [5], rather than as a part of the core language. This may be seen as a limitation in practice, since modeling tools are less likely to support UML extensions on their standard distribution.

1.2 Specificity of CBSE metrics

One of the goals of Component-Based Software Engineering (CBSE) is achieving predictability of system quality based on the quality attributes of the constituent components [6]. Currently, developers are unable to make such predictions in an automated fashion. Difficulties hampering this task include determining which properties would be useful to component developers and users, how to combine the properties of individual components to predict the properties of assemblies, how to measure such properties, and how to present this information to component users. Current component models used in industry are not *prediction-enabled*, although this is an active topic of research [7, 8]. The term *"prediction-enabled"* is mostly associated with run-time quality attributes, with a focus on those that have objective definitions (e.g. latency) rather than subjective ones (e.g. usability and maintainability). The latter should also be assessed.

Quality attributes can be classified within a quality model. As noted by some authors [9, 10], specific quality models must be developed for CBSE, to address the focus shift from structured, or object-oriented development to Component-Based Development (CBD). For instance, a component user is more concerned with the complexity involved in selecting and composing components than with the implementation details of each component. From his perspective, components are black-boxes whose evolution he does not control. Existing metrics for structured and object oriented development are not well suited for CBSE, since they are mainly concerned with internal complexity of the components. Several traditional complexity metrics such as the McCabe's [11] and Halstead's [12] ones are useless to a component user, since their computation requires the access to the implementation details, which are not available to component users, due to the black-box nature of components.

1.3 Open problems in metrics for CBSE

In a recent survey, we identified several proposals for the quantitative assessment of components and assemblies, based upon their functional properties [13]. We observed several recurrent problems in those proposals, which are also common in metrics proposals for other purposes, such as OO design evaluation:

(1) lack of a quality framework – occurs when a metrics definition is not framed by a particular quality model;

(2) lack of an ontology – occurs when the architectural concepts to be quantified, either of functional or non-functional nature, are not clearly defined, namely in their interrelationships;

(3) lack of an adequate formalism – sometimes, metrics are defined with a formalism that requires a strong mathematical background, which is often not held by practitioners. This limitation hampers metrics usability. On the other hand, it is also common to find metrics defined using natural language in the literature. These definitions tend to be subjective to some extent, thus jeopardizing the correctness and comparability of metrics collection;

(4) lack of computational support – occurs when metrics proponents do not produce tools for metrics collection, or when they do not make them available to other researchers and practitioners;
(5) lack of flexibility – occurs when metrics collection tools are available, but they are either proprietary, or, if they are open source, the metrics definitions are somehow obscurely tangled in the code. The latter hampers the assessment of the correction of their algorithm, as well as the ability to modify them (a recurrent need in experimental software engineering projects);
(6) 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.

One possible solution to address (1) is using the Goal-Question-Metric (GQM) approach [14]. This approach starts with the definition of goals, which in this case should be aligned with the needs of the CBD actors. These goals should also be defined within the scope of a suitable quality model. Questions are formulated in order to assess the level of achievement of the defined goals. Finally, metrics are defined to provide the information required for answering the formulated questions. In other words, metrics definitions should always be preceded by clear objectives.

In this paper, we present an approach to mitigate the remaining problems (2 to 6). 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 JavaBeans, COM or .Net. The CCM has a standard metamodel where the architectural concepts and their interrelationships become more intelligible. We define Object Constraint Language (OCL) expressions upon the metamodel, to specify and collect the metrics, thus solving problem (2). OCL combines formality with a syntax easily understood by practitioners familiar with OO development and is therefore an adequate formalism to help solving problem (3). We circumvent problem (4) by using executable metrics definitions. Indeed, OCL expressions can be evaluated with several existing UML tools, upon a given metamodel. Some of those tools only allow OCL syntax and model conformance checking. Others, such as the USE tool [15], allow loading metadata (objects and links representing model instances and their interrelationships) and evaluate the result of OCL expressions upon that workload. Using OCL expressions makes metrics definitions open and clearly separated from other code, thus solving problem (5). 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 (6).

1.4 Paper organization

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 identi-

fied in the introduction, to stress how it helps mitigating those problems. Conclusions are presented in section 7.

2 Related Work

Recently, researchers have tried to establish requisites and guidelines for CBD metrics, both concerning individual components [16] and component assemblies [17]. 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.

Other proposals have contributed to the evaluation of component interfaces and dependencies [18-20]. They 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) [21]. Hoek *et al.* proposed metrics to assess service utilization in component assemblies [22].

The previous proposals include, to some extent, informal specifications. Frequently, these contain ambiguous information, particularly in what concerns the basic counting elements used in the metrics formulae. Different interpretations of such elements may lead to different results, when computing the metrics' values. This is a major drawback, when comparing the results from independent metrics collection experiments.

In this paper we present a collection of metrics taken from some of these proposals, formalized with OCL upon the CCM metamodel. Such a formalization implies an explicit interpretation of the original (informal) metrics definition and provides an executable specification for them. We build up on some previous contributions of our team. In [23] we formalized a metrics set for component reusability assessment [20]. We used that formalization to conduct an independent validation experiment on the same metrics set in [24], 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 technique, originally proposed in [25], has also been used in other contexts, therefore with different metamodels. In [26], we formalized well-known OO design metrics upon a OCL functions library named FLAME, aimed at helping metrics extraction in UML 1.x models [27]. More recently, we have successfully applied the same technique with object-relational database schema metrics [28].

3 CORBA components

The CCM [2, 29, 30] 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 and required interfaces (also known as facets and receptacles) as well as the ability to publish and subscribe events (by means of event sources and sinks). Components also offer navigation and introspection capabilities. The CCM also has support for distribution and Quality of Service (QoS) properties.

The CCM specification includes a Meta Object Facility–compliant metamodel [31], where the CCM modeling elements are defined precisely. The metamodel includes three packages (see Fig. 1). The `BaseIDL` package contains the modeling elements concerning the CORBA Interface Description Language (IDL). `BaseIDL` is extended by `ComponentIDL`, to add the component specific constructs. Finally, `ComponentIDL` is extended by the `CIF` package, with the model elements of the Component Implementation Framework, which include the definitions relating to the component life cycle.

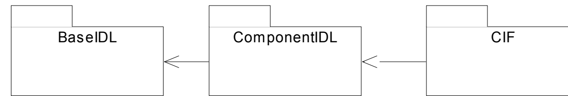

Fig. 1. CCM metamodel packages

The metrics formalizations presented throughout this paper use only metamodel abstractions defined in the `BaseIDL` and `ComponentIDL` packages, as we were able to find all the required abstractions in them. Nevertheless, the `CIF` package abstractions could also be used, with an extended metrics set.

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, with OCL expressions, we can collect information on the assembly we want to analyze. Those expressions provide us the distilled information required for our metrics computation. Consider a small excerpt of the `BaseIDL` package of the CCM metamodel, where the relationship between `InterfaceDef` and `OperationDef` is specified.

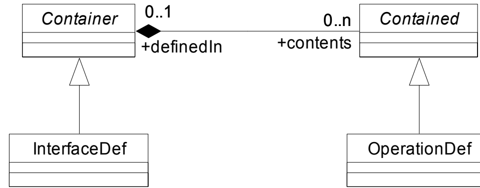

Fig. 2. Interfaces and Operations, in the CCM metamodel

Let us assume we wish to define a metric to count the operations defined within an interface. We can 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 cardinality of 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 the metrics formalizations, we built a library of reusable OCL functions [32] that includes several utility functions such as these two. We reuse some of those utility functions in the formalizations presented in this paper. Fig. 3 presents the classes where we defined the reusable OCL functions, represented here through their signature.

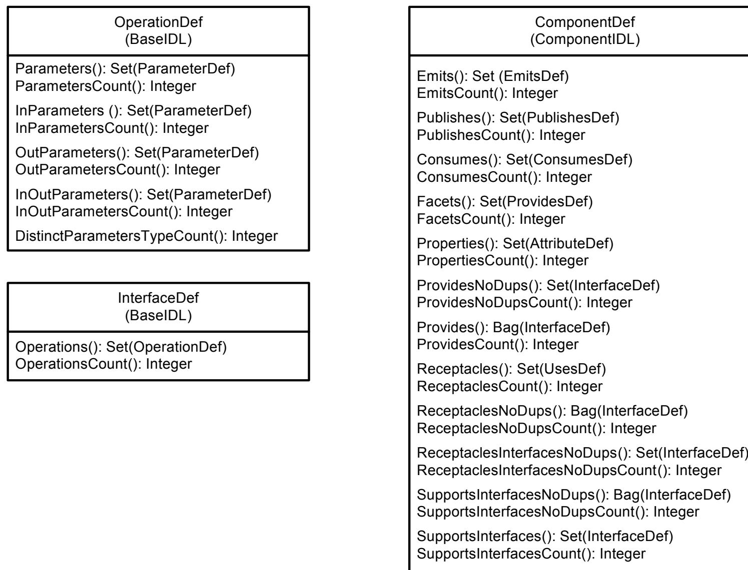

Fig. 3. Reusable OCL functions for CBD metrics collection

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 *name and original specification* ***(i)***, keeping the notation used by their proponents (thus illustrating the variability of notations commonly used in metrics definitions), and the *metric's rationale* ***(ii)***, in their proponents' view. These are followed by *considerations and assumptions made during the formalization process* and, finally, the *formalization of the metric in OCL* ***(iii)***. The latter may include auxiliary functions. The `AuxilaryFunction` typeface is used to identify these functions. The final expression of the metric is shown within a bounding box.

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 [18]. These metrics aim to assess the understandability of a component interface. Boxall and Araban assume that understandability has a positive influence on the reusability of software components. As such, a higher understandability leads to a higher reusability of the components.

Arguments per Procedure (APP)

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

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

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 [33]. The number of chunks of information in the procedure definition (in this case, its arguments) should be limited. Boxall and Araban suggest that an increased number of arguments reduces the interface's understandability and, therefore, its reusability.

(iii) The original proposal of this metric uses C/C++ component interfaces to illustrate the metric definition. 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, as these will be invisible to component users.

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 operations 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 metric different than the one proposed by Boxall and Araban. To be precise in our formalization, 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 cardinality 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)

(i) The number of distinct arguments in publicly declared procedures (*DAC*) is defined as in (Eq. 2). Its percentage on the component interface (*DAR*) is 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. However, unlike

DAC, *DAR* is immune to the size of the interface, because its value corresponds to *DAR*, when normalized by n_a.

(iii) Boxall and Araban consider a parameter as a duplicate of another 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)

(i) The *ARS* aims to account for the repetitiveness of arguments in a component's interface (Eq. 4). In other words, it is used for measuring the consistency of the naming and typing of arguments within the publicly declared procedures of an operation.

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

where:

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

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

n_a = argument count in the interface

(ii) The rationale for this metric is that the repetitiveness of arguments increases the interface's understandability, and, therefore, the component's reusability. According to Boxall and Araban, $|a|$ is squared in this definition to create a bias that favors consistent arguments definitions in the interface. Interfaces with a higher ARS *"will tend to be dominated by fewer distinct arguments which are repeated more often"*.

(iii) We define two 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()
```

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 [21].

Component Packing Density (CPD)

(i) The *CPD* represents the average number of constituents of a given type (e.g. lines of code, interfaces, classes, modules) in a component (Eq. 5).

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

where:
constituent_type = type of the constituents whose density is being assessed
#<*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 suggested, 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.

(iii) We exemplify a possible formalization of this metric, considering the number of operations in the provided interfaces as the constituent type, which is only one of the possibilities. The **`CPD`** function formalizes the metric definition in OCL. To compute the number of operations made available by each component, we reuse the auxiliary function `ProvidedOperationsCount`, which we presented earlier (in the formalization of Eq. 1). Finally, the formalization of this metric is performed using a different context than the previous ones. CPD is computed within the scope of a module (represented by **`ModuleDef`**, in the metamodel), rather than the one of an individual component.

```
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 so far in this paper focus mainly on the provided interfaces of components. In what concerns their formal definition, adapting these metrics to the required interfaces of components would be straightforward. Rather than using the `ProvidedOperations` auxiliary function, we would replace it by the `RequiredOperations` function. This illustrates the flexibility of the formalization approach. Naturally, these adaptations would only make sense within the scope of a goal-driven extension of the metrics set, to cover quality attributes that are not being sufficiently assessed by these metrics.

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 assess 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), 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. In other words, a higher complexity leads to a lower understandability. The same applies to **`CPC`**. More configurable properties imply a higher complexity in configuring a component. On the other hand, they also increase the flexibility of its configuration. In this last example, stresses that sometimes we may find that the same metric may be useful for assessing conflicting quality attributes.

5 Metrics collection example

5.1 The elevator control system example

Consider the elevator control system depicted in Fig. 4, with 4 components: `Alarm`, `MotorsController`, `ElevatorsController` and `RequestManager`.

Fig. 4. The Elevator CCM assembly

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.

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

We computed the formalized metrics for the elevator example by loading the CCM metamodel with the metadata representing the elevator example, and then calculating the results of the OCL expressions presented in section 4.

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 example

Context	APP	DAC	DAR	ARS	EFI	EFO	CPC
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. It should be noted that the goal of this paper is not to validate these metrics, but rather to show how they can be formalized with OCL upon the CCM metamodel. The elevator toy example is deliberately simple, to illustrate the metrics computation. With a real-world example, the manual collection of these metrics would require too much effort and be an error-prone task. Automated metrics collection is essential, if they are to be used by practitioners.

To illustrate the metrics computation, consider the `MotorsController` component. The following OCL expressions show the partial results of the function calls involved in the APP's metric computation. For simplicity, let us assume that the meta-objects have the same name as the concepts they represent, preceded by an underscore (e.g. the motor controller component is represented by a meta-object called `_MotorsController`, the `Off` operation is represented by the meta-object `_Off`, and so on).

```
_MotorsController.ProvidedOperations()= {_On, _Off, _Up, _Down,
                                          _NewSpeed, _Stop}
_MotorsController.ProvidedOperationsCount()= 6
_MotorsController.NA()= 9
_MotorsController.NP()= 6

_MotorsController.APP()= 1.5
```

6 Discussion

This section enumerates the six problems identified in section 1.3 and addresses how our approach helps dealing with them.

6.1 Quality framework

Without a clear notion of the quality attributes we wish to assess and the criteria we will use to interpret the metrics values, it is not possible to analyze the results. 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.

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. In this case, the CCM metamodel was used as an ontology.

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 these 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 without 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 Computational support

Most of the computational support required for collecting metrics defined in OCL is either publicly available, or can be built with relatively low effort. The core of the computational support consists of an OCL-enabled UML tool (e.g. the USE tool), with the ability to load a metamodel (as a class model) and create instances of those models (e.g., using object diagrams). In this case, we need to load the CCM metamodel and populate it with the appropriate meta-data, representing the CORBA components we want to evaluate. This metamodel instrumentation can be done manually in several UML tools by creating a meta-object diagram. However, it is more practical to develop a script that generates the meta-data from the original component's specifications. A detailed description of a similar architecture can be found in [34], in that case applied to UML 2.0 components.

6.5 Flexibility

By specifying the metrics definitions with OCL we have completely removed the code tangling between the metrics definitions and the tool computing the metrics. The metrics definitions are loaded in the UML tool just as any other OCL expression. Tailoring the metrics set to one's specific needs is, then, a matter of writing new OCL functions, similar to those presented in this paper.

6.6 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 in sections 6.1 through 6.5, it should become clear that the ideal conditions for independent scrutiny of these metrics were not present in their original definitions. Several plausible interpretations could be provided for each definition, and no tool support was available to collect them. These conditions hampered experimental replicability. By using the approach presented in this paper, independent validation efforts can be carried out, without jeopardizing the comparability of results.

7. Conclusions

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 the mitigation of recurrent problems with metrics definitions (lack of a quality framework, lack of an ontology, inadequate specification formalism, computational support, flexibility, and insufficient validation). 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 one of three essays from our team on the formalization of metrics sets for CBSE proposed by other authors. Both this paper and [34] focus on metrics applicable to components in isolation, using different metamodels (CCM and UML 2.0, respectively), to illustrate the expressiveness of the formalization approach. In [35] we present a similar formalization essay focused on component assemblies, using the CCM metamodel. 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. They can 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), allowing the computation of estimates on deployment and evolution costs.

References

1. Matena, V. and Hapner, M., *Enterprise JavaBeans Specification 1.1*, Sun MicroSystems, Inc., (1999)
2. OMG, *CORBA Components - Version 3.0*, Specification, Object Management Group Inc., formal/02-06-65, June. (2002)
3. OMG, *UML 2.0 Infrastructure Final Adopted Specification*, Object Management Group, Inc., ptc/03-09-15, September. (2003)
4. OMG, *UML 2.0 Superstructure Final Adopted Specification*, Object Management Group Inc., ptc/03-08-02, August. (2003)
5. OMG, *UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanics*, OMG Adopted Specification, Object Management Group Inc., ptc/04-09-01, September. (2004)
6. Crnkovic, I., Schmidt, H., Stafford, J.A., and Wallnau, K.: 6th ICSE Workshop on Component-Based Software Engineering: Automated Reasoning and Prediction. *ACM SIGSOFT Software Engineering Notes*. Vol. 29 (3): p. 1-7, (2004).
7. Wallnau, K., *Volume III: A Technology for Predictable Assembly from Certifiable Components*, Technical Report, Carnegie Mellon, Software Engineering Institute, CMU/SEI-2003-TR-009, April. (2003)
8. Larsson, M., *Predicting Quality Attributes in Component-based Software Systems*, PhD, Mälardalen University, (2004)
9. Bertoa, M. and Vallecillo, A. Quality Attributes for COTS Components. *In Proceedings of the 6th International Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE'2002)*. Málaga, Spain. (2002)
10. Simão, R.P.S. and Belchior, A.D., Quality Characteristics for Software Components: Hierarchy and Quality Guides, in *Component-Based Software Quality: Methods and Techniques*, Cechich, A., Piattini, M., and Vallecillo, A., Editors, Springer. p. 184-206. (2003).
11. McCabe, T.: A Complexity Measure. *IEEE Transactions on Software Engineering*. Vol. 2 (4): p. 308-320, (1976).
12. Halstead, M.: *Elements of Software Science*. Operating and Programming Systems. New York, EUA: Elsevier Computer Science Library / North-Holland1977).
13. Goulão, M. and Abreu, F.B. Software Components Evaluation: an Overview. *In Proceedings of the 5ª Conferência da APSI*. Lisbon. (2004)
14. Basili, V.R., Caldiera, G., and Rombach, D.H., Goal Question Metric Paradigm, in *Encyclopedia of Software Engineering*, Marciniak, J.J., Editor John Wiley & Sons. p. 469-476. (1994).
15. Richters, M.: A UML-based Specification Environment, University of Bremen: http://www.db.informatik.uni-bremen.de/projects/USE (2001)
16. Gill, N.S. and Grover, P.S.: Component-Based Measurement: Few Useful Guidelines. *ACM SIGSOFT Software Engineering Notes*. Vol. 28 (6): p. 4-4, (2003).
17. Sedigh-Ali, S., Ghafoor, A., and Paul, R.A.: Software Engineering Metrics for COTS-Based Systems. *IEEE Computer*. Vol., (2001).

18. Boxall, M.A.S. and Araban, S. Interface Metrics for Reusability Analysis of Components. *In Proceedings of the Australian Software Engineering Conference (ASWEC'2004)*. Melbourne, Australia: IEEE Computer Society. p. 40-51. (2004)
19. Gill, N.S. and Grover, P.S.: Few Important Considerations for Deriving Interface Complexity Metric for Component-Based Software. *Software Engineering Notes*. Vol. 29 (2): p. 4-4, (2004).
20. Washizaki, H., Yamamoto, H., and Fukazawa, Y. A Metrics Suite for Measuring Reusability of Software Components. *In Proceedings of the 9th IEEE International Software Metrics Symposium (METRICS 2003)*. Sydney, Australia: IEEE Computer Society. (2003)
21. Narasimhan, V.L. and Hendradjaya, B. A New Suite of Metrics for the Integration of Software Components. *In Proceedings of the The First International Workshop on Object Systems and Software Architectures (WOSSA'2004)*. South Australia, Australia: The University of Adelaide. (2004)
22. Hoek, A.v.d., Dincel, E., and Medvidovic, N. Using Service Utilization Metrics to Assess and Improve Product Line Architectures. *In Proceedings of the 9th IEEE International Software Metrics Symposium (Metrics'2003)*. Sydney, Australia: IEEE Computer Society Press. (2003)
23. Goulão, M. and Abreu, F.B. Formalizing Metrics for COTS. *In Proceedings of the International Workshop on Models and Processess for the Evaluation of COTS Components (MPEC 2004) at ICSE 2004*. Edimburgh, Scotland: IEE. p. 37-40. (2004)
24. Goulão, M. and Abreu, F.B. Cross-Validation of a Component Metrics Suite. *In Proceedings of the IX Jornadas de Ingeniería del Software y Bases de Datos*. Málaga, Spain. (2004)
25. Abreu, F.B., *Using OCL to formalize object oriented metrics definitions*, Technical Report, INESC, ES007/2001, May. (2001)
26. Baroni, A.L. and Abreu, F.B. Formalizing Object-Oriented Design Metrics upon the UML Meta-Model. *In Proceedings of the Brazilian Symposium on Software Engineering*. Gramado - RS, Brazil. (2002)
27. Baroni, A.L. and Abreu, F.B. A Formal Library for Aiding Metrics Extraction. *In Proceedings of the International Workshop on Object-Oriented Re-Engineering at ECOOP'2003*. Darmstadt, Germany. (2003)
28. Baroni, A.L., Calero, C., Piattini, M., and Abreu, F.B. A Formal Definition for Object-Relational Database Metrics. *In Proceedings of the 7th International Conference on Enterprise Information System*. Miami, USA. (2005)
29. Wang, N., Schmidt, D.C., and O'Ryan, C., Overview of the CORBA Component Model, in *Component-Based Software Engineering: Putting the Pieces Together*, Heineman, G.T. and Councill, W.T., Editors, Addison-Wesley Publishing Company. p. 557-571. (2001).
30. Estublier, J. and Favre, J.-M., Component Models and Technology, in *Building Reliable Component-Based Software Systems*, Crnkovic, I. and Larsson, M., Editors, Artech House. p. 57-86. (2002).
31. OMG, *Meta Object Facility (MOF) Specification (Version 1.4)*, Object Management Group, April. (2002)
32. Goulão, M.: OCL library for metrics collection in CBD using the CCM: http://ctp.di.fct.unl.pt/QUASAR/resources/DataFiles/CCMEA.use (2005)
33. Miller, G.A.: The Magical Number Seven, Plus or Minus Two : Some limits in our Capacity for Processing Information. *The Psychological Review*. Vol. 63: p. 81-97, (1956).
34. Goulão, M. and Abreu, F.B.: Validação Cruzada de Métricas para Componentes. *IEEE Transactions Latin America*. Vol. 3 (1), (2005).
35. Goulão, M. and Abreu, F.B. Composition Assessment Metrics for CBSE. *In Proceedings of the 31st Euromicro Conference - Component-Based Software Engineering Track*. Porto, Portugal: IEEE Computer Society. (2005)